SISTEMAS OPERATIVOS I - Práctica 4 (con soluciones)
Grado en Ingeniería Informática - Escuela Superior de Informática (UCLM))

1. Actividades de Laboratorio

Escriba un programa C estándar para cada uno de los distintos enunciados con la funcionalidad indicada **utilizando las llamadas al sistema UNIX**. Salvo que se especifique lo contrario, se entenderá que la entrada y salida del programa corresponderá a la entrada estándar y salida estándar del terminal y puede realizarse usando la biblioteca estándar C.

1. Construir un programa llamado **escudo1** que ejecute un bucle sin fin y responda a la señal indicada en su argumento (**escudo1** <*número*>) ignorando la señal. Probar su ejecución desde el shell utilizando el programa de sistema **kill** y comprobar la diferencia de comportamiento del programa según la señal recibida. Ejecute el programa con la señal SIGINT e intente abortar el programa pulsando la combinación de teclas `<Ctrl-C>`. Compruebe el comportamiento del programa con la señal SIGKILL. Se aconseja consultar la página del manual correspondiente a la llamada al sistema *signal*.

```c
/* escudo1.c - Ejecución de un bucle sin fin ignorando la señal indicada */
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

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

  /* Tratamiento de la línea de órdenes */
  if (argc != 2) {
    fprintf(stderr, "Error en el número de argumentos\n");
    return EXIT_FAILURE;
  }
  if ((num = atoi(argv[1])) <= 0) {
    fprintf(stderr, "Error en el argumento de la señal\n");
    return EXIT_FAILURE;
  }

  /* Tratamiento de la señal */
  if (signal(num , SIG_IGN) == SIG_ERR) { /* Ignorar la señal */
    fprintf(stderr, "Error en la manipulación de la señal\n");
    return EXIT_FAILURE;
  }
  while (1) ; /* Bucle infinito */
  return EXIT_SUCCESS;
}
```

2. Construir un programa llamado **escudo2** que ejecute un bucle sin fin y responda a la señal indicada en su argumento (**escudo2** <*número*>) escribiendo por la salida estándar el mensaje "Recibida la señal <número de la señal recibida>" y terminando. Probar su ejecución desde el shell. Se aconseja consultar las páginas del manual correspondientes a las llamadas al sistema *signal* y *_exit*.

```c
/* escudo2.c - Ejecución de un bucle sin fin e impresión de la señal recibida
   y fin del proceso */
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

/* Manejador: Manejador de la señal */
```

```c
void Manejador(int num)
{
  printf("Recibida la señal %d\n", num);
  _exit(EXIT_SUCCESS); /* Llamada al sistema de fin de proceso */
}

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

  /* Tratamiento de la línea de órdenes */
  if (argc != 2) {
    fprintf(stderr, "Error en el número de argumentos\n");
    return EXIT_FAILURE;
  }
  if ((num = atoi(argv[1])) <= 0) {
    fprintf(stderr, "Error en el argumento de la señal\n");
    return EXIT_FAILURE;
  }

  /* Tratamiento de la señal */
  if (signal(num, Manejador) == SIG_ERR) { /* Instalar manejador de la señal */
    fprintf(stderr, "Error en la manipulación de la señal\n");
    return EXIT_FAILURE;
  }
  while (1) ; /* Bucle infinito */
}
```

3. Construir un programa llamado **escudo3** que ejecute un bucle sin fin y responda a la señal indicada en su argumento (**escudo3** <*número*>) escribiendo por la salida estándar el mensaje "Recibida la señal <número de la señal recibida>" y continúe la ejecución del bucle. Probar su ejecución desde el shell. Se aconseja consultar la página del manual correspondiente a la llamada al sistema *signal*.

```c
/* escudo3.c - Ejecución de un bucle sin fin e impresión de la señal recibida y
   continuación de la ejecución del proceso */
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

/* Manejador: Manejador de la señal */
void Manejador(int num)
{
  printf("Recibida la señal %d\n", num);
  if (signal(num, Manejador) == SIG_ERR) { /* Instalar manejador de la señal */
    fprintf(stderr, "Error en la manipulación de la señal\n");
    exit(EXIT_SUCCESS); /* Función del ANSI C que llama a _exit */
  }
}

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

  /* Tratamiento de la línea de órdenes */
  if (argc != 2) {
    fprintf(stderr, "Error en el número de argumentos\n");
    return EXIT_FAILURE;
  }
  if ((num = atoi(argv[1])) <= 0) {
    fprintf(stderr, "Error en el argumento de la señal\n");
```

```
29      return EXIT_FAILURE;
    }
31
    /* Tratamiento de la señal */
33  if (signal(num, Manejador) == SIG_ERR) { /* Instalar manejador de la señal */
      fprintf(stderr, "Error en la manipulación de la señal\n");
35    return 3;
    }
37  while (1) ; /* Bucle infinito */
}
```

