Universidad de Murcia

Facultad de Informática

Departamento de Ingeniería y Tecnología de Computadores

Área de Arquitectura y Tecnología de Computadores

PRÁCTICAS DE SS.OO.

I.I./I.T.I. Sistemas/I.T.I. Gestión

Práctica 3 – Llamadas al Sistema en Linux

Enero de 2007

Índice

1. La llamada `fork` **3**
1.1. Ejemplo de uso de `fork` 3
1.2. Ejemplo de uso de `exec` 4
1.3. Ejemplo de uso de `fork` y `exec` conjuntamente 5

2. Listados **6**
2.1. fork_exec_wait.c 6
2.2. runcom.c 7
2.3. signal.c 8
2.4. sigaction.c 9
2.5. sigaction_chld.c 10
2.6. envia.c 12
2.7. alarma.c 13
2.8. pipe3men.c 15
2.9. pipecapa.c 16
2.10. pipe2com.c 17
2.11. fifo_esc.c 19
2.12. fifo_lee.c 21

3. PROCESOS **22**

4. COMUNICACIONES ENTRE PROCESOS. SEÑALES **23**

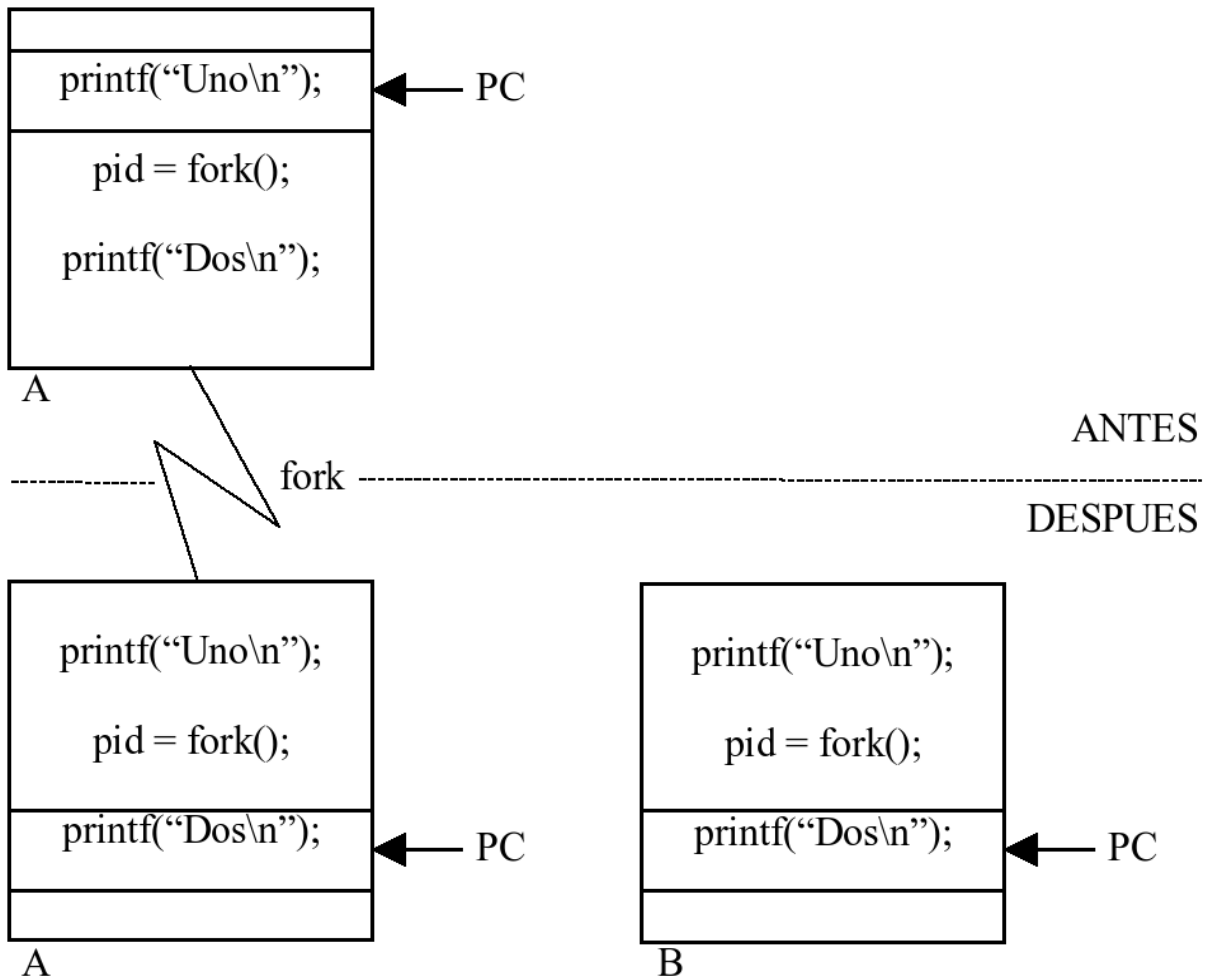
5. COMUNICACIONES ENTRE PROCESOS. TUBERÍAS **24**

