T E M A 7

Programación en UNIX

Contenido

TEMA 7

PROGRAMACIÓN EN UNIX

7.1. Introducción

La programación en UNIX se basa en el uso de un conjunto de funciones que se denominan **llamadas al sistema**. Éstas constituyen un conjunto básico de elementos que permiten construir programas que pueden explotar los recursos del sistema. Mediante llamadas al sistema el programador puede:

- Utilizar el sistema de ficheros.
- Ejecutar varios procesos de forma concurrente.
- Comunicar y sincronizar procesos tanto de forma local como en red.

Las llamadas al sistema tienen un formato estándar, tanto en la documentación que suministra el manual del sistema como en la forma de invocación. Por ejemplo, la llamada al sistema *read()* tiene el siguiente formato:

```
int read(df, buf, contador)
int df ;
char *buf ;
int contador ;
```

La forma de uso de una llamada al sistema es idéntica a cualquier otra función del lenguaje C. La mayoría de las llamadas devuelven un valor, que suele indicar si la operación se ha efectuado con éxito o

7.2. Gestión de Errores

Cuando se produce un error al realizar una llamada al sistema, ésta devuelve el valor -1. Todo proceso contiene una variable global llamada *errno*, de manera que cuando una llamada al sistema termina de forma satisfactoria, el valor de esta variable permanece inalterado, pero cuando la llamada falla, en *errno* se almacena un código de error. *errno* no da información detallada sobre el error que se ha producido. Por otro lado, existe una función, *perror()*, que describe los errores que se producen en las llamadas al sistema.

```
void perror(cadena)
char *cadena ;
```

El siguiente programa muestra el uso de *errno* y *perror()*. En primer lugar se trata de abrir un fichero que no existe y después se trata de abrir en modo escritura un fichero del que no se posee tal permiso.

```
#include <stdio.h>
#include <sys/file.h>
#include <errno.h>
main()
{
    int fd ;
    fd = open("/asdaf", O_RDONLY) ;
    if (fd == -1)
    {
        printf("errno = %d\n", errno) ;
        perror("main") ;
    }
    if ((fd = open("/", O_WRONLY)) == -1)
    {
```

```
        printf("errno = %d\n", errno) ;
        perror("main") ;
    }
}
```

7.3. Gestión de Ficheros Regulares

Cada proceso en UNIX dispone de 20 descriptores de ficheros, numerados de 0 a 19. Por convención, los tres primeros descriptores se abren automáticamente cuando el proceso empieza su ejecución, y tienen un significado especial:

- descriptor 0: entrada estándar
- descriptor 1: salida estándar
- descriptor 2: error estándar

Cada descriptor de fichero posee un conjunto de propiedades privadas y que no depende del tipo de fichero que esté asociado al descriptor:

- Desplazamiento (*offset*) del fichero.
- Un indicador (*flag*) que indica si el descriptor se debe de cerrar de forma automática si el proceso utiliza la llamada al sistema `exec()`.
- Un indicador que indica si la salida hacia el fichero debe de añadirse al final del mismo.

Existe información adicional que tiene sentido si el fichero asociado al descriptor es un fichero especial.

7.3.1. Apertura de un fichero: `open()`

```
int open (ruta, indicadores [,modo])
char *ruta ;
int indicadores ;
mode_t modo ;
```

- `open()` permite abrir o crear un fichero para lectura y/o escritura.
- `ruta` es el nombre de fichero a abrir y puede estar expresado en forma de direccionamiento absoluto o relativo.
- `indicadores` es una máscara de bits, lo que significa que varias opciones pueden estar presentes usando el operador lógido `OR` (|) a nivel de bits. Indica al kernel el modo de apertura del fichero.
- `modo` indica los permisos de acceso del fichero en el caso de que éste vaya a ser creado. Se expresa octal.

Los indicadores de lectura/escritura son los siguientes:

- `O_RDONLY` Abrir para sólo lectura.
- `O_WRONLY` Abrir para sólo escritura.
- `O_RDWR` Abrir para lectura/escritura.

Otros indicadores son los siguientes:

- `O_APPEND` El desplazamiento (*offset*) se sitúa al final del fichero.
- `O_CREAT` No tiene efecto si el fichero existe, a no ser que el indicador `O_EXCL` esté activo. Si el fichero no existe, se crea con los permisos indicados en *mode*.
- `O_EXCL` Si el indicador `O_CREAT` está presente, `open()` da un mensaje de error si el fichero ya existe.
- `O_TRUNC` Si el fichero existe trunca su longitud a cero bytes.

Ejemplos:

```
int fd ;

/*
 * Abrir un fichero en modo sólo lectura
```

```
 */
fd = open("datos", O_RDONLY) ;

/*
 * Abrir un fichero en modo sólo escritura. Si el fichero existe,
truncar
 * su tamaño a cero bytes.  Si no existe, crearlo con permisos de
lectura
 * y escritura para el propietario y ningun permiso para el grupo y el
 * resto de los usuarios.
 */
fd = open("datos", O_WRONLY | O_TRUNC | O_CREAT, 0600) ;
```

7.3.2. Lectura de datos de un fichero: *read()*

```
int read (df, buf, contador)
int df ;
char *buf ;
int contador ;
```

- *read()* copia *contador* bytes desde el fichero referenciado por *df* en el buffer apuntado por *buf*. Si no existen errores, *read()* devuelve el número de bytes leídos y copiados en *buf*. Este número será menor o igual que *contador*. Es cero si se intenta leer cuando se ha llegado al final del fichero.
- La llamada *read()* es de bajo nivel y no tiene las capacidades de formateo de datos de *scanf()* *buffers* adicional de la biblioteca de funciones de C, por lo que es muy rápida.

Ejemplos:

```
int fd, nbytes ;
char buf1[1024] ;
float buf2[20] ;

/*
 * Leer en un buffer de 1024 caracteres.
 */
nbytes = read(fd, buf1, sizeof(buf1)) ;

/*
 * Leer en un array de 20 elementos float.
 */
nbytes = read(fd, buf2, 20 * sizeof(float)) ;
```

7.3.3. Escritura de datos en un fichero: *write()*

```
int write (df, buf, contador)
int df ;
char *buf ;
int contador ;
```

- *write()* copia *contador* bytes desde el *buffer* apuntado por *buf* en el fichero referenciado por *df*. Si no existen errores, *write()* devuelve el número de bytes escritos. Este número será menor o igual que *contador*.
- La llamada *write()* es de bajo nivel y no tiene las capacidades de formateo de datos de *printf()*. Tiene la ventaja de que no usa el sistema de *buffers* adicional de la biblioteca de funciones de C, por lo que es muy rápida.

