· int **system** (char * cadena)
#include <stdio.h>,<stdlib.h>

i.e. salida **= system (**"ps –U jsan5732"**);**

· int **execl** (camino, arg0 [, arg1, ...] , 0)
#include <stdio.h>,<unistd.h>

i.e. execl ("/bin/ps", "ps", "-fu", getenv ("USER"), 0);

· pid_t **fork** ();
#include <unistd.h>

Devuelve 0 al proceso hijo y
PID del hijo al proceso padre
(-1, si error)

· pid_t **getpid ();** Devuelve el PID del proceso.

· pid_t **waitpid** (pid_t PID, int * estados, int opciones);
#include <sys/wait.h>

PID **-1** waitpid actúa igual que wait, esperando cualquier hijo.
>0 PID de un proceso hijo determinado.
0 Para cualquier hijo con el mismo grupo de procesos que el padre.
<-1 para cualquier hijo cuyo grupo de proceso sea igual al valor absoluto de PID.

opciones
WNOHANG: evita la suspensión del padre mientras esté esperando a algún hijo.
WUNTRACED: el padre obtiene información adicional si el hijo recibe alguna de las señales **SIGTTIN**, **SIGTTOU**, **SIGSSTP** o **SIGTSTOP** .

estados
Puntero a una tabla con los estados de salida de los procesos.

· pid_t **wait** (int * estados); Igual que **wait**(-1,estados,0);
#include <sys/wait.h>

· **WIFSTOPPED** (pid_t estado) <>0, si *estado* es de un hijo parado
· **WSTOPSIG** (pid_t estado) Num de señal que ha causado la parada
· **WIFEXITED** (pid_t estado) <>0, si *estado* es de salida normal
· **WEXITSTATUS** (pid_t estado) 8 bits bajos del estado de salida
· **WIFSIGNALED** (pid_t estado) <>0, si *estado* es de salida anormal
· **WTERMSIG** (pid_t estado) Num de sañal que ha causado la slida

· pid_t **exit** (int estado);

- Se limpian las áreas de E/S (bloqueanda)
- Se cierran todos los descriptores de fichero.
- Si el proceso padre está en espera (ver wait), se devuelve el valor de los 8 bits menos significativos del estado de salida.
- Se envía una señal SIGCHLD al proceso padre. La acción por defecto es ignorar esta señal. Si no se ignora, el proceso hijo puede quedar como proceso zombi.
- Se eliminan los bloqueos de ficheros.

SIGNALS

· **Señal:** Evento que debe ser procesado y que puede interrumpir el flujo normal de un programa.

· **Capturar una señal:** Una señal puede asociarse con una función que procesa el evento que ha ocurrido.

· **Ignorar una señal:** El evento no interrumpe el flujo del programa. Las señales **SIGINT** y **SIGSTOP** no pueden ser ignoradas.

· **Acción por defecto:** Proceso suministrado por el sistema para capturar la

· **Alarma:** Señal que es activada por los temporizadores del sistema.

· **Error:** Fallo o acción equivocada que puede provocar la terminación del proceso.

· **Error crítico:** Error que provoca la salida inmediata del programa.