6. COMUNICACIONES ENTRE PROCESOS. FIFO's **25**

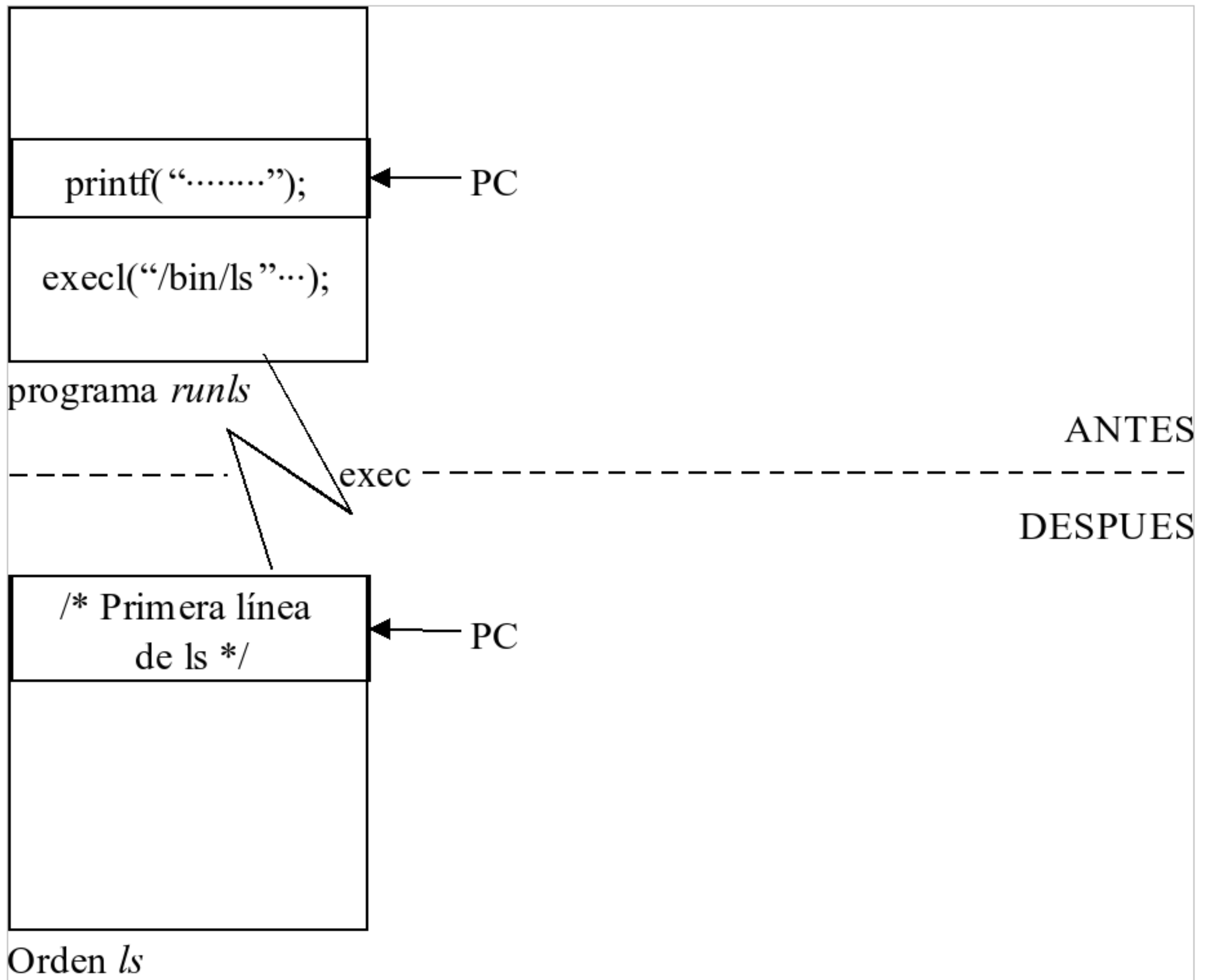
7. BIBLIOGRAFÍA **25**

1. La llamada fork

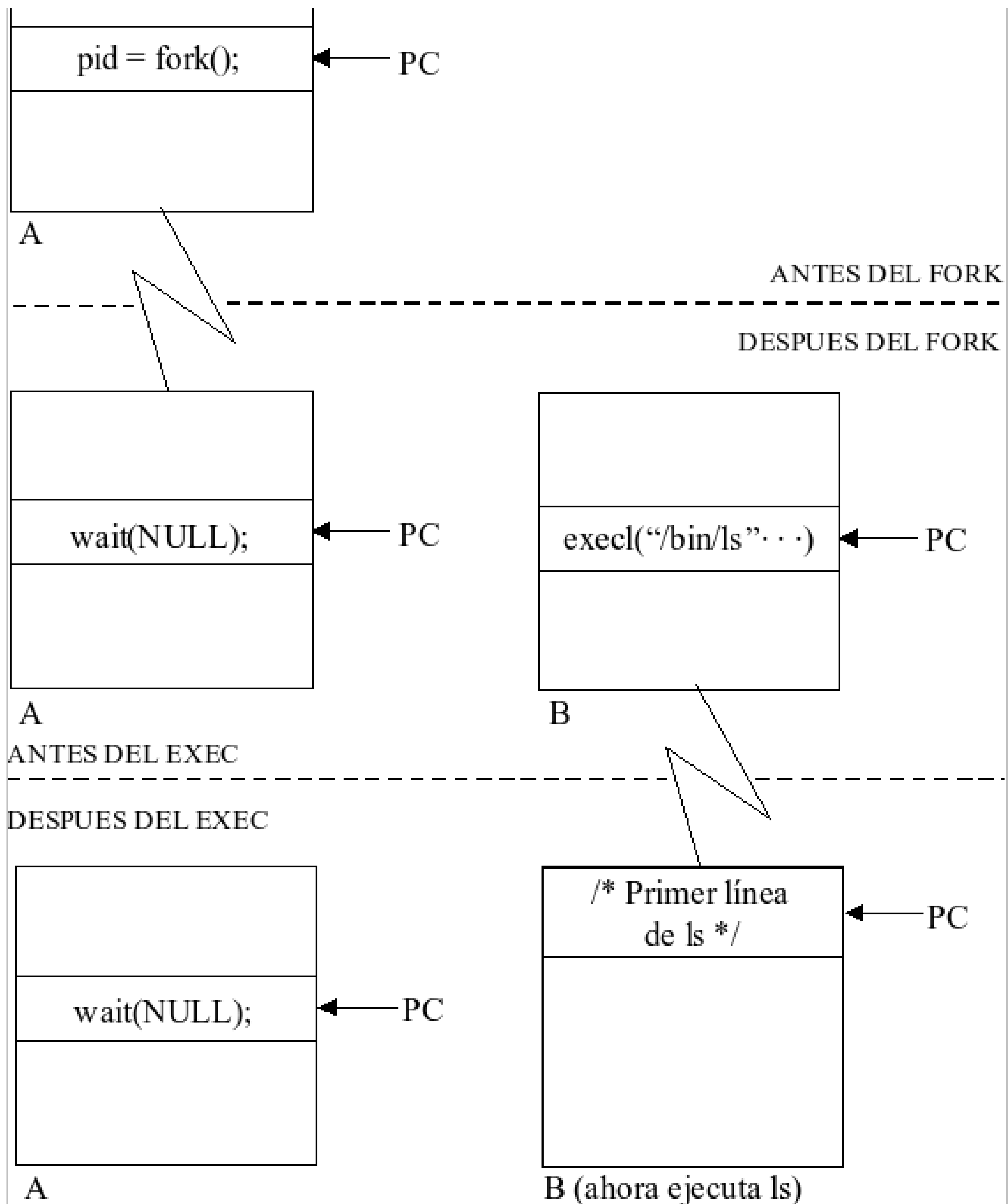
1.1. Ejemplo de uso de fork



1.2. Ejemplo de uso de `exec`



1.3. Ejemplo de uso de `fork` y `exec` conjuntamente



2. Listados

2.1. fork_exec_wait.c

```
1  /* El siguiente programa muestra el uso de fork, exec y wait... */

3  #include <stdio.h>
   #include <stdlib.h>
5  #include <unistd.h>

7  int main(void) {
       int pid;
9
       printf("Hasta aquí hay un único proceso...\n");
11     printf("Primera llamada a fork...\n");

13     /* Creamos un nuevo proceso. */
       pid = fork();
15
       if (pid == 0) {
17         printf("HIJO1: Hola, yo soy el primer hijo...\n");
           printf("HIJO1: Voy a pararme durante 60 seg. y luego terminaré...\n");
19         sleep(60);
           }
21     else if (pid > 0) {
           printf("PADRE: Hola, soy el padre. El pid de mi hijo es: %d\n",pid);
23
           /* Creamos un nuevo proceso. */
25         pid = fork();
           if (pid == 0) {
27             printf("HIJO2: Hola, soy el segundo hijo...\n");
               printf("HIJO2: El segundo hijo va a ejecutar la orden 'ls'...\n");
29             execlp("ls","ls",NULL);
               printf("HIJO2: Si ve este mensaje, el execlp no funcionó...\n");
31         }
           else if (pid > 0) {
33             printf("PADRE: Hola otra vez. Pid de mi segundo hijo: %d\n",pid);
               printf("PADRE: Voy a esperar a que terminen mis hijos...\n");
35             printf("PADRE: Ha terminado mi hijo %d\n",wait(NULL));
               printf("PADRE: Ha terminado mi hijo %d\n",wait(NULL));
37         }