Ejemplos:

```
int fd, nbytes ;
char *cadena = "Ejemplo de uso de write" ;
double num ;

/*
 * Ejemplos de uso de la llamada write.
 */
nbytes = write(fd, cadena, strlen(cadena)) ;

nbytes = write(fd, &num, sizeof(double)) ;
```

7.3.4. Cierre de un fichero: *close()*

```
int close (df)
int df ;
```

- *close()* libera un descriptor de fichero df y devuelve el valor 0. Si df era el último descriptor de fichero asociado con un fichero abierto, el kernel libera los recursos asociados con ese fichero.
- Al cerrar un fichero, la entrada que ocupaba en la tabla de descriptores de ficheros del proceso queda liberada y puede ser usada por una llamada a *open()*.
- Si un proceso no cierra los ficheros que tiene abiertos, al terminar su ejecución el kernel los cierra.

Ejemplos:

```
int fd, err ;

/*
 * Leer en un buffer de 1024 caracteres.
 */
err = close(fd) ;
```

7.3.5. Acceso aleatorio: *lseek()*

```
long lseek (df, desp, modo)
int df ;
int desp ;
int modo ;
```

- *lseek()* modifica el puntero de lectura/escritura (*offset*) asociado a *df* según el valor del parámetro *mode*:
  - ❑ *SEEK_SET*: el puntero avanza *desp* bytes con respecto al inicio del fichero.
  - ❑ *SEEK_CUR*: el puntero avanza *desp* bytes con respecto a su posición actual.
  - ❑ *SEEK_END*: el puntero avanza *desp* bytes con respecto al final del fichero.
- El puntero avanza hacia el final del fichero o hacia el principio dependiendo del signo de *desp*.
- En caso de fallo se devuelve -1, y en caso de éxito se devuelve un número entero no negativo que presenta el nuevo desplazamiento del puntero de lectura/escritura con respecto al principio del fichero.

7.3.6. Duplicación de un descriptor: *dup()/dup2()*

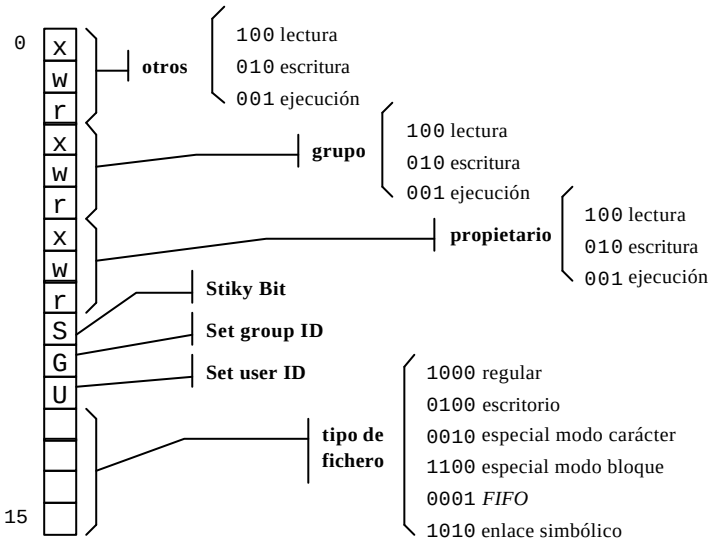
```
int dup (df_antiguo)
int df_antiguo;
int dup2 (df_antiguo, df_nuevo)
int df_antiguo;
int df_nuevo ;
```

- *dup()* busca el descriptor de fichero más pequeño que exista en la tabla de descriptores de ficheros y lo referencia hacia el mismo fichero que *df_antiguo*. *dup2()* cierra *df_nuevo* si estaba activo y lo referencia al mismo fichero que *df_antiguo*.

7.3.7. Información sobre un fichero: *stat()*

```
int stat (fichero, buf)
char *fichero ;
struct stat *buf ;
```

- *stat()* devuelve en *buf* la información que se almacena en la tabla de nodos-i sobre el estado del fichero *fichero*.
  La estructura *struct stat* tiene, entre otros, los siguientes campos:
  - *st_dev:* Número de dispositivo.
  - *st_ino:* Número de nodo-i.
  - *st_mode:* Modo del fichero.
  - *st_nlink:* Número de enlaces.
  - *st_uid:* Identificador de usuario del propietario.
  - *st_size:* Tamaño en bytes del fichero.
  - *st_atime:* Fecha del último acceso.
  - *st_mtime:* Fecha de última modificación.
  - *st_ctime:* Fecha de última modificación de información de tipo administrativo.
- Los bits de modo de un fichero, además de los nueve bits de protección, constituyen una máscara de 16 bits.
- El bit 9 *(stiky bit)* indica al kernel que el fichero es un programa con capacidad para que varios procesos compartan su segmento de código.
- El bit 10 *(Set Group ID)* tiene un significado similar al bit 11 pero referido a grupos de usuarios.
- El bit 11 *(Set User ID)* indica al kernel que cuando un proceso acceda a este fichero cambie el UID del proceso por el del propietario del fichero. Se habla en este caso de usuario efectivo (*efective UID*) para distinguirlo del usuario real (*real UID*). El identificador de usuario efectivo se emplea siempre para determinar permisos. El identificador de usuario real refleja la indentidad del usuario. Este mecanismo permite acceder a ficheros de otro usuario sobre los cuales no se tiene permiso.



- Los bits 12-15 indican el tipo del fichero. En el fichero de cabecera *<sys/stat.h>* existen un conjunto de macros predefinidas que, dado el modo de un fichero, devuelven el valor verdadero si el fichero es del tipo preguntado :
  - *S_ISDIR:* directorio
  - *S_ISCHR:* dispositivo de caracteres
  - *S_ISBLK:* dispositivo de bloques
  - *S_ISREG:* fichero regular
  - *S_ISFIFO:* *FIFO*.

7.3.8. Modificación del modo de un fichero: *chmod()*

```
int chmod (ruta, modo)
char *ruta ;
mode_t modo ;
```

- *chmod()* cambia el modo del fichero indicado por *ruta* por *modo*.