4. Construir un programa llamado **alarma** que escriba cada n segundos en la salida estándar el mensaje "Alarma <número>" siendo <número>el número de mensajes de alarmas escritos. El programa no debe tener una espera activa y su sintaxis será **alarma** <*segundos de espera*>. Se aconseja consultar las páginas del manual correspondientes a las llamadas al sistema *signal, alarm* y *pause*.

```
/* alarma.c - Impresión de alarmas recibidas sin un bucle de espera activa */
2 #include <stdio.h>
#include <stdlib.h>
4 #include <signal.h>
#include <unistd.h>
6
/* Manejador: Manejador de la señal */
8 void Manejador(int num)
{
10  static int total=0; /* Variable estática con nº total de alarmas recibidas */

12  printf("Alarma %d\n", ++total);
    if (signal(num, Manejador) == SIG_ERR) { /* Instalar manejador de la señal */
14    fprintf(stderr, "Error en la manipulación de la señal\n");
      exit(EXIT_FAILURE);
16  }
}
18
int main(int argc, char *argv[])
20 {
  int segundos; /* Segundos de espera de la alarma */
22
  /* Tratamiento de la línea de órdenes */
24  if (argc != 2) {
      fprintf(stderr, "Error en el número de argumentos\n");
26    return EXIT_FAILURE;
    }
28  if ((segundos = atoi(argv[1])) <= 0) {
      fprintf(stderr, "Error en el argumento de segundos de espera\n");
30    return EXIT_FAILURE;
    }
32
  /* Tratamiento de la señal de alarma */
34  if (signal(SIGALRM, Manejador) == SIG_ERR) { /* Instalar manejador alarma */
      fprintf(stderr, "Error en la manipulación de la señal\n");
36    return EXIT_FAILURE;
    }
38  while (1) { /* Bucle infinito */
      alarm(segundos);
40    pause(); /* Espera pasiva de recepción de la alarma */
    }
42 }
```

5. Construir un programa llamado **mikill** que envíe una señal al proceso que se desee. La sintaxis del programa será **mikill** *<nº de la señal><pid del proceso destinatario>*. Probar la ejecución de los programas anteriores utilizando **mikill**. Se aconseja consultar la página del manual correspondiente a la llamada al sistema *kill*.

```
/* mikill.c - Envía la señal indicada a un proceso */
#define _POSIX_SOURCE /* Elimina el warning en declaración de kill() */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <signal.h>

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

  /* Tratamiento de la línea de órdenes */
  if (argc != 3) {
    fprintf(stderr, "Error en el número de argumentos\n");
    return EXIT_FAILURE;
  }
  if ((num = atoi(argv[1])) <= 0) {
    fprintf(stderr, "Error en el argumento de nº de la señal\n");
    return EXIT_FAILURE;
  }
  if ((pid = atoi(argv[2])) <= 0) {
    fprintf(stderr, "Error en el argumento de pid del proceso receptor\n");
    return EXIT_FAILURE;
  }

  /* Envío de la señal */
  if (kill(pid, num) != 0) {
    fprintf(stderr, "Error en el envío de la señal\n");
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}
```