Núm.	Nombre	Comentarios
1	SIGHUP	Colgar. Generada al desconectar el terminar.
2	SIGINT	Interrupción. Generada por teclado.
3	SIGQUIT1	Salir. Generada por teclado.
4	SIGILL1	Instrucción ilegal. No se puede recapturar.
5	SIGTRAP1	Trazado. No se puede recapturar.
6	SIGABRT1	Abortar proceso.
8	SIGFPE1	Excepción aritmética, de coma flotante o división por cero.
9	SIGKILL1	Matar proceso. No puede capturarse, ni ignorarse.
10	SIGBUS1	Error en el bus.
11	SIGSEGV1	Violación de segmentación.
12	SIGSYS1	Argumento erróneo en llamada al sistema.
13	SIGPIPE	Escritura en una tubería que otro proceso no lee.
14	SIGALRM	Alarma de reloj.
15	SIGTERM	Terminación del programa.
16	SIGURG2	Urgencia en canal de E/S.
17	SIGSTOP3	Parada de proceso. No puede capturarse, ni ignorarse.
18	SIGTSTP3	Parada interactiva. Generada por teclado.
19	SIGCONT4	Continuación. Generada por teclado.
20	SIGCHLD2	Parada o salida de proceso hijo.
21	SIGTTIN3	Un proceso en 2o plano intenta leer del terminal.
22	SIGTTOU3	Un proceso en 2o plano intenta escribir en el terminal.
23	SIGIO2	Operación de E/S posible o completada.
24	SIGXCPU	Tiempo de UCP excedido.
25	SIGXFSZ	Excedido el límite de tamaño de fichero.
30	SIGUSR1	Definida por el usuario número 1.
31	SIGUSR2	Definida por el usuario número 2.
34	SIGVTALRM	Alarma de tiempo virtual.
36	SIGPRE	Excepción programada. Definida por el usuario.

· void **signal** (SIGNAL_ID, void *accion())

```
#include <signal.h>
```

i.e.

```
void alarma () { stop=1;  }

main { signal (SIGALRM, alarma);
       alarm(15);
       while (¡stop) ; }
```

· void **kill** (pid_t, signal)

```
#include <signal.h>
```

Envia señal SIGNAL a proceso.

PIPES

Descriptor de fichero: Número entero positivo usado por un proceso para identificar un fichero abierto. Esta traducción se realiza mediante una tabla de descriptores de fichero, ubicado en la zona de datos del proceso.

Descriptores reservados

- 0: entrada normal (**stdin**).
- 1: salida normal (**stdout**).
- 2: salida de error (**stderr**).

Redirección: Establecer copias del descriptor de ficheros de un archivo para encauzar las operaciones de E/S hacia otro fichero.

Tubería: Mecanismo de intercomunicación entre procesos que permite que 2 o más procesos envíen información a cualquier otro.

Tubería sin nombre: Enlace de comunicación unidireccional, capaz de almacenar su entrada.

Tuberías nombradas (FIFO): Permiten una comunicación menos restringida, ya que las colas FIFO existen en el sistema de archivos hasta que son borradas.

- Permite comunicar procesos no emparentados.
- Tiene una entrada en el sistema de archivos.
- Usa una política de colas "primero en llegar, primero en servirse".

· int **dup** (int desc_abierto)
#include <unistd.h>,<fcntl.h>,<sys/types.h>

= **fcntl** (desc_abierto, F_DUPFD, 0);

· int **dup2** (int desc_abierto, int desc_nuevo)
#include <unistd.h>,<fcntl.h>,<sys/types.h>

= **close**(desc_nuevo); **fcntl**(desc_abierto, F_DUPFD, desc_nuevo);

i.e.

```
desc_fich = creat (args[1]);
dup2 (desc_fich, 1);
close (desc_fich);
execvp ("ls","-o");
```

· int **fcntl** (int descriptor, int comando, int argumento)
#include <unistd.h>,<fcntl.h>,<sys/types.h>

comando	Descripción
F_DUPFD	Obtener el menor descriptor de fichero disponible que sea mayor que el parámetro descriptor. Mantiene el mismo puntero y las mismas características del fichero original.
F_GETFD	Obtener características del descriptor.
F_SETFD	Poner características del descriptor.
F_GETFL	Obtener estado del fichero.
F_SETFL	Poner estado del fichero.
F_GETLK	Obtener información de bloqueo.
F_SETLK	Poner bloqueo.
F_SETLKW	Poner bloqueo en una zona bloqueada.
F_GETOWN	Obtener PID (>0) o PGID (<0) del proceso que recibe las señales SIGIO o SIGURG.
F_SETOWN	Poner PID (>0) o PGID (<0) del proceso gestor de la E/S asíncrona.
F_CLOSEM	Cierra todos los descriptores desde descriptor hasta el valor máximo (OPEN_MAX).

