APÉNDICE

B Funciones de biblioteca de uso más frecuente

Función	Tipo de salida	Propósito	Archivo de cabecera
abs(i)	int	Devuelve el valor absoluto de i.	stdlib.h
acos(d)	double	Devuelve el arco coseno de d.	math.h
asin(d)	double	Devuelve el arco seno de d.	math.h
atan(d)	double	Devuelve el arco tangente de d.	math.h
atan(d1,d2)	double	Devuelve el arco tangente de d1/d2.	math.h
atof(s)	double	Convierte la cadena s a una cantidad en doble precisión.	stdlib.h
atoi(s)	int	Convierte la cadena s a un entero.	stdlib.h
atol(s)	long	Convierte la cadena s a un entero largo.	stdlib.h
calloc(u1,u2)	void*	Reserva memoria para un array de u1 elementos, cada uno de u2 bytes. Devuelve un puntero al principio del espacio reservado.	malloc.h stdlib.h
ceil(d)	double	Devuelve un valor redondeado por exceso al siguiente entero mayor.	math.h
cos(d)	double	Devuelve el coseno de d.	math.h
cosh(d)	double	Devuelve el coseno hiperbólico de d.	math.h
difftime(t1,t2)	double	Devuelve la diferencia de tiempo t1-t2, donde t1 y t2 representan el tiempo transcurrido después de un tiempo base.	time.h
exp(d)	double	Eleva el número e (2.7182818...) a la potencia d.	math.h
fabs(d)	double	Devuelve el valor absoluto de d.	math.h
fclose(f)	int	Cierra el fichero f. Devuelve 0 si el archivo se ha cerrado con éxito.	stdio.h
feof(f)	int	Determina si se ha encontrado el fin del archivo f. Si es así, devuelve un valor distinto de cero; en otro caso devuelve 0.	stdio.h
fgetc(f)	int	Lee un carácter del archivo f.	stdio.h
fgets(s,i,f)	char*	Lee una cadena s, con i caracteres, del archivo f.	stdio.h
floor(d)	double	Devuelve un valor redondeado por defecto al entero menor más cercano.	math.h

Función	Tipo de salida	Propósito	Archivo de cabecera
`fopen(s1,s2)`	`file*`	Abre un archivo llamado `s1`, del tipo `s2`. Devuelve un puntero al archivo.	`stdio.h`
`fprintf(f,...)`	`int`	Escribe datos en el archivo `f` de acuerdo a un determinado formato especificado en los restantes argumentos.	`stdio.h`
`fputc(c,f)`	`int`	Escribe un carácter en el archivo `f`.	`stdio.h`
`fputs(s,f)`	`int`	Escribe una cadena de caracteres `s` en el archivo `f`.	`stdio.h`
`fread(s,i1,i2,f)`	`int`	Lee `i2` elementos, cada uno de tamaño `i1` bytes, desde el archivo `f` hasta la cadena `s`.	`stdio.h`
`free(p)`	`void`	Libera un bloque de memoria reservada cuyo principio está indicado por `p`.	`malloc.h` `stdlib.h`
`fscanf(f,...)`	`int`	Lee datos del archivo `f` de acuerdo a un determinado formato especificado en los restantes argumentos.	`stdio.h`
`fseek(f,l,i)`	`int`	Mueve el puntero al archivo `f` una distancia de `l` bytes desde la posición `i`.	`stdio.h`
`ftell(f)`	`long int`	Devuelve la posición actual del puntero dentro del archivo `f`.	`stdio.h`
`fwrite(s,i1,i2,f)`	`int`	Escribe `i2` elementos, cada uno de tamaño `i1` bytes, desde la cadena `s` hasta el archivo `f`.	`stdio.h`
`getc(f)`	`int`	Lee un carácter del archivo `f`.	`stdio.h`
`getchar()`	`int`	Lee un carácter desde el dispositivo de entrada estándar.	`stdio.h`
`gets(s)`	`char*`	Lee una cadena de caracteres desde el dispositivo de entrada estándar.	`atdio.h`
`isalnum(c)`	`int`	Determina si el argumento es alfanumérico. Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	`ctype.h`
`isalpha(c)`	`int`	Determina si el argumento es alfabético. Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	`ctype.h`
`isascii(c)`	`int`	Determina si el argumento es un carácter ASCII. Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	`ctype.h`
`iscntrl(c)`	`int`	Determina si el argumento es un carácter ASCII de control. Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	`ctype.h`
`isdigit(c)`	`int`	Determina si el argumento es un dígito decimal. Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	`ctype.h`