           else
39           printf("Ha habido algún error al llamar por 2ª vez al fork\n");
       }
41     else
         printf("Ha habido algún error al llamar a fork\n");
43     return 0;
   }
```

2.2. runcom.c

```c
#include "smallsh.h"

/* Ejecuta una orden. "cline" es una array de palabras que contiene el nombre
 * de la orden y los parámetros de dicha orden. "where" indica si dicha
 * orden se debe ejecutar en primer o segundo plano.
 */

int
runcommand(char **cline, int where) {
        int pid, exitstat, ret;

        if ((pid = fork()) < 0) {
                perror("smallsh");
                return(-1);
        }

        if (pid == 0) {                         /* Estamos en el hijo. */
                execvp(*cline, cline);          /* Ejecutamos la orden. */
                perror(*cline);                 /* Se llega aquí si no se ha podido
                                                   ejecutar. Por tanto, se ha
                                                      producido
                                                   un error.*/
                exit(127);
        }

        /* Estamos en el padre. Si la orden se ejecuta en segundo plano, no
         * debemos esperar por lo que mostramos el pid del nuevo proceso y
         * regresamos. */
        if (where == BACKGROUND) {
                printf("[Identificador de proceso: %d]\n",pid);
                return(0);
        }

        /* Si la orden se ejecuta en primer plano, debemos esperar a que
         * termine.*/
        while ((ret = wait(&exitstat)) != pid && ret != -1)
        ;

        return(ret == -1 ? -1 : exitstat);
}

/* $Id: runcom.c 1399 2007-12-20 09:45:07Z pedroe $ */
```