7.3.9. Creación de enlaces: *link()*

```
int link (ruta1, paht2)
char *ruta1 ;
char *ruta2 ;
```

- *link()* crea un nueva entrada de directorio *ruta2* que apunta al mismo fichero que *ruta1*. El contador de enlaces del fichero se incrementa en una unidad.
- Si *ruta1* y *ruta2* residen en dispositivos físicos diferentes la orden *link()* no se puede llevar a cabo.
- Únicamente el superusuario puede hacer enlaces de directorios.

7.3.10. Borrado de un fichero: *unlink()*

```
int unlink (ruta)
char *ruta ;
```

- *unlink()* elimina el enlace de *ruta* al fichero que tiene asociado. Si *ruta* es el último enlace del fichero, los recursos del fichero son devueltos al sistema.
- Si se va a eliminar físicamente un fichero y algún proceso tiene un descriptor de fichero asociado con él, la entrada del directorio es eliminada, pero el fichero no es borrado realmente hasta que se haya cerrado el último descriptor asociado al fichero.

7.3.11. Operaciones sobre descriptores de fichero: *fcntl()*

```
int fcntl (df, oper, arg)
int df ;
int oper ;
int arg ;
```

- *fcntl()* realiza la operación indicada por *oper* sobre el fichero asociado con *df*. *arg* es un argumento opcional para *oper*.
- Mediante *fcntl()* se puede controlar un fichero abierto de forma que, entre otras posibilidades, es posible modificar los modos permitidos de acceso al fichero, bloquear el acceso a parte del mismo o a su totalidad, etc.

7.4. Gestión de Directorios

La gestión de directorios depende de si el sistema UNIX concreto es System V o BSD, ya que cada uno de ellos implementa los directorios de forma diferente. Para evitar que los programas no sean portables entre distintos sistemas, el estándar POSIX define una biblioteca que contiene una serie de funciones de manejo, a alto nivel, de directorios, de forma que no se requiere conocer la estructura física de los mismos.

7.4.1. Creación de un directorio: *mknod(), mkdir()*

```
int mknod(nombre, modo, disp)
char *nombre ;
mode_t modo ;
char disp ;
```

- *mknod()* crea un nuevo fichero llamado *nombre*, que puede ser regular, directorio o especial. El tipo del fichero se especifica por *modo*, que es una máscara de bits que puede tomar, entre otros, los siguientes valores:
  - *S_IFDIR*: directorio
  - *S_IFCHR*: fichero especial orientado a caracteres
  - *S_IFBLK*: fichero especial orientado a bloques
  - *S_IFREG*: fichero regular
  - *S_IFIFO*: FIFO.
- Esta llamada al sistema únicamente puede ser usada por el superusuario, excepto en el caso de que el fichero sea FIFO.
- *disp* se usa si se va a crear un fichero especial.
- Para crear directorios si no es superusuario, se emplea la función de biblioteca *mkdir()*.

```
int mkdir(nombre, modo)
char * nombre ;
mode_t modo;
```

- *mkdir()* crea un directorio llamado *nombre* con los permisos indicados por *modo*.

7.4.2. Borrado de un directorio: *rmdir()*

```
int rmdir (dir)
char *dir ;
```

- *rmdir()* borra el directorio *dir si* está vacío. Devuelve 0 si se ejecuta satisfactoriamente y -1 en caso de error.

7.4.3. Apertura de un directorio: *opendir()*

```
DIR *opendir (dir)
char *dir ;
```

- *opendir()* abre el directorio *dir* y devuelve un puntero a una estructura de tipo *DIR* (denominada *stream*). En caso de error, devuelve *NULL*.

Ejemplos:

```
#include <sys/types.h>
#include <dirent.h>
DIR *dir ;
char directorio = "/usr/bin/X11" ;

if (dir = opendir(directorio)) == NULL) {
    perror(directorio) ;
    exit(-1) ;
```

```
}
```

7.4.4. Lectura de un directorio: *readdir()*

```
struct dirent *readdir(punt_dir)
DIR *punt_dir ;
```

– *readdir()* lee la entrada a la que apunta *punt_dir*, que deberá de apuntar a un directorio abierto con *opendir()*, y va a actualizar el puntero a la siguiente entrada para permitir una lectura de tipo secuencial. Devuelve *NULL* cuando llega al final del directorio o se produce algún error.

La estructura *struct dirent* se compone de los siguientes campos:

```
struct dirent
{
 ino_t d_ino;    /* Nodo-i de la entrada   */
 short d_reclen; /* Longitud de la entrada */
 short d_namlen; /* Longitud del nombre    */
 char  d_name[_MAXNAMLEN + 1] /* Nombre    */
}
```

7.4.5. Cierre de un directorio: *closedir()*

```
int closedir(punt_dir)
DIR * punt_dir;
```

– *closedir()* cierra el directorio apuntado por *punt_dir*. Devuelve 0 si se realiza satisfactoriamente o -1 en caso de error.

7.4.6. Cotrol del puntero de lectura de un directorio: *seekdir(), telldir(), rewinddir()*

```
void seekdir(punt_dir, pos)
DIR * punt_dir;
long pos ;
```

– *seekdir()* sitúa el puntero de lectura de un directorio en la posición indicada por *pos*.

```
long telldir(punt_dir)
DIR * punt_dir;
```

– *telldir()* devuelve la posición actual del puntero de lectura.