6. Construir un programa llamado **entrada** que ejecute el programa indicado en la línea de ordenes con su entrada estándar redirigida al archivo propuesto. La sintaxis del programa será **entrada** *<archivo><programa><arg1><arg2>... <último argumento>*. Probar su ejecución desde el shell (por ejemplo, `entrada /etc/passwd wc -l -w`). Se aconseja consultar las páginas del manual correspondiente a las llamadas al sistema *execvp, open, close* y *dup*.

```
/* entrada.c - Ejecuta el programa indicado con la entrada estándar redirigida
   al archivo solicitado */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

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

  /* Tratamiento de la línea de órdenes */
  if (argc < 3) {
    fprintf(stderr, "Error en el número de argumentos\n");
```

```
17      return EXIT_FAILURE;
    }
19
    /* Redirección de la entrada estándar al archivo */
21  if ((fd = open(argv[1], O_RDONLY)) == -1) {
      fprintf(stderr, "Error en la apertura del archivo de datos\n");
23    return EXIT_FAILURE;
    }
25  close(0); /* Cierre de la entrada estándar actual */
    if (dup(fd) != 0) { /* Asignación de entrada estándar por duplicación */
27    fprintf(stderr, "Error en la duplicación del descriptor\n");
      return EXIT_FAILURE;
29  }
    execvp(argv[2], &argv[2]); /* Ejecución del programa solicitado */
31  fprintf(stderr, "Error en la ejecución del programa\n");
    return EXIT_FAILURE;
33 }
```

7. Basado en el programa anterior construir otro programa llamado **salida** que redirija la salida estándar en vez de la entrada estándar.

```
1  /* salida.c - Ejecuta el programa indicado con la entrada estándar redirigida
       al archivo solicitado */
3  #include <stdio.h>
   #include <stdlib.h>
5  #include <sys/types.h>
   #include <sys/stat.h>
7  #include <fcntl.h>
   #include <unistd.h>
9
   #define PERMISOS 0644 /* Permisos en la creación del archivo */
11
   int main(int argc, char *argv[])
13 {
     int fd;
15
     /* Tratamiento de la línea de órdenes */
17   if (argc < 3) {
       fprintf(stderr, "Error en el número de argumentos\n");
19     return EXIT_FAILURE;
     }
21
     /* Redirección de la salida estándar al archivo, creando dicho archivo si
23      no existe (O_CREAT) y truncándolo si ya existe (O_TRUNC) */
     if ((fd = open(argv[1], O_WRONLY|O_CREAT|O_TRUNC, PERMISOS)) == -1) {
25     fprintf(stderr, "Error en la apertura del archivo de datos\n");
       return EXIT_FAILURE;
27   }
     close(1); /* Cierre de la salida estándar actual */
29   if (dup(fd) != 1) { /* Asignación de salida estándar por duplicación */
       fprintf(stderr, "Error en la duplicación del descriptor\n");
31     return EXIT_FAILURE;
     }
33   execvp(argv[2], &argv[2]); /* Ejecución del programa solicitado */
     fprintf(stderr, "Error en la ejecución del programa\n");
35   return EXIT_FAILURE;
   }
```

8. Construir un programa llamado **tuberia** que cree dos procesos hijos que ejecuten dos programas indicados en la línea de órdenes de tal forma que la salida estándar del primero se conecte a la entrada