2.3. signal.c

```c
/* El siguiente programa muestra cómo trabajan las señales. Este        *
 * programa captura la señal SIGINT (señal de interrupción generada    *
 * con CTRL-C). Probar el programa repetidas veces. En una              *
 * de ellas, no pulsar CTRL-C, en otra pulsar "una vez" CTRL-C          *
 * antes de que termine la ejecución y otra pulsar "más de una vez"     *
 * CTRL-C antes de que termine la ejecución. */

#include <signal.h>
#include <stdio.h>

/* Este es el prototipo de la función que va a tratar la señal. */
void trata_sig(int);

int main(void) {

    /* Hasta que no se ejecute la siguiente orden, la señal SIGINT *
     * provocar  la terminación del proceso (acción por defecto). */

    signal(SIGINT, trata_sig);

    /* Tras la ejecución del "signal" anterior, el control pasará  *
     * a la función "trata_sig" si se recibe la señal SIGINT.      */

    printf("Sleep numero 1.\n");
    sleep(1);
    printf("Sleep numero 2.\n");
    sleep(1);
    printf("Sleep numero 3.\n");
    sleep(1);
    printf("Sleep numero 4.\n");
    sleep(1);

    printf("Se ha terminado la ejecución.\n");
    return 0;
}

void trata_sig(int num_sig) {
    printf("\nSe ha capturado la señal número %d\n", num_sig);
    printf("Salimos de la rutina de atención a la interrupción\n\n");
}
```

2.4. sigaction.c

```
/* El siguiente programa muestra el uso de la función sigaction. Este      *
 * programa captura la señal SIGINT (señal de interrupción generada       *
 * con CTRL-C) una vez. Probar el programa repetidas veces. En una        *
 * de ellas, no pulsar CTRL-C, en otra pulsar "una vez" CTRL-C            *
 * antes de que termine la ejecución y otra pulsar "más de una vez"       *
 * CTRL-C antes de que termine la ejecución. */

#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

/* Este es el prototipo de la función que va a tratar la señal. */
void trata_sig(int);

int main(void) {
   struct sigaction accion;
   sigset_t conjunto;

   /* Establecemos el comportamiento del programa ante la señal *
    * SIGINT. La función sigemptyset permite construir un con-  *
    * junto de señales a bloquear. No vamos a bloquear ninguna. */
   sigemptyset(&conjunto);
   accion.sa_handler=trata_sig;
   accion.sa_mask=conjunto;
   accion.sa_flags=SA_ONESHOT;
   accion.sa_restorer=NULL;

   /* Hasta que no se ejecute la siguiente orden, la señal SIGINT *
    * provocar  la terminación del proceso (acción por defecto).  */

   sigaction(SIGINT,&accion,NULL);

   /* Tras la ejecución del "signal" anterior, el control pasará  *
    * a la función "trata_sig" si se recibe la señal SIGINT.      */

   printf("Sleep numero 1.\n");
   sleep(1);
   printf("Sleep numero 2.\n");
   sleep(1);
   printf("Sleep numero 3.\n");
   sleep(1);
   printf("Sleep numero 4.\n");
   sleep(1);

   printf("Se ha terminado la ejecución.\n");
   return 0;
}

void trata_sig(int num_sig) {
   printf("\nSe ha capturado la señal número %d\n",num_sig);
   printf("Salimos de la rutina de atención a la interrupción\n\n");
}
```

2.5. sigaction_chld.c

```
/* El siguiente programa muestra el uso de la función sigaction con la
 * señal SIGCHLD. La rutina de tratamiento obtiene un parámetro
 * adicional llamado "info" con información adicional, entre otras, el
 * PID del programa que acaba de terminar y que hizo que se lanzara la
 * señal SIGCHLD. El hijo termina tras imprimir un mensaje, y el padre
 * recibe la señal en cuanto este termina. */
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

/* Este es el prototipo de la función que va a tratar la señal. */
void tratamiento(int nsig, siginfo_t* info, void*nada);

int main(void) {
    struct sigaction accion;
    sigset_t conjunto;
    int pid;

    /* Establecemos el comportamiento del programa ante la señal
     * SIGCHLD. La función sigemptyset permite construir un con-
     * junto de señales a bloquear mientras se trata SIGCHLD. */
    sigemptyset(&conjunto);
    sigaddset(&conjunto,SIGCHLD);
    accion.sa_sigaction=tratamiento;
    accion.sa_mask=conjunto;
    accion.sa_flags=SA_RESTART | SA_SIGINFO;
    accion.sa_restorer=NULL;

    /* Activa el tratamiento de SIGCHLD */
    sigaction(SIGCHLD,&accion,(void*)NULL);

    /* Tras la ejecución del "signal" anterior, el control pasará
     * a la función "tratamiento" si se recibe la señal SIGCHLD (cuando
     * termine cualquier hijo) */

    printf("Inicio: Llamando a fork.\n");

    /* Creamos un nuevo proceso. */
    pid = fork();

    if (pid == 0) {
        printf("HIJO: Hola, yo soy el primer hijo...\n");
        printf("HIJO: Tras 5 segundos voy a terminar para que "
               "el padre reciba mi señal.\n");
        sleep (5);
        return 0;
    } else if (pid > 0) {
        printf("PADRE: Hola, soy el padre. El pid de mi hijo es: %d\n",pid);
        printf("PADRE: Realizando cualquier otra operación...\n");

        sleep (10);

        printf("PADRE: Se ha terminado la ejecución.\n");
        return 0;
```

```
    }
56
    return 0;
58 }

60 void tratamiento(int nsig, siginfo_t* info, void*nada)
   {
62    int exitstat;

64    printf("\nPADRE: Se ha capturado la señal número %d\n", nsig);

66    /* Acceso a info->si_pid, info->si_signo, etc.*/
      /* (si_pid es el pid del proceso que envía la señal */
68    printf("PADRE: El hijo que ha terminado tiene el pid %d\n", info->si_pid);

70    /* Esperar al proceso hijo (obligatorio para que no quede Zombie) */
      wait (&exitstat);
72
      printf("PADRE: El hijo retornó %d.\n", exitstat);
74 }
```