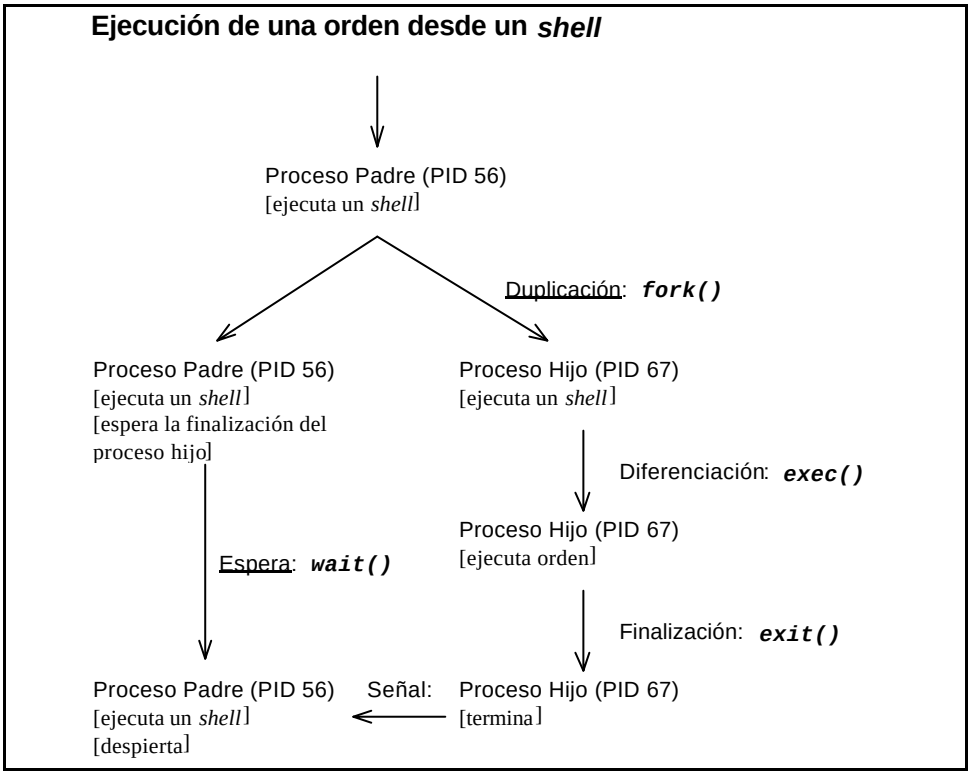
```
void rewinddir(punt_dir)
DIR * punt_dir;
```

– *rewinddir()* posiciona el puntero de lectura al principio del directorio.

7.5. Gestión de Procesos

Cuando UNIX comienza su ejecución únicamente existe un proceso en el sistema. Este proceso se llama ***init*** y su PID es 1.

La única forma de crear nuevos procesos en UNIX consiste en duplicar procesos existentes. Cuando un proceso se duplica, el proceso padre y el hijo son idénticos: el proceso hijo contiene una copia del codígo, datos y pila del padre. Las únicas diferencias entre ambos son sus PID y PPID. Cuando un proceso hijo termina, su finalización es notificada al proceso padre. Un proceso padre tiene la capacidad de poder esperar a que uno de sus hijos termine. Un proceso hijo puede reemplazar su código con el de un fichero ejecutable.



7.5.1. Creación de un proceso: *fork()*

```
int fork ()
```

- *fork()* duplica un proceso. El proceso hijo hereda del padre una copia de su código, datos, pila, descriptores de ficheros abiertos y tabla de señales. El padre y el hijo se diferencian en sus PID y PPID.
- Si la llamada tiene éxito, devuelve el PID del proceso hijo al proceso padre y 0 al proceso hijo. Si falla, devuelve -1 al padre y el hijo no se crea.

Ejemplo:

```
#include <stdio.h>

main()
{
  switch(fork())
  {
    int i = 0 ;

    case -1: /* Error */
        perror("Error al crear proceso\n") ;
        exit(-1) ;
        break ;

    case 0:   /* Proceso hijo */
        printf("soy el hijo: %d\n", i++) ;
        break ;

    default:  /* Proceso padre */
        printf("soy el padre: %d\n", i++);
  }
}
```

7.5.2. Obtención de identificadores: *getuid(), getgid(), ...*

```
int getuid ()
```

- *getuid()* devuelve el identificador de usuario real (*UID*).

```
int getgid ()
```

- *getgid()* devuelve el identificador de grupo real.

```
int geteuid ()
```

- *geteuid()* devuelve el identificador de usuario efectivo.

```
int getegid ()
```

- *getegid()* devuelve el identificador de grupo efectivo.

```
int getpid ()
```

- *getpid()* devuelve el identificador de proceso (*PID*).

```
int getpgrp ()
```

- *getpgrp()* devuelve el identificador de grupo del proceso.

```
int getppid ()
```

- *getppid()* devuelve el identificador del padre del proceso (*PPID*).

7.5.3. Finalización de un proceso: *exit()*

```
void exit (estado)
int estado ;
```

- *exit()* cierra todos los descriptores de ficheros, devuelve al sistema los recursos ocupados por el código, datos y pila, y termina el proceso. El valor de *estado* es devuelvo al sistema, y se puede consultar mediante la variable de entorno *$?*.
- Cuando el proceso hijo termina le envía al padre una señal *SIGCLD* y espera hasta que el valor *estado* sea aceptado por el padre, permaneciendo en estado zombie.
- Todos aquellos procesos hijos que sean huérfanos son adoptados por el proceso *init*, que siempre acepta los códigos de terminación de sus procesos hijo.

7.5.4. Espera de terminación de un hijo: *wait()*

```
int wait (estado)
int *estado ;
```

- *wait()* provoca la suspensión del proceso hasta que uno de sus hijos termine. Si la llamada tiene *PID* del hijo que ha terminado y *estado* contiene el valor que el proceso hijo devuelve mediante *exit()*.

7.5.5. Ejecución de programas: *exec()*

```
int execl (ruta, arg0, arg1,..., argn, NULL)
char *ruta, *arg0, *arg1, ..., *argn ;
int execv (ruta, argv)
char *ruta, *argv[] ;
int execlp (ruta, arg0, ..., argn, NULL)
char *ruta, *arg0, ..., *argn ;
int execvp (ruta, argv)
char *ruta, *argv[] ;
```

- La familia de llamadas al sistema *exec()* reemplazan el código, datos y pila del proceso que las invoca por el programa ejecutable indicado por *ruta*.
- *execl()* es idéntica a *execlp()*, y *execv()* es idéntica a *execvp()*, salvo que *execl()* y *execv()* requieren el camino absoluto o relativo para encontrar el fichero ejecutable, mientras que *execlp()* y *execvp()* usan la variable de entorno *$PATH* para encontrar *ruta*.
- Si el fichero ejecutable no se encuentra, se devuelve -1. En caso contrario, el proceso que realiza la llamada reemplaza su código, datos y pila por los del fichero ejecutable y comienza a ejecutar su nuevo

- Si la ejecución se realiza correctamente, no se devuelve ningún valor.
- *execl()* y *execlp()* invocan el fichero ejecutable usando como argumentos las cadenas de caracteres apuntadas por *arg0*, *arg1*, ..., *argn*. *arg0* debe ser el nombre del fichero ejecutable, y la lista de argumentos debe de terminar con la cadena vacía *NULL* (*(char *)0*).
- *execv()* y *execvp()* invocan el fichero ejecutable usando como argumentos un array de cadenas de caracteres apuntado por *argv*. *argv[n+1]* es *NULL* y *argv[0]* contiene el nombre del fichero ejecutable.