Estados del modo de acceso al fichero
- **O_RDONLY** Abierto sólo para lectura.
- **O_RDWR** Abierto para lectura y escritura.
- **O_WRONLY** Abierto sólo para escritura.

Bloqueos
- **F_RDLCK** Bloqueo de lectura (compartido).
- **F_WRLCK** Bloqueo de escritura (exclusivo).
- **F_UNLCK** Sin bloqueo.

Comentarios
- Un bloqueo de lectura evita que otros procesos activen bloqueos de lectura en cualquier zona del área protegida. Sí se permiten otros bloqueos de lectura en toda el área o en partes de ella.
- Un bloqueo de escritura evita que otros procesos bloqueen dicha zona.
- Los "abrazos mortales" en un sistema distribuido no siempre son detectables.
- El programa deberá usar temporizadores para poder liberar sus bloqueos.

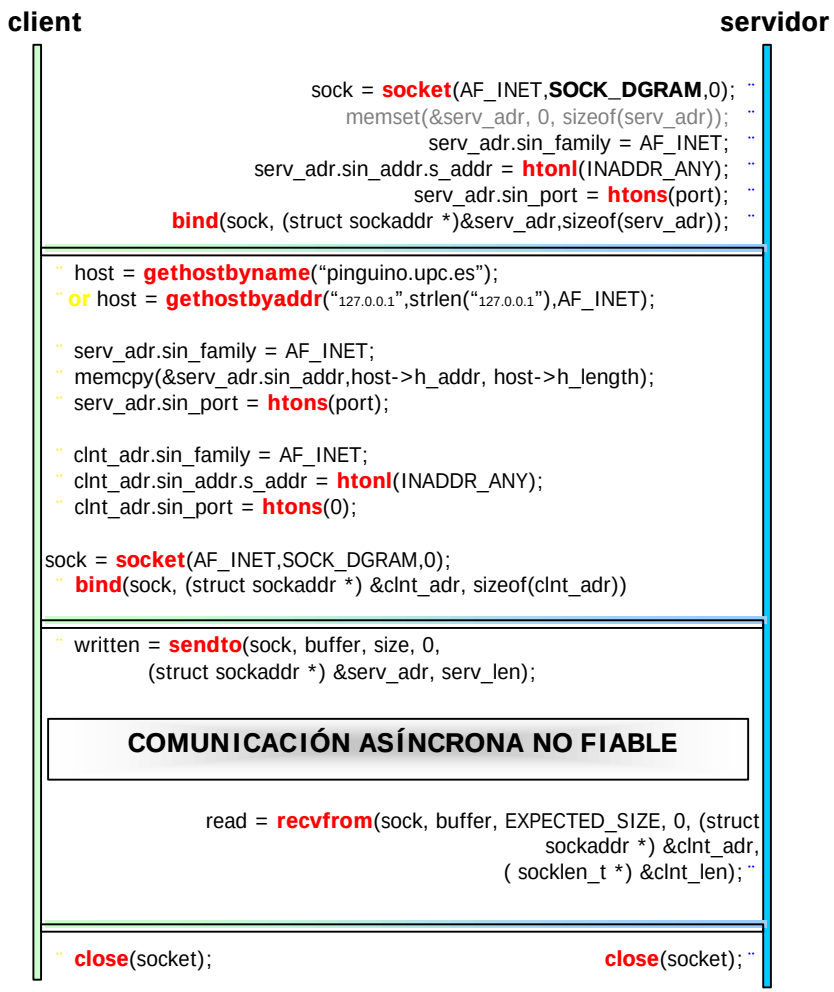
· int **pipe** (int descriptores[2])
#include <unistd.h>

- *descriptores*[0] se abre para lectura y *descriptores*[1], para escritura.
- La operación de lectura en *descriptores*[0] accede a los datos escritos en *descriptores*[1] como en una cola FIFO (primero en llegar, primero en servirse),