2.6. envia.c

```
/* El siguiente programa muestra el uso de kill para enviar señales de un *
 * proceso a otro. */

#include <signal.h>
#include <stdio.h>

int ntimes=0;
int main(int argc, char *argv[]) {
    int pid, ppid;
    void p_action(int), c_action(int);

    /* Establecemos la accion de la señal SIGUSR1 para el padre.*/
    signal(SIGUSR1,p_action);

    switch(pid = fork()) {
      case -1:                          /* Error. */
        perror(argv[0]);
        return 1;
      case 0:                           /* Hijo. */

        /* Establecemos la acción para el hijo. */
        signal(SIGUSR1,c_action);

        /* Obtenemos el pid del padre. */
        ppid=getppid();
        for(;;) {
            sleep(1);
            kill(ppid,SIGUSR1);
            pause();
        }
      default:                          /* Padre. */
        for(;;) {
            pause();
            sleep(1);
            kill(pid,SIGUSR1);
        }
    }
    return 0;
}

/* El siguiente procedimiento se ejecuta cuando se envía la señal        *
 * SIGUSR1 al padre. */
void p_action(int sig) {
    printf("El padre ha capturado la señal nº %d\n",++ntimes);
}

/* Idem al anterior pero para el hijo. */
void c_action(int sig) {
    printf("El hijo ha capturado la señal # %d\n",++ntimes);
}
```

2.7. alarma.c

```
/* El siguiente programa me permite establecer una alarma para que      *
 * imprima  un mensaje cuando transcurran los minutos que se le dan     *
 * como parámetro. */

#include <stdio.h>
#include <signal.h>
#include <stdlib.h>

#define TRUE 1
#define FALSE 0
#define BELLS "\007\007\007"

int alarm_flag = FALSE;

/* Este procedimiento maneja la señal SIGALRM. */
void setflag(int nada){
    alarm_flag = TRUE;
}

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

    if (argc <= 2) {
        fprintf(stderr, "Uso: %s #minutos mensaje\n", argv[0]);
        exit (1);
    }

    if ((nsecs = atoi(argv[1])*60) <= 0) {
        fprintf(stderr,"%s: número inválido de minutos",argv[0]);
        exit (2);
    }


    /* Aquí creamos un proceso en segundo plano. */
    switch(pid = fork()) {
    case -1:          /* Error. */
        perror(argv[0]);
        exit (1);
    case 0:           /* Hijo. Es el proceso que se queda en segundo plano. */
        break;
    default:          /* Padre. Muestra pid del proceso hijo y termina. */
        printf("%s -> Identificador de proceso: %d\n",argv[0],pid);
        return 0;
    }

    /* Fijamos la acción para la alarma. */
    signal(SIGALRM, setflag);

    /* Activamos la alarma. */
    alarm(nsecs);

    /* Esperamos hasta que recibamos una señal... */
    pause();
```

```
56      /* Si la señal recibida es SIGALRM, mostramos el mensaje. */
        if (alarm_flag == TRUE) {
58         printf(BELLS);
           putchar('\n');
60         for(j = 2; j < argc; j++)
              printf("%s ", argv[j]);
62         printf("\n");
        }
64
        return 0;
66  }
```

2.8. pipe3men.c

```
/* El siguiente programa muestra el uso de tuberías para pasar          *
 * información de un proceso a uno de sus descendientes o viceversa.    *
 * En nuestro ejemplo, un padre envía tres mensajes a un hijo que se    *
 * encarga de leerlos. */

#include <stdio.h>

#define MSGSIZE 17    /* Tamaño del mensaje. */

/* Mensajes a enviar. */
char *msg1 = "Hola, mundo nº 1";
char *msg2 = "Hola, mundo nº 2";
char *msg3 = "Hola, mundo nº 3";

int main(void) {
   char inbuf[MSGSIZE];
   int p[2], j, pid;

   /*Abrimos la tubería. */
   if (pipe(p) < 0) {
      perror("llamada a pipe");
      return 1;
   }

   /* Creamos el proceso hijo. Dicho proceso hereda los descriptores de los *
    * dos extremos de la tubería. */
   if ((pid = fork()) < 0) {
      perror("llamada al fork");
      return 2;
   }

   /* Si es el padre, entonces cierra el descriptor del fichero de lectura
    * y escribe en la tubería los tres mensajes.*/
   if (pid > 0) {
      close(p[0]);
      write(p[1], msg1, MSGSIZE);
      write(p[1], msg2, MSGSIZE);
      write(p[1], msg3, MSGSIZE);
      wait(NULL);
   }

   /*Si es el hijo, entonces cierra el descriptor del fichero de escritura y *
   * lee de la tubería */
   if (pid == 0) {
      close(p[1]);
      for(j = 0; j < 3; j++) {
         read(p[0], inbuf, MSGSIZE);
         printf("%s \n", inbuf);
      }
   }
   return 0;
}
```