Ejemplo:

```
/*
 * Este programa escribe un mensaje y
 * reemplaza su código por el de la orden ls.
 */
#include <stdio.h>
```

```
main()
{
  printf("soy el proceso %d\n", getpid()) ;
  execl("/usr/bin/ls", "ls", "-l", NULL) ;
  printf("Esta linea no se debe imprimir\n");
}
```

7.6. Señales

En muchas situaciones los programas deben de estar preparados para tratar situaciones inesperadas o impredecibles, como:

- error en operación en punto flotante,
- aviso de un reloj de alarma,
- la muerte de un proceso hijo,
- solicitud de terminación por parte del usuario (*Control-C*),
- solicitud de suspensión por parte del usuario (*Control-Z*),
- etc.

Cuando una de estas situaciones se produce, el kernel envía una señal al proceso correspondiente. Además, cualquier proceso puede enviar una señal a otro proceso, si tiene permiso. En System V hay definidas 19 señales, mientras que BSD define 11 señales más.

Cuando un proceso recibe una señal puede tratarla de tres formas diferentes:

- Ignorar la señal, con lo cual es inmune a la misma.
- Invocar a una rutina de tratamiento por defecto. Esta rutina la posee el kernel.
- Invocar a una rutina propia para tratar la señal.

La rutina de tratamiento por defecto de una señal realiza una de las siguientes acciones:

- Termina el proceso y genera un fichero *core*, que contiene un volcado de memoria del contexto del proceso (*dump*).
- Termina el proceso sin generar un fichero *core* (*quit*).
- Ignora la señal (*ignore*).
- Suspende el proceso (*suspend*).
- Reanuda la ejecución del proceso.

En la siguiente tabla se muestran algunas señales de la versión de UNIX SunOS.

Nombre Señal	Nº	Descripción	Acción por Defecto		
			Generar core	Terminar	Ignorar
SIGHUP	01	suspender (hangup)		✓	
SIGINT	02	interrupción		✓	
SIGQUIT	03	terminar (quit)	✓	✓	
SIGILL	04	instrucción ilegal	✓	✓	
SIGFPE	08	error en punto flotante	✓	✓	
SIGKILL	09	terminar (no ignorable)		✓	
SIGBUS	10	bus error	✓	✓	
SIGSEGV	11	violación de segmento	✓	✓	
SIGPIPE	13	error de escritura en pipe		✓	
SIGALRM	14	alarm clock		✓	
SIGTERM	15	terminación software		✓	
SIGCHLD[1]	20	muerte del proceso hijo			✓
SIGIO	23	e/s posible en un descriptor			✓
SIGUSR1	30	señal 1 definida por el usuario		✓	
SIGUSR2	31	señal 2 definida por el usuario		✓	

[1] SIGCLD en System V

7.6.1. Envío de una señal: *kill()*

```
int kill (pid, sig)
int pid ;
int sig ;
```

- *kill()* envía la señal con valor *sig* al proceso cuyo *PID* es *pid*.
- La señal se envía de forma satisfactoria si el proceso que envía y el que recibe son del mismo usuario, o bien si el proceso que envía es del superusuario.
- *kill()* funciona de forma diferente dependiendo del valor de pid:
  - Si *pid > 0* la señal se envía al proceso cuyo *PID* es *pid*.
  - Si *pid = 0* la señal se envía a todos los procesos que pertenecen al mismo grupo del proceso
  - Si *pid = -1* la señal se envía a todos procesos cuyo *UID* real es igual al *UID* efectivo del proceso que la envía. Si el proceso que la envía tiene *UID* efectivo de superusuario, la señal es enviada a todos los procesos, excepto al proceso 0 (*swapper*) y 1 (*init*).
  - Si *pid < -1* la señal es enviada a todos los procesos cuyo *ID* de grupo coincide con el valor absoluto de *pid*.

7.6.2. Tratamiento de una señal: *signal()*

```
void (*signal (sig, accion))()
int sig ;
void (*accion)() ;
```

- *signal()* permite a un proceso especificar la acción a tomar cuando reciba una señal en particular. *sig* especifica el número de la señal a tratar y *accion* puede tomar tres tipos de valores:
  - *SIG_DFL*: indica que se use el manejador por defecto del kernel.
  - *SIG_IGN*: indica que la señal de debe ignorar.
  - una dirección de una función definida por el usuario para el tratamiento de la señal.

- Un proceso hijo puede heredar las especificaciones sobre señales del proceso padre durante la llamada a *fork()*. Cuando se realiza un *exec()*, las señales previamente ignoradas permanecen ignoradas pero con las manejadores por defecto.
- Con la excepción de *SIGCLD*, la señales no se apilan, lo que significa que si un proceso está durmiendo y recibe tres señales idénticas, únicamente recibe una de ellas.

7.7. Comunicación entre procesos

La comunicación entre procesos se realiza en UNIX mediante un conjunto de mecanismos que van a permitir a los procesos el intercambio de datos y sincronización. Algunos de estos mecanismos son:

- ***pipes*** (*unnamed pipes*): canales de comunicación síncrona unidireccional entre procesos relacionados.
- ***FIFOS*** (*named pipes*): igual que los pipes pero pueden ser usados por procesos que no están relacionados. Sólo están disponibles en System V.
- ***sockets***: permiten la comunicación entre procesos que se encuentran en máquinas diferentes.
- memoria compartida: permite la comunicación de processos mediante la compartición de una zona

Los pipes o tuberías son canales unidireccionales que tienen una capacidad de almacenamiento temporal (*buffering*) de 4K en BSD y 40K en System V. Cada extremo del pipe tiene asociado un descriptor, de forma que sobre un extremo un proceso puede escribir mediante la llamada al sistema ***write()*** y otro puede leer en el otro extremo mediante la llamada ***read()***.

Cuando un proceso no necesita leer o escribir en un pipe, los descriptores del mismo deben de cerrarse mediante ***close()***.