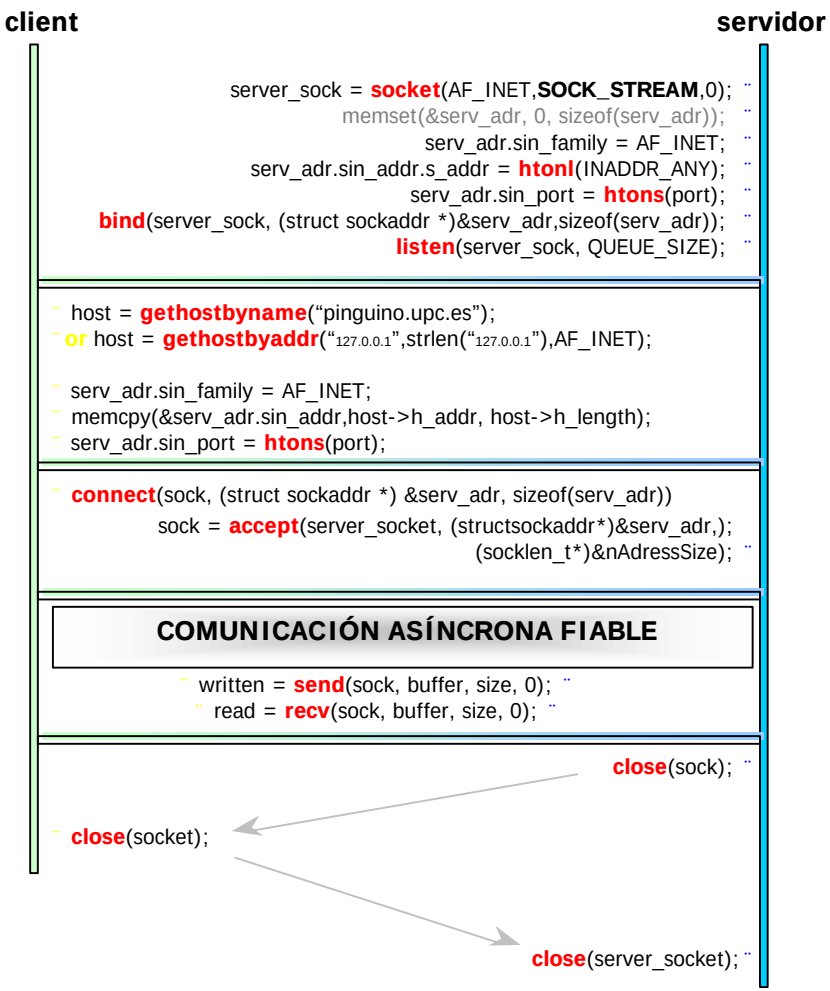
· int **mkfifo** (const char *camino, int modo)

SERVIDOR	CLIENT
unlink ("tuberia"); mkfifo ("tuberia", 0); chmod ("tuberia", 460); f = open ("tuberia", O_RDONLY); ...	f = open ("tuberia", O_WRONLY); write(f,...) Close(f);

UDP/IP


client
servidor
sock = socket(AF_INET,SOCK_DGRAM,0);
memset(&serv_adr, 0, sizeof(serv_adr));
serv_adr.sin_family = AF_INET;
serv_adr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_adr.sin_port = htons(port);
bind(sock, (struct sockaddr *)&serv_adr,sizeof(serv_adr));
host = gethostbyname("pinguino.upc.es");
or host = gethostbyaddr("127.0.0.1",strlen("127.0.0.1"),AF_INET);
serv_adr.sin_family = AF_INET;
memcpy(&serv_adr.sin_addr,host->h_addr, host->h_length);
serv_adr.sin_port = htons(port);
clnt_adr.sin_family = AF_INET;
clnt_adr.sin_addr.s_addr = htonl(INADDR_ANY);
clnt_adr.sin_port = htons(0);
sock = socket(AF_INET,SOCK_DGRAM,0);
bind(sock, (struct sockaddr *) &clnt_adr, sizeof(clnt_adr))
written = sendto(sock, buffer, size, 0,
(struct sockaddr *) &serv_adr, serv_len);
COMUNICACIÓN ASÍNCRONA NO FIABLE
read = recvfrom(sock, buffer, EXPECTED_SIZE, 0, (struct
sockaddr *) &clnt_adr,
(socklen_t *) &clnt_len);
close(socket);
close(socket);