estándar del segundo. El programa **tuberia** deberá esperar a la terminación de los dos procesos hijos y escribirá una línea en la salida estándar por cada programa con el siguiente formato "`Fin proceso <pid>:<código>`" siendo <pid> el identificador del proceso que ha terminado y <código> su código de terminación. La sintaxis del programa será **tuberia** *<programa1><programa2>*. Probar su ejecución desde el shell (por ejemplo, `tuberia ls sort`). Se aconseja consultar las páginas del manual correspondiente a las llamadas al sistema *fork, execlp, pipe, close, dup, wait* y/o *waitpid*.

```c
/* tuberia.c -  Ejecuta dos programas con la salida estándar del primer
   programa conectada a la entrada estándar del segundo programa. El proceso
   tuberia espera la terminación de los dos procesos hijos e imprime su pid y
   código de terminación */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <wait.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
  int fd[2]; /* Descriptores de la tubería de comunicación */
  int pid; /* PID de un proceso hijo */
  int status; /* Código de terminación de un proceso hijo */
  int i;

  /* Tratamiento de la línea de órdenes */
  if (argc != 3) {
    fprintf(stderr, "Error en el número de argumentos\n");
    return EXIT_FAILURE;
  }

  /* Creación de la tubería de comunicación */
  if (pipe(fd) != 0) {
    fprintf(stderr, "Error en la creación de la tubería\n");
    return EXIT_FAILURE;
  }
  /* Ejecución del primer programa */
  switch (fork()) {
    case -1 : fprintf(stderr, "Error en la creación del proceso hijo\n");
              return EXIT_FAILURE;
    case 0 :  /* Ejecución por el proceso hijo */
              close(1); /* Cierre de la salida estándar actual */
              if (dup(fd[1]) != 1) { /* Asignación de salida estándar */
                fprintf(stderr, "Error en la duplicación del descriptor\n");
                return EXIT_FAILURE;
              }
              execlp(argv[1], argv[1], NULL);
              fprintf(stderr, "Error en la ejecución de %s", argv[1]);
              return EXIT_FAILURE;
    default : /* Ejecución por el proceso padre */
              /* Se cierra el lado escritor de la tubería para que el segundo
                 programa reciba un EOF con el cierre de la salida estándar
                 del primer programa */
              close(fd[1]);
              break;
  }
  /* Ejecución del segundo programa */
  switch (fork()) {
    case -1 : fprintf(stderr, "Error en la creación del proceso hijo\n");
              return EXIT_FAILURE;
    case 0  : /* Ejecución por el proceso hijo */
              close(0); /* Cierre de la entrada estándar actual */
              if (dup(fd[0]) != 0) { /* Asignación de salida estándar */
```

```
56                  fprintf(stderr, "Error en la duplicación del descriptor\n");
                    return EXIT_FAILURE;
58                }
                  execlp(argv[2], argv[2], NULL);
60                fprintf(stderr, "Error en la ejecución de %s\n", argv[2]);
                  return EXIT_FAILURE;
62      default : /* Ejecución por el proceso padre */
                  break;
64    }
      /* Espera a la terminación de los procesos hijos */
66    for (i=0; i<2; i++) {
        if ((pid = wait(&status)) == -1) {
68        fprintf(stderr, "Error en espera de terminación de un proceso hijo\n");
          continue;
70      }
        printf("Fin proceso %d:%d\n", pid, status);
72    }
      return EXIT_SUCCESS;
74  }
```

9. Construir un programa llamado **consumidor** que copie en la salida estándar el contenido de una tubería cuyo nombre recibe como argumento. El programa debe continuar esperando datos de la tubería aunque haya leído un fin de archivo. La sintaxis del programa será **consumidor** *<nombre de la tubería>*. Crear en el directorio de trabajo actual una tubería con nombre **ejemplopipe** mediante el programa de sistema **mkfifo**. Probar la ejecución del programa **consumidor** y escribir con otro programa en la tubería (por ejemplo, con el programa de sistema **cat**). Compruebe que el programa **consumidor** sigue leyendo datos en sucesivas ejecuciones de los programas que escriben en la tubería. También compruebe que si el programa **consumidor** termina y los otros programas siguen escribiendo en la tubería, en la siguiente ejecución del programa **consumidor** se obtienen todos los datos escritos. El programa debe minimizar el tiempo del procesador invertido en los bucles de espera. Se aconseja consultar las páginas del manual correspondientes a las llamadas al sistema *open, read, write* y *sleep*.

```
  /* consumidor.c - Copia el contenido de la tuberia en la salida estándar
2    esperando de forma indefinida */
  #include <stdio.h>
4 #include <stdlib.h>
  #include <sys/types.h>
6 #include <sys/stat.h>
  #include <fcntl.h>
8 #include <unistd.h>

10 #define ESPERA 1 /* Segundos de espera en lectura de EOF en la tubería */
   #define MAXBUFFER 256 /* Tamaño del buffer de lectura/escritura */
12
   int main(int argc, char *argv[])
14 {
     char buffer[MAXBUFFER]; /* Buffer de lectura/escritura */
16   int fd; /* Descriptor asociado a la tubería */
     int n;
18
     /* Tratamiento de la línea de órdenes */
20   if (argc != 2) {
       fprintf(stderr, "Error en el número de argumentos\n");
22     return EXIT_FAILURE;
     }
24
     /* Apertura de la tubería con nombre */
26   if ((fd = open(argv[1], O_RDONLY)) == -1) {
       fprintf(stderr, "Error de apertura de la tubería\n");
28     return EXIT_FAILURE;
```

```
    }
30  /* Bucle infinito de lectura de la tubería y escritura en salida estándar.
       El bucle infinito se consigue con la condición while (n == 0) que se
32     recibe cuando algún proceso escritor de la tubería cierra su descriptor */
    while ((n = read(fd, buffer, sizeof(buffer))) >= 0) {
34    if (n == 0) sleep(ESPERA); /* Minimiza el uso de CPU en lectura EOF */
      else if (write(1, buffer, n) != n) { /* Escritura en la salida estándar */
36      fprintf(stderr, "Error de escritura\n");
        return EXIT_FAILURE;
38    }
    }
40  fprintf(stderr, "Error de lectura en la tubería\n");
    return EXIT_FAILURE;
42 }
```