7.7.1. Creación de un pipe: ***pipe()***

```
int pipe (df)
int df[2] ;
```

- ***pipe()*** crea un *pipe* sin nombre y devuelve dos descriptores de fichero: el descriptor asociado para la lectura en un extremo del *pipe* se almacena en ***df[0]*** y el de la escritura en el otro extremo en ***df[1]***.

  La lectura de un *pipe* se rige por las siguientes reglas:
  - Si un proceso lee de un *pipe* cuyo descriptor de escritura ha sido cerrado, ***read()*** devuelve 0 indicando fin de la entrada.
  - Si un proceso lee de un *pipe* vacío cuyo descriptor de escritura sigue abierto, el proceso espera hasta que se produzca alguna entrada.
  - Si un proceso intenta leer más bytes de un *pipe* de los que éste contiene en un instante dado, todo el contenido del pipe es obtenido por ***read()*** y esta llamada devuelve el número de bytes que se han leído.

  La escritura en un pipe se rige por las siguientes reglas:
  - Si un proceso escribe en un pipe cuyo descriptor de lectura ha sido cerrado, ***write()*** falla y el proceso recibe una señal ***SIGPIPE***. La acción por defecto de esta señal es terminar el proceso.
  - Si un proceso escribe un número menor de bytes en un *pipe* de los que éste puede contener, la llamada ***write()*** es atómica. Esto significa que el proceso terminará su llamada sin que le sea quitada la CPU. Esto no se garantiza si se escriben más bytes de los que puede contener el *pipe*.

7.8. Referencias

- "Modern Operating Systems". A.S. Tanenbaum. Prentice-Hall. 1992.
- "Advanced UNIX Programming". M.J. Rochkind. Prentice-Hall. 1985.
- SunOS Reference Manual.

7.9. Apendice 1: Programas ejemplo

7.9.1. Programa `copy.c`

```
/*
 ******************************************************************************
 *  NOMBRE:
 *     copy.c
 *  DESCRIPCION:
 *     Copia un fichero a otro fichero
 *  SINTAXIS:
 *     copy fichero_origen fichero_destino
 *  DEVUELVE:
 *      0: ejecucion correcta.
 *     -1: se ha producido un error.
 *  FECHA: 16-3-95
 *  COMENTARIOS:
 *     - Version en ANSI-C
 ******************************************************************************
 */


/*
 * Fichero de cabecera de la biblioteca estandar de entrada/salida.
 */

#include <stdio.h>
/*
 * Fichero de cabecera para llamadas a al sistema de ficheros.
 */
#include <fcntl.h>


char buffer[BUFSIZ] ; /* Buffer para almacenamiento temporal. */
                      /* BUFSIZ esta definido en stdio.h.     */


/*
 *  Definicion de los flags de que indican el modo de apertura del
 *  fichero destino.
 */
#define FLAGS_CREACION (O_WRONLY | O_TRUNC | O_CREAT)

/*
 *  Definicion del modo por defecto del fichero destino.
 */
#define MODO_CREACION 0644

/*----------------------
 *  Nombre        : main
 *  Descripcion   : Funcion principal del programa.
 *  Argumentos    :
 *            [E]  argc : numero de argumentos de la linea de ordenes,
 *                        incluyendo la orden.
 *            [E]  argv : puntero a un array de cadenas de caracteres.
 *                        Permite acceder a la linea de ordenes.
 *            [E]  envp : puntero a un array de cadenas de caracteres.
 *                        Permite acceder a las variables de entorno.
 *  Devolucion    : nada
 *  Observaciones : ninguna
 */

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

```
{
    int  fd_origen ;   /* Descriptor del fichero origen.  */
    int  fd_destino ;  /* Descriptor del fichero destino. */
    int  num_bytes ;   /* Contador.                       */

    /*
     *  1.-  Analisis de la linea de ordenes.
     */
    if (argc != 3)
    {
         fprintf(stderr, "Sintaxis: copy fichero_origen fichero_destino\n") ;
         exit(-1) ;
    }

    /*
     *  2.-  Apertura de fichero origen en modo solo lectura.
     */
    fd_origen = open(argv[1], O_RDONLY) ;
    if (fd_origen == -1)
    {
         perror (argv[1]) ;
         exit(-1) ;
    }

    /*
     *  3.-  Apertura o creacion del fichero destino en modo solo escritura.
     */
    if ((fd_destino = open(argv[2], FLAGS_CREACION , MODO_CREACION)) == -1)
    {
         perror (argv[2]) ;
         exit(-1) ;
    }

    /*
     *   4.-  Copia del contenido del fichero origen en el fichero destino.
     */
    while ((num_bytes = read(fd_origen, buffer, sizeof buffer)) > 0)
         write(fd_destino, buffer, num_bytes) ;

    /*
     *   5.-  Cierre de los dos ficheros.
     */
    close(fd_origen) ;
    close(fd_destino) ;
}
```

7.9.2. Programa `info.c`

```
/*
 ******************************************************************************
 *  NOMBRE:
 *     info.c
 *  DESCRIPCION:
 *     Muestra el contenido del nodo-i de uno o varios ficheros
 *  SINTAXIS:
 *     info fichero
 *  DEVUELVE:
 *     0: ejecucion correcta.
 *     -1: se ha producido un error.
 *  FECHA: 22-3-95
 *  COMENTARIOS:
 *     - Version en C K&R
 ******************************************************************************
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

char permisos[] = {'x', 'w', 'r'} ;

/*----------------------
 *  Nombre        : main
 *  Descripcion   : Funcion principal del programa.
 *  Argumentos    :
 *            [E]  argc : numero de argumentos de la linea de ordenes,
 *                        incluyendo la orden.
 *            [E]  argv : puntero a un array de cadenas de caracteres.
 *                        Permite acceder a la linea de ordenes.
 *            [E]  envp : puntero a un array de cadenas de caracteres.
 *                        Permite acceder a las variables de entorno.
 *  Devolucion    : nada
 *  Observaciones :
 */
main (argc, argv, envp)
int argc ;
char **argv ;
char **envp ;
{
   int i ;

   void estado() ;  /* Funcion que imprime el estado de un fichero */

   /*
    * 1.- Analisis de la linea de ordenes
    */
    if (argc < 2)
    {
         fprintf(stderr, "Sintaxis: info fichero\n") ;
         exit(-1) ;
    }

    for (i = 1;i < argc; i++)
       estado(argv[i]) ;
}


/*----------------------
 *  Nombre        : estado
 *  Descripcion   : imprime por pantalla informacion sobre un fichero.
 *  Argumentos    :
```