2.9. pipecapa.c

```c
/* El siguiente programa escribe datos en una tubería hasta que ésta se *
 * bloquea. Este hecho es utilizado para determinar la capacidad de una *
 * tubería. En nuestro caso, sale un tamaño de 4096 bytes. */

#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
int count;
void alrm_action(int);

int main(void) {
    int p[2];
    char c= 'x';

    if (pipe(p) <0) {
        perror("llamada a pipe");
        exit(1);
    }
    signal(SIGALRM, alrm_action);
    for(count = 0;;) {
        /* Activamos la alarma. */
        alarm(20);

        write(p[1], &c, 1);

        /* Desactivamos la alarma. Si no llegamos aquí, es por que la      *
         * orden write anterior se ha quedado bloqueada al estar llena la *
         * tubería. Al no desactivar la alarma, se dispara y se ejecuta   *
         * el procedimiento "alrm_action" que muestra un mensaje y        *
         * termina la ejecución del proceso. */
        alarm(0);

        if ((++count % 1024) == 0)
            printf("%d caracteres en la tubería\n", count);
    }
    return 0;
}

/* Se ejecuta cuando se recibe la señal SIGALRM. */
void alrm_action(int nada) {
    printf("Escritura bloqueada después de %d caracteres\n", count);
    exit(0);
}
```

2.10. pipe2com.c

```
/* El siguiente programa muestra el uso de las tuberías y la llamada al *
 * sistema "exec". En concreto, se ve cómo se puede conectar la salida *
 * estándar de un programa a la entrada estándar de otro (este meca-    *
 * nismo se consigue en el shell con el símbolo '|'). */

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

void fatal(char *s) {
    perror(s);
    exit(1);
}

/* Une dos órdenes mediante una tubería. Como parámetros recibe         *
 * dos arrays de cadenas, cada uno con el nombre de la orden y          *
 * parámetros correspondientes. */
int join(char *com1[], char *com2[]) {

    int p[2], status;

    /* Creamos un proceso hijo para ejecutar las órdenes. */
    switch(fork()) {
     case -1:    /* Error. */
       fatal("Primera llamada a fork en join");
     case 0:     /* Hijo. */
       break;
     default:    /* Padre. Espera a que su hijo termine y regresa. */
       wait(&status);
       return(status);
    }

    /* Este es el código del primer hijo creado. */
    /* Creamos la tubería. */
    if (pipe(p) < 0)
       fatal("Llamada a pipe en join");

    /* Creamos un nuevo proceso, hijo del anterior hijo. */
    switch(fork()) {
     case -1:    /* Error. */
       fatal("Segunda llamada a fork en join");
     case 0:     /* Código del segundo hijo. */


       close(1);            /* Cerramos la salida estándar actual. */
       dup(p[1]);           /* Creamos un nuevo descriptor para el descriptor de *
                             * escritura de la tubería. */
       close(p[0]);         /* Cerramos los descriptores de la tubería. */
       close(p[1]);

       /* Si se regresa de "execvp" es porque ha fallado. */
       execvp(com1[0], com1);
       fatal("Primera llamada a execvp en join");

     default:    /* Código del primer hijo. */
```

```
55
        close(0);            /* Cerramos la entrada estándar actual. */
57      dup(p[0]);           /* Creamos un nuevo descriptor de fichero para el *
                              * descriptor de lectura de la tubería. */
59      close(p[0]);         /* Cerramos los descriptores de la tuber¡a. */
        close(p[1]);
61
        /* Si se regresa de "execvp" es porque ha fallado. */
63      execvp(com2[0], com2);
        fatal("Segunda llamada a execvp en join");
65    }
  }
67
  /* Primera orden y sus parámetros. */
69 char *one[] = {
     "ls","-l","/usr/lib",NULL
71 };

73 /* Segunda orden y sus parámetros. */
  char *two[] = {
75   "grep","^d",NULL
  };
77
  int main(void) {
79   int ret;

81   ret = join(one, two);
     printf("join devolvió %d\n",ret);
83   return ret;
  }
```

2.11. fifo_esc.c

```
/* El siguiente programa muestra el uso de una "FIFO". Este      *
 * programa escribe en la FIFO los mensajes que recibe como      *
 * parámetros. Antes de ejecutar este programa debemos crear     *
 * la FIFO (con "mkfifo tubo") y ejecutar en segundo plano el    *
 * programa que lee los mensajes de la FIFO ("pag160&").         */

#include <fcntl.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>

#define MSGSIZ 63    /* Tamaño máximo del mensaje. */

extern int errno;
char *fifo = "tubo";    /* Nombre de la FIFO. */


void fatal(char *s) {
    perror(s);
    exit(1);
}

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

    int fd, j, nwrite;
    char msgbuf[MSGSIZ+1];

    if (argc < 2) {
        fprintf(stderr, "Uso: %s mensaje....\n",argv[0]);
        exit(1);
    }

    /* Abrimos la FIFO para sólo escritura y con escrituras no bloqueantes *
     * cuando la FIFO está llena (O_NDELAY). */
    if ((fd = open(fifo, O_WRONLY | O_NDELAY)) < 0)
      fatal("falló la apertura de la FIFO");



    /* Enviamos mensajes. */
    for(j = 1; j < argc; j++) {

        if(strlen(argv[j]) > MSGSIZ) {
            fprintf(stderr, "mensaje demasiado largo %s\n", argv[j]);
            continue;
        }

        strcpy(msgbuf, argv[j]);

        /* Escribimos el mensaje en la FIFO. */
        if ((nwrite = write(fd, msgbuf, MSGSIZ + 1)) <= 0) {

            /* Si falla la escritura en la FIFO por estar llena, mostramos *
```

```
            * el correspondiente mensaje de error. */
56          if (nwrite == 0)
              errno = EAGAIN;
58          fatal("Ha fallado la escritura del mensaje");
        }
60    }
    return 0;
62 }

64 /* Nota. El significado de EAGAIN es: "Recurso temporalmente no *
   * disponible, intentar otra vez más tarde". Esto significa normalmente *
66 * que una tabla particular de sistema está llena. Por ejemplo, este    *
   * error puede ser generado por una llamada a fork cuando hay dema-    *
68 * siados procesos. */
```