10. Construir un programa llamado **mimkfifo** que cree una tubería con nombre. La sintaxis del programa será **mimkfifo**<*permisos en octal*><*nombre de la tubería*>. Probar la ejecución del programa anterior creando la tubería con **mimkfifo**. Se aconseja consultar las páginas del manual correspondientes a la llamada al sistema *mkfifo* y a la función de la biblioteca estándar *sscanf*.

```
  /* mimkfifo.c - Crea una tubería con nombre y permisos indicados */
2 #include <stdio.h>
  #include <stdlib.h>
4 #include <sys/types.h>
  #include <sys/stat.h>
6
  int main(int argc, char *argv[])
8 {
    unsigned int permisos;
10
    /* Tratamiento de la línea de órdenes */
12  if (argc != 3) {
      fprintf(stderr, "Error en el número de argumentos\n");
14    return EXIT_FAILURE;
    }
16  /* Conversión a entero en octal de los permisos almacenados en argv[1] */
    if (sscanf(argv[1], "%o", &permisos) != 1) {
18    fprintf(stderr, "Error en el argumento de permisos\n");
      return EXIT_FAILURE;
20  }

22  /* Creación de la tubería con nombre */
    if (mkfifo(argv[2], permisos) != 0) {
24    fprintf(stderr, "Error en creación de la tubería\n");
      return EXIT_FAILURE;
26  }
    return EXIT_SUCCESS;
28 }
```

11. Construir un programa denominado **padre** que cree un número determinado (definido en una constante) de procesos que ejecuten un código de programa denominado **hijo**. El código del programa **hijo** hará que cada proceso hijo duerma un número aleatorio de segundos entre 1 y un número máximo y, justo a continuación, finalizará. El número máximo de segundos será indicado por el programa **padre** a la hora de asignar un nuevo segmento de código a cada proceso hijo mediante la primitiva *execl*, siendo la responsabilidad de cada proceso hijo generar un número aleatorio (entre 1 y ese número máximo) que utilizará en la llamada a la primitiva *sleep* para dormir.

    La sintaxis para la ejecución de los programas será:

    a) **padre** <*número máximo segundos*>

b) **hijo** <*número máximo segundos*>

El proceso padre mostrará por la salida estándar dos mensajes:

1. Antes de finalizar el proceso: `[Proceso padre] finaliza`
2. Cuando el proceso padre conoce que un proceso hijo ha terminado siendo <pid> el identificador del proceso hijo finalizado: `[Proceso padre] el proceso <pid> ha terminado`

Asimismo el proceso padre puede finalizar su ejecución si el usuario pulsa la combinación de teclas `<Ctrl+C>`. En este caso, el proceso padre debe forzar la terminación de los procesos hijos vivos mediante la señal SIGINT. Cada proceso hijo mostrará por la salida estándar los dos mensajes siguientes, siendo <pid> su identificador de proceso y <n> el nº de segundos que el proceso hijo va a dormir:

1. Antes de dormir: `[Proceso <pid>] duerme <n> segundos`
2. Antes de finalizar el proceso: `[Proceso <pid>] finaliza`

Se aconseja consultar las páginas del manual correspondientes a las llamadas al sistema *fork, execl, getpid, wait* y/o *waitpid , signal, kill* y *sleep* y a las funciones de la biblioteca estándar *srand* y *rand*.

```c
/* hijo.c - Espera durmiendo el tiempo indicado y finaliza */
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main (int argc, char *argv[]) {
  int seg_espera; /* Segundos a esperar durmiendo */
  int seg_maximo; /* Nº máximo de segundos a dormir */

  /* Tratamiento de la línea de órdenes */
  if (argc != 2) {
    fprintf(stderr, "Error en el número de argumentos\n");
    return EXIT_FAILURE;
  }
  if ((seg_maximo = atoi(argv[1])) <= 0) {
    fprintf(stderr, "Error en el argumento de segundos\n");
    return EXIT_FAILURE;
  }

  /* Espera un nº de segundos aleatorio y finaliza */
  srand((int)getpid()); /* Inicializa la secuencia aleatoria */
  seg_espera = 1 + (rand() % seg_maximo);
  printf("[Proceso %d] duerme %d segundos\n", getpid(), seg_espera);
  sleep(seg_espera);
  printf("[Proceso %d] finaliza\n", getpid());

  return EXIT_SUCCESS;
}
```

```c
/* padre.c - Crea procesos hijos y espera su finalización */
#define _POSIX_SOURCE /* Elimina el warning en declaración de kill() */
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <wait.h>

#define NUM_HIJOS 5 /* Nº de procesos hijos a crear */
#define NOMBREHIJO "hijo" /* Nombre del programa hijo */

void FinalizarProcesos();
```

```
void Manejador(int num);

pid_t pids[NUM_HIJOS]; /* Tabla de procesos hijos */

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

  /* Tratamiento de la línea de órdenes */
  if (argc != 2) {
    fprintf(stderr, "Error en el número de argumentos\n");
    exit(EXIT_FAILURE);
  }

  /* Tratamiento de la señal */
  if (signal(SIGINT, Manejador) == SIG_ERR) {
    fprintf(stderr, "Error en la manipulación de la señal\n");
    exit(EXIT_FAILURE);
  }

  /* Creación de los procesos hijos */
  for (i = 0; i < NUM_HIJOS; i++)
    switch(pids[i] = fork()) { /* Se guarda el pid en la tabla de procesos */
    case -1 :
      fprintf(stderr, "Error en la creación del proceso hijo\n");
      FinalizarProcesos();
      break;
    case 0:
      if (execl(NOMBREHIJO, NOMBREHIJO, argv[1], NULL) == -1) {
        fprintf(stderr, "Error en la ejecución del proceso hijo\n");
        exit(EXIT_FAILURE);
      }
      break;
    }
  /* Espera a la finalización de los procesos hijos */
  for (i = 0; i < NUM_HIJOS; i++) {
    pid = wait(NULL);
    for (j = 0; j < NUM_HIJOS; j++) {
      if (pid == pids[j]) {
        printf("[Proceso padre] el proceso %d ha terminado\n", pid);
        pids[j] = 0; /* 0 para indicar que no existe el proceso */
      }
    }
  }
  printf("[Proceso padre] finaliza\n");
  return EXIT_SUCCESS;
}

/* FianlizarProcesos: Termina todos los procesos hijos vivos */
void FinalizarProcesos(void) {
  int i;

  for (i = 0; i < NUM_HIJOS; i++)
    if (pids[i] != 0) { /* Sólo se mata los procesos hijos vivos */
      if (kill(pids[i], SIGINT) == -1) {
        fprintf(stderr, "Error al enviar una señal\n");
      }
    }
}

/* Manejador: Manejador de la señal */
void Manejador(int num) {
```

```
    FinalizarProcesos();
77  printf("[Proceso padre] finaliza\n");
    exit(EXIT_SUCCESS);
79 }
```