TCP/IP


client
servidor
server_sock = socket(AF_INET,SOCK_STREAM,0);
memset(&serv_adr, 0, sizeof(serv_adr));
serv_adr.sin_family = AF_INET;
serv_adr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_adr.sin_port = htons(port);
bind(server_sock, (struct sockaddr *)&serv_adr,sizeof(serv_adr));
listen(server_sock, QUEUE_SIZE);
host = gethostbyname("pinguino.upc.es");
or host = gethostbyaddr("127.0.0.1",strlen("127.0.0.1"),AF_INET);
serv_adr.sin_family = AF_INET;
memcpy(&serv_adr.sin_addr,host->h_addr, host->h_length);
serv_adr.sin_port = htons(port);
connect(sock, (struct sockaddr *) &serv_adr, sizeof(serv_adr))
sock = accept(server_socket, (structsockaddr*)&serv_adr,);
(socklen_t*)&nAdressSize);
COMUNICACIÓN ASÍNCRONA FIABLE
written = send(sock, buffer, size, 0);
read = recv(sock, buffer, size, 0);
close(sock);
close(socket);
close(server_socket);

Rutinas útiles

```
int leer_hasta(int fd, char * s, int max, char hasta)
{
        int i = -1;
        do { i++; if(read(fd, &s[i], 1)<0) return –1;
             if (i==max) { s[i] = 0; return i; }
         } while (s[i]!=hasta);
        s[i] = 0; return i;
}

int leer_todo(int fd, char * s, int max)
{
        int i = -1;
        do { i++; if(read(fd, &s[i], 1)<0) return –1;
            } while (i<max);
        return i;
}

int  abrir_fifo_bloqueante(char * fname, int mode, int &fd)
{
        do { fd = open(fname, mode);
            } while ((fd==-1) && (errno==ENOENT));
        return fd<=0?-1:0;
}

int algo_esperando(int fd, int timeout)
{
    fd_set rfds;    int retval;   timeval tv;
    FD_ZERO(&rfds);  FD_SET(fd, &rfds);
    tv.tv_sec  = timeout;   tv.tv_usec = 0;
    retval = select(fd+1, &rfds, NULL, NULL, (struct timeval*)&tv);
    return retval<0?-1: retval?1:0;
}
```

```
int escribir_archivo_protegido ()
{
        int fd; struct flock lock;

        fd = open(“win.ini”, O_WRONLY|O_CREAT);

        lock.l_whence = SEEK_SET;
        lock.l_start = desde;
        lock.l_len = longitud;
        lock.l_type = F_WRLCK;
        fnctl(fd, F_SETLK, &lock);

            // escribir en archivo en zona protegida

        lock.l_type = F_UNLCK;
        fnctl(fd, F_SETLK, &lock);
        close(fd);
}

int leer_archivo_protegido ()
{
        int fd; struct flock lock;
        fd = open(“win.ini”, O_RDONLY);
        lock.l_whence = SEEK_SET;
        lock.l_start = desde; lock.l_len = longitud;
        lock.l_type = F_RDLCK;
        lseek(fd,  lock.l_start, SEEK_SET);
        fnctl(fd, F_SETLK, &lock);

            // leer de archivo en zona protegida

        lock.l_type = F_UNLCK;   fnctl(fd, F_SETLK, &lock);
        close(fd);
}
```

```
int tiempo_de_muestreo(int fd, int espera)
{
    int retval;   timeval tv;
    tv.tv_sec  = espera;   tv.tv_usec = 0;
    retval = select(0,NULL, NULL, NULL, (struct timeval*)&tv);
    return retval<0?-1:0;
}
```