2.12. fifo_lee.c

```
/* Este programa lee los mensajes escritos en una FIFO y los muestra    *
 * por pantalla. Funciona junto con el programa que escribe los         *
 * mensajes en la FIFO. Se debe ejecutar en segundo plano antes que     *
 * el programa que envía los mensajes, siempre que antes se cree la     *
 * FIFO que se utiliza ("mkfifo tubo"). */

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

#define MSGSIZ 63

char *fifo = "tubo";    /* Nombre de la FIFO que se utiliza. */

void fatal(char *s) {
    perror(s);
    exit(1);
}

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

    int fd;
    char msgbuf[MSGSIZ+1];

    /* Abrimos la FIFO para lectura y escritura. */
    if ((fd = open(fifo, O_RDWR)) < 0)
      fatal("Ha fallado la apertura de la fifo");

    for (;;) {

        if (read(fd, msgbuf, MSGSIZ+1) < 0)
          fatal("Ha fallado la lectura del mensaje");

        /* Escribimos los mensajes conforme se van recibiendo. */
        printf("mensaje recibido: %s\n", msgbuf);
    }
    return 0;
}
```

3. PROCESOS

- **Proceso** = programa en ejecución.
- Un proceso puede crear otros procesos ⇒ Árbol de procesos. Su raíz la ocupa el proceso «init», que es el segundo proceso que se ejecuta (el primero es el planificador o scheduler).
- **int pid = fork()**. Crea un nuevo proceso (hijo del que hace la llamada a fork). Padre e hijo continúan la ejecución en la instrucción inmediatamente posterior al fork (VER FIGURA).
- **pid = 0 en el hijo, >0 en el padre** (y contiene el pid del hijo) y <0 si error
- **exec**. Se ejecuta en el espacio del proceso llamador (VER FIGURA). Posibilidades:

  ```
  int execl(char *path, char *arg0, char *arg1,......, char *argn,
  NULL);

  int execv(char *path, char *argv[]);

  int execlp(char *file, char *arg0, char *arg1,......, char *argn,
  NULL);

  int execvp(char *file, char *argv[]);
  ```

  argv es similar al que aparece en main(int argc, char *argv[]);
- **Herencia en fork**: el hijo hereda todas las variables del padre con los mismos valores (excepto el valor que devuelve el fork), pero son independientes. Los ficheros abiertos en el padre, lo están en el hijo también ⇒ se comparten (y, por tanto, el puntero del fichero,....). No obstante, cerrar un fichero en el padre no afecta a los del hijo, y viceversa.
- **Herencia en exec**: deja los ficheros abiertos. Se debe activar el flag «close_on_exec» (por defecto a off) con «fcntl», para que se cierren automáticamente tras al hacer el exec. Por ejemplo:

  ```
  fcntl(fd,f_setfd,1);
  ```

  hace que el fichero representado por FD se cierre en un exec.
- **void exit(int estado)**. Para terminar un proceso. También se termina al llegar al final del código o con un return desde main. Se suele colocar exit(0) si ha habido éxito o exit(!=0) si ha habido algún tipo de error. Los 8 bits bajos de un exit están disponibles en el padre en los 8 bits altos (ver wait).
- **int resultado = wait(int *estado)**. Se utiliza en el padre para esperar a que terminen los hijos.
  - Si como puntero se pasa NULL, se ignora el valor de estado.
  - La función devuelve el pid del hijo que termina o -1 si ha habido error (no hay hijos,....).
  - Si los 8 bits menos significativos del estado tienen un valor distinto de 0 significa que el hijo terminó por una señal (y el valor es el número de señal).

    - Si son 0, en los 8 bits más significativos tenemos el valor que se colocó como parámetro en el exit.

- **Zombie**: hijo que hizo exit pero cuyo padre no ha hecho un wait.
- **Huérfano**: un padre puede terminar sin wait ⇒ hijos adoptados por el proceso de inicialización del sistema (init).
- **int pid = getpid()**. Devuelve el pid del proceso que la ejecuta.
- **int pid_padre = getppid()**. Devuelve el pid del padre.
- Variables de entorno.

  ```
  void main(int argc, char *argv[], char *envp[]);
  ```

  Por ejemplo, envp[0] puede tener unvalor como «HOME=/home/usuario».

4. COMUNICACIONES ENTRE PROCESOS. SEÑALES

- **void (*was) () = signal(int sig, void (*fun) ())**. Captura la señal «sig» mediante la función «fun». Devuelve un puntero a la función que previamente capturaba «sig».
- Tipos de señales:
    - **SIGINT**: Interrupción. Generada por el núcleo al pulsar la tecla de interrupción (CTRL-C).
    - **SIGQUIT**: Terminar. Muy similar a SIGINT. Enviada por el núcleo cuando se pulsa la tecla de terminación (CTRL-\). A diferencia de SIGINT, provoca una «terminación anormal» (toda terminación anormal provoca un «core dump», es decir, un vaciado de memoria).
    - **SIGILL**: Instrucción ilegal ⇒ Terminación anormal.
    - **SIGFPE**: Error de punto flotante ⇒ Terminación anormal.
    - **SIGKILL**: Enviada por un proceso o por el núcleo a otro proceso para que termine forzosamente éste último. No se puede capturar.
    - **SIGPIPE**: Enviada por el núcleo cuando se escribe en una tubería que no tiene un proceso que lea.
    - **SIGALRM**: Alarma. Enviada por el núcleo a un proceso transcurrido un cierto tiempo (ver la función ALARM).
    - **SIGTERM**: Señal de terminación software. Se utiliza para solicitar a un proceso que termine. No la envía el núcleo (ver KILL).
    - **SIGUSR1** y **SIGUSR2**: No las envía el núcleo y se pueden utilizar para lo que se quieran.