```
 *            [E]  fichero : nombre del fichero.
 *  Devolucion    : nada
 *  Observaciones : ninguna
 */
void estado(fichero)
char *fichero ;
{
    struct stat buf ; /* Variable usada en la llamada stat() */
    int i ;

    /*
     *  Invocacion de la llamada al sistema stat()
     */
    if (stat(fichero, &buf) == -1)
    {
         perror(fichero) ;
         exit(-1) ;
    }

    /*
     *  Impresion del nombre del fichero y su numero de inode
     */
    printf("\n\n") ;
    printf("Fichero        : %s\n", fichero) ;
    printf("Numero de inode: %d\n", buf.st_ino) ;

    /*
     *  Impresion del tipo del fichero
     */
    printf("Tipo           : ") ;

    if (S_ISDIR(buf.st_mode))
    {
         printf("directorio\n") ;
    }
    else if (S_ISREG(buf.st_mode))
    {
         printf("regular\n") ;
    }
    else if (S_ISCHR(buf.st_mode))
    {
         printf("especial modo caracter\n") ;
    }
    else if (S_ISBLK(buf.st_mode))
    {
         printf("especial modo bloque\n") ;
    }

    /*
     *  Se comprueba si el bit 11 (Set User ID) esta activo
     */
    if (buf.st_mode & S_ISUID)
         printf("bit 11 (Set User ID) activo\n") ;

    /*
     *  Se comprueba si el bit 10 (Set Group ID) esta activo
     */
    if (buf.st_mode & S_ISGID)
         printf("bit 10 (Set Group ID) activo\n") ;

    /*
     *  Se comprueba si el bit 9 (Stiky bit) esta activo
     */     if (buf.st_mode & S_ISVTX)
         printf("bit 9 (Stiky bit) activo\n") ;

    /*
```

```
     *  Impresion de los permisos del fichero
     */
    printf("Permisos       : ") ;
    printf("0%o ", buf.st_mode & 0777) ;
    for (i = 0; i < 9; i++)
       if (buf.st_mode & (0400 >> i))
          printf("%c", permisos[(8-i)%3]) ;
       else
          printf("-") ;
    printf("\n") ;

    /*
     *  Impresion del numero de enlaces "hard"
     */
    printf("Enlaces        : %d\n", buf.st_nlink) ;

    /*
     *  Impresion del UID y GID
     */
    printf("User ID        : %d\n", buf.st_uid) ;
    printf("Group ID       : %d\n", buf.st_gid) ;

    /*
     *  Impresion de la longitud del fichero
     */
    printf("Longitud       : %d\n", buf.st_size) ;

    /*
     *  Impresion de la fecha de ultimo acceso,
     *  ultima modificacion y ultimo cambio en el estado del fichero
     */
    printf("Fecha de ultimo acceso: %s", asctime(localtime(&buf.st_atime))) ;
    printf("Fecha de ultimo modificacion: %s",
                  asctime(localtime(&buf.st_mtime))) ;
    printf("Fecha de ultimo cambio de estado: %s",
                  asctime(localtime(&buf.st_ctime))) ;
}
```

7.9.3. Programa `padre_hijo.c`

```
/*
 ******************************************************************************
 *  NOMBRE:
 *     padre_hijo.c
 *  DESCRIPCION:
 *     Ejemplo del uso de las llamadas fork(), wait() y exit()
 *  SINTAXIS:
 *     padre_hijo
 *  DEVUELVE:
 *     0: ejecucion correcta.
 *     -1: se ha producido un error.
 *  FECHA: 22-3-95
 *  COMENTARIOS:
 *     - Version en C K&R
 ******************************************************************************
 */

#include <stdio.h>

/*----------------------
 *  Nombre        : main
 *  Descripcion   : Funcion principal del programa.
 *  Argumentos    :
 *  Devolucion    : nada
 *  Observaciones :
 */
main ()
{
   int pid1, pid2 ; /* variables para almacenar pids */

   printf("Soy el proceso padre (PID: %d)\n", getpid()) ;

   pid1 = fork() ;

   if (pid1 != 0)   /* Proceso padre */
   {
       printf("Soy el proceso padre otra vez (PID: %d)\n", getpid()) ;
       pid2 = wait(NULL) ;  /* No me interesa el status de mi hijo */
       printf("Mi proceso hijo (PID: %d) ha terminado\n", pid2) ;
   }
   else  /* Proceso hijo */
   {
       printf("Soy el proceso hijo (PID: %d, PPID: %d)\n", getpid(), getppid());
       exit(0) ; /* He terminado sin problemas */
   }
   printf("Yo, el proceso padre (PID: %d) he terminado\n", getpid()) ;
}
```

7.9.4. Programa `pipe.c`

```
/*
 ******************************************************************************
 *  NOMBRE:
 *     pipe.c
 *  DESCRIPCION:
 *     Programa de ejemplo del uso de pipes.  Un proceso padre se comunica
 *     con un proceso hijo mediante un pipe.
 *  SINTAXIS:
 *     pipe
 *  DEVUELVE:
 *     0: ejecucion correcta.
 *     -1: se ha producido un error.
 *  FECHA: 23-3-95
 *  COMENTARIOS:
 *     - Version en C K&R
 ******************************************************************************
 */

#include <stdio.h>

#define MAX 10

char *MENSAJE = "Mensaje del proceso padre al hijo. FIN" ;


/*----------------------
 *  Nombre        : main
 *  Descripcion   : Funcion principal del programa.
 *  Argumentos    :
 *            [E]  argc : numero de argumentos de la linea de ordenes,
 *                        incluyendo la orden.
 *            [E]  argv : puntero a un array de cadenas de caracteres.
 *                        Permite acceder a la linea de ordenes.
 *            [E]  envp : puntero a un array de cadenas de caracteres.
 *                        Permite acceder a las variables de entorno.
 *  Devolucion    : nada
 *  Observaciones :
 */
main (argc, argv, envp)
int  argc  ;
char **argv ;
char **envp ;
{
    int tuberia[2] ;  /* Array para almacenar los descriptores del pipe */
    int pid ;

    char buffer[MAX] ;  /* Buffer para uso de read() y write() */

    if (pipe(tuberia) == -1)
    {
        perror("pipe") ;
        exit(-1) ;
    }

    if ((pid = fork()) == -1)
    {
        perror("fork") ;
        exit(-1) ;
    }
    else if (pid == 0)  /* Proceso hijo */
    {
        while (read(tuberia[0], buffer, MAX) >0)
           printf("Proceso hijo: %s\n", buffer) ;
```

```
        close(tuberia[0]) ;
        close(tuberia[1]) ;
        exit(0) ;
    }
    else  /* Proceso padre */
    {
        write(tuberia[1], MENSAJE, strlen(MENSAJE)+1) ;

        close(tuberia[0]) ;
        close(tuberia[1]) ;
        exit(0) ;
    }

}
```

7.9.5. Programa `pipe_sh.c`

```
/*
 ******************************************************************************
 *  NOMBRE:
 *     pipe_sh.c
 *  DESCRIPCION:
 *     Programa de ejemplo del uso de pipes para mostrar el mecanismo por
 *     el cual el shell permite el encadenamiento de procesos.
 *     procesos .
 *  SINTAXIS:
 *     pipe_sh programa1 programa2
 *  DEVUELVE:
 *     0: ejecucion correcta.
 *     -1: se ha producido un error.
 *  FECHA: 23-3-95
 *  COMENTARIOS:
 *     - Version en C K&R
 *     - Los programas a encadenar no pueder tener ni opciones ni argumentos
 *     - La secuencia de pasos para redireccionar la salida de una orden
 *       a la entrada la otra, una vez que se han creados dos procesos
 *       con un pipe compartido, es la siguiente:
 *       1.- En el proceso que va a ejecutar el primer programa de la cadena,
 *           se cierra el descritor de salida estandar y se redirecciona
 *           mediante la llamada dup() al descriptor de escritura del pipe.
 *       2.- En el proceso que se va a ejecutar el segundo programa,
 *           se cierra el descriptor de entrada estandar y se redirecciona
 *           al descriptor de lectura del pipe.
 ******************************************************************************
 */

#include <stdio.h>

/*----------------------
 *  Nombre        : main
 *  Descripcion   : Funcion principal del programa.
 *  Argumentos    :
 *            [E]  argc : numero de argumentos de la linea de ordenes,
 *                        incluyendo la orden.
 *            [E]  argv : puntero a un array de cadenas de caracteres.
 *  Devolucion    : nada
 *  Observaciones :
 */
main (argc, argv)
int  argc ;
char **argv ;
{
    int tuberia[2] ;  /* Array para almacenar los descriptores del pipe */
    int pid ;

    if (argc != 3)
    {
        fprintf(stderr, "Sintaxis: pipe_sh programa_1 programa_2\n") ;
        exit(-1) ;
    }

    if (pipe(tuberia) == -1)
    {
        perror("pipe") ;
        exit(-1) ;
    }

    /*
     *  Despues de forr(), ambos procesos comparten los descriptores de
     *  ficheros abiertos.  Concretamente, comparten los descriptores del
```

```
     *  pipe.
     */
    if ((pid = fork()) == -1)
    {
        perror("fork") ;
        exit(-1) ;
    }
    else if (pid == 0)  /* El proceso hijo ejecuta el primer programa */
    {                   /* de la cadena.                              */
        /*
         *  Paso 1: Se cierra el descriptor 0, que se corresponde con la
         *          entrada estandar.
         */
         close(1) ;

        /*
         *  Paso 2: La entrada 0 de la tabla de descriptores de fichero del
         *          proceso hijo esta libre.  Al realizar la llamada a la
         *          funcion dup(), el descriptor 0 sera el duplicado.  De
         *          esta forma, la salida estandar del primer programa de la
         *          cadena sera la entrada del pipe.
         */
         dup(tuberia[1]) ;

         /*
          *  Paso 3: Se cierran aquellos descriptores compartidos que no
          *          van a ser usados por este proceso.  Concretamente,
          *          no son necesarios los descriptores del pipe.
          */
          close(tuberia[0]) ;
          close(tuberia[1]) ;

         /*
          *  Paso 4: Se ejecuta el primer programa de la cadena, que hereda
          *          los descriptores de ficheros abiertos.
          */
          execlp(argv[1], argv[1], 0) ;
          /*
           * Si se ejecutan estas instrucciones es que la llamada execlp ha fallado.
           */
          fprintf(stderr, “Error al ejecutar el primer programa del pipe” ) ;
          exit(-1)
    }
    else  /* El proceso padre ejecuta el segundo programa de la cadena */
    {
          close(0) ;
          dup(tuberia[0]) ;
          close(tuberia[0]) ;
          close(tuberia[1]) ;
          if (execlp(argv[2], argv[2], 0) == -1)
          {
                fprintf(stderr, “Error al ejecutar el primer programa del pipe” ) ;
                exit(-1) ;
          }
    }
}
```

7.9.6. Programa `signal.c`

```
/*
 ******************************************************************************
 *  NOMBRE:
 *     signal.c
 *  DESCRIPCION:
 *     Muestra un ejemplo basico de uso de signal().  Concretamente, se
 *     utiliza como ejemplo el tratamiento de la senal SIGINT.
 *  SINTAXIS:
 *     signal1
 *  DEVUELVE:
 *     0: ejecucion correcta.
 *     -1: se ha producido un error.
 *  FECHA: 22-3-95
 *  COMENTARIOS:
 *     - Version en C K&R
 ******************************************************************************
 */

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

/*----------------------
 *  Nombre        : main
 *  Descripcion   : Funcion principal del programa.
 *  Argumentos    :
 *  Devolucion    :
 *  Observaciones :
 */

main ()
{
   void manejador() ;  /* Funcion de tratamiento de interrupcion */

   signal(SIGINT, manejador) ;

   printf("soy el proceso %d\n", getpid()) ;

   while (1)  /* Bucle infinito */
   {
        printf("En espera de un CTRL-C\n") ;
        sleep(2) ; /* Me duermo durante 2 segundos */
   }

}
/*----------------------
 *  Nombre        : manejador
 *  Descripcion   : funcion de tratamiento de la interrupcion SIGINT.
 *  Argumentos    :
 *            [E]   sig : identificador de senal.
 *  Devolucion    : nada
 *  Observaciones : esta funcion es llamada por el kernel
 */
void manejador(sig)
int sig ;
{
    printf("Se ha recibido la senal %d\n", sig) ;
}
```