  Hay algunos tipos más de señales.

- **Funciones especiales**. Como segundo parámetro de la función «signal» se puede pasar una función normal o las funciones SIG_IGN (simplemente hace que la señal se ignore) y SIG_DFL (que indica que se realice la acción por defecto asociada a la señal).

- Si una señal no se captura ⇒ Terminación (normal o anormal, según el tipo de señal) del proceso receptor.
- Se pueden tratar varios tipos de señales a la misma vez.
- Las señales ignoradas por un proceso (SIG_IGN) siguen ignoradas tras un exec. ¿Qué pasa si un proceso en segundo plano no quiere que se ignoren?
- Hay llamadas al sistema que, excepcionalmente, son interrumpidas por una señal, como read, write,.....
- La función sigaction y otras funciones asociadas permiten un control total sobre las señales. Entre otras cosas, permite bloquear señales mientras se tratan otras, y si la rutina de tratamiento de una señal permanece instalada indefinidamente o se desinstala tras recibir la señal.
- En el caso concreto de tratar la señal sigchld el parámetro sa_flags debe ser SA_RESTART | SA_SIGINFO, y la función callback tendrá la forma:

```
void tratamiento(int nsig, siginfo_t* info, void*nada)
{
  /* Acceso a info->si_pid, info->si_signo, etc.*/
  /* (si_pid es el pid del proceso que envía la señal */
}
```

- **int kill(int pid, int sig)**. Permite enviar una señal de un proceso a otro, siempre que coincidan sus UID (identificador de usuario) reales o efectivos. Los procesos de superusuario pueden enviar señales a cualquier otro. KILL devuelve -1 si:
  - No coinciden los UID's reales o efectivos.
  - No existe el proceso.
  - No existe ese tipo de señal.

  Hay valores de pid que se le pasan a KILL con significados especiales.
- **unsigned int alarm(unsigned int seg)**. Hace que se envíe una señal SIGALRM al proceso, después de ¿seg¿ segundos. Una alarma se desactiva con ¿alarm(0)¿. Devuelve los segundos que quedaban para que se cumpliera la anterior alarma. Una alarma permanece activa tras un exec pero no en un hijo, tras un fork.
- **int pause(void)**. Detiene temporalmente un proceso hasta que éste reciba una señal (siempre que no la ignore porque, entonces, pause también la ignorará). Devuelve -1 cuando la señal se captura. Un proceso suspendido temporalmente no consume recursos del sistema.

5. COMUNICACIONES ENTRE PROCESOS. TUBERÍAS

- **Una tubería (pipe)** es un canal de comunicaciones unidireccional que conecta dos procesos. Es una generalización del concepto de fichero ya que se utilizan las mismas órdenes «read», «write» y «close» que se utilizan con ficheros.
- **int pipe(int filedes[2])**. Para crear la tubería. filedes[0] es el descriptor de fichero de lectura de la tubería y filedes[1] es el descriptor para la escritura en la tubería. «lseek» no funciona en una tubería. Los buffers de lectura y escritura en la tubería pueden tener distinto tamaño.

- Tras un fork, el hijo tiene dos descriptores propios para la tubería. Conviene que cada uno cierre (close) el que no vaya a utilizar.
- **Bloqueos**. Un write en una tubería se bloquea si está llena y un read se bloquea si está vacía (aunque se puede hacer que tanto el write como el read sean no bloqueantes).
- Cerrar un extremo de una tubería:
  - **Extremo de «sólo escritura»**: si hay más procesos que escriban en la tubería, no pasa nada. Si no es así, a los procesos que lean de la tubería a partir de ese momento reciben del read un 0. Si había procesos bloqueados porque la tubería estaba vacía, se les despierta del read, que devuelve también 0.
  - **Extremo de «sólo lectura»**: si hay más procesos que lean de la tubería, no pasa nada. Si no es así, a todos los procesos que están bloqueados en write o que intenten escribir a partir de ese momento, se les envía una señal «SIGPIPE» por el núcleo. Si capturan SIGPIPE, los siguientes writes devuelven -1.

6. COMUNICACIONES ENTRE PROCESOS. FIFO's

- Problemas de las tuberías:
  - Comunican procesos que comparten un ancestro (por ejemplo, un proceso padre y uno de sus hijos). Este problema surge al intentar crear un servidor.
  - No pueden ser permanentes.
- **Una FIFO** o tubería nombrada (named pipe) funciona prácticamente igual que una tubería. Diferencias:
  - Es permanente.
  - Tiene un nombre de fichero, un propietario, un tamaño y permisos.
  - Se abre, se cierra y borrar igual que un fichero.
- **Creación de una FIFO**: mkfifo <nombre>. También se puede utilizar la orden «mknod <nombre>p».

7. BIBLIOGRAFÍA

- «UNIX: programación avanzada». Márquez García, F^{o} Manuel.
- «UNIX: sistema y entorno». Fontaine, A.B.
- «UNIX system programing». Haviland, Keith.