Función	Tipo de salida	Propósito	Archivo de cabecera
`islower(c)`	`int`	Determina si el argumento es una minúscula. Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	`ctype.h`
`isodigit (c)`	`int`	Determina si el argumento es un dígito octal. Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	`ctype.h`
`isprint(c)`	`int`	Determina si el argumento es un carácter ASCII imprimible (hex `0x20-0x70`; octal `040-176`). Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	`ctype.h`
`ispunct(c)`	`int`	Determina si el argumento es un carácter de puntuación. Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	`ctype.h`
`isspace(c)`	`int`	Determina si el argumento es un espacio en blanco. Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	`ctype.h`
`isupper(c)`	`int`	Determina si el argumento es una mayúscula. Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	`ctype.h`
`isxdigit (c)`	`int`	Determina si el argumento es un dígito hexadecimal. Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	`ctype.h`
`labs(l)`	`long int`	Devuelve el valor absoluto de `l`.	`math.h`
`log(d)`	`double`	Devuelve el logaritmo natural de `d`.	`math.h`
`log10(d)`	`double`	Devuelve el logaritmo en base 10 de `d`.	`math.h`
`malloc(u)`	`void*`	Reserva `u` bytes de memoria. Devuelve un puntero al principio del espacio reservado.	`stdlib.h`
`pow(d1,d2)`	`double`	Devuelve `d1` elevado a la potencia `d2`.	`math.h`
`printf(...)`	`int`	Escribe datos en el dispositivo de salida estándar de acuerdo a un determinado formato especificado en los restantes argumentos.	`stdio.h`
`putc(c,f)`	`int`	Escribe un carácter en el archivo `f`.	`stdio.h`
`putchar(c)`	`int`	Escribe un carácter en el dispositivo de salida estándar.	`stdio.h`
`puts(s)`	`char*`	Escribe una cadena de caracteres en el dispositivo de salida estándar.	`atdio.h`
`rand()`	`int`	Devuelve un entero positivo aleatorio.	`stdlib.h`
`rewind(f)`	`void`	Mueve el puntero al principio del archivo `f`.	`stdio.h`
`scanf(...)`	`int`	Lee datos en el dispositivo de entrada estándar de acuerdo a un determinado formato especificado en los restantes argumentos.	`stdio.h`

Función	Tipo de salida	Propósito	Archivo de cabecera
`sin(d)`	`double`	Devuelve el seno de `d`.	`math.h`
`sinh(d)`	`double`	Devuelve el seno hiperbólico de `d`.	`math.h`
`sqrt(d)`	`double`	Devuelve la raíz cuadrada de `d`.	`math.h`
`srand()`	`void`	Inicializa el generador de números aleatorios.	`stdlib.h`
`strcmp(s1,s2)`	`int`	Compara dos cadenas de caracteres lexicográficamente. Devuelve un valor negativo si `s1 < s2`; `0` si `s1` y `s2` son idénticas; y un valor positivo si `s1 > s2`.	`string.h`
`strcmpi(s1,s2)`	`int`	Compara dos cadenas de caracteres lexicográficamente, sin diferenciar entre mayúsculas y minúsculas. Devuelve un valor negativo si `s1 < s2`; `0` si `s1` y `s2` son idénticas; y un valor positivo si `s1 > s2`.	`string.h`
`strcpy(s1,s2)`	`char*`	Copia la cadena de caracteres `s2` en la cadena `s1`.	`string.h`
`strlen(s)`	`int`	Devuelve el número de caracteres de una cadena.	`string.h`
`strset(c,s)`	`char*`	Copia todos los caracteres de `s` a `c` (excluyendo el carácter nulo al final `\0`).	`string.h`
`strset(c,s)`	`char*`	Copia todos los caracteres de `s` a `c` (excluyendo el carácter nulo al final `\0`).	`string.h`
`system(s)`	`int`	Pasa la orden al intérprete de comandos. Devuelve `0` si la orden se ejecuta correctamente; en otro caso devuelve un valor distinto de `0`, típicamente `-1`.	`stdlib.h`
`tan(d)`	`double`	Devuelve la tangente de `d`.	`math.h`
`tanh(d)`	`double`	Devuelve la tangente hiperbólica de `d`.	`math.h`
`toascii(a)`	`int`	Convierte el valor del argumento a ASCII.	`ctype.h`
`tolower(c)`	`int`	Convierte una letra a minúscula.	`ctype.h` `stdlib.h`
`toupper(c)`	`int`	Convierte una letra a mayúscula.	`ctype.h` `stdlib.h`

APÉNDICE

C

Recopilación de llamadas al sistema

En este apéndice se recopilan las llamadas al sistema que han ido apareciendo en el texto, más otras adicionales relacionadas de forma significativa con las anteriores o con determinadas órdenes del intérprete de comandos. Conviene recordar que las llamadas al sistema suelen devolver algún resultado. El valor `0` o un valor positivo indican que la llamada se ha ejecutado satisfactoriamente, mientras que el valor `-1` indica que se ha producido algún error. En caso de error, la variable global `errno` contendrá un número que codifica el tipo de error producido. Este número tendrá un significado u otro dependiendo de cada llamada concreta.

- **alarm**

```
unsigned long alarm (unsigned long sec);
```

Activa un temporizador regresivo en tiempo real de `sec` segundos. Cuando el temporizador llega a 0, el proceso que realizó la llamada recibirá la señal SIGALARM.

- **brk, sbrk**

```
int brk (char *endds);
char *sbrk (int incr);
```

Llamadas para cambiar el tamaño del segmento de datos de un proceso. `brk` cambia la posición del fin del segmento de datos, haciendo que tome el valor `endds`. `Sbrk` incrementa el tamaño del mismo segmento en la cantidad de bytes especificados por `incr`.

- **chdir**

```
int chdir (char *path);
```

Cambia el directorio de trabajo actual asociado a un proceso. El nuevo directorio será el asociado a la ruta apuntada por `path`.

- **chmod, fchmod**

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod (char *path, mode_t mode);
int fchmod (int fildes, modet_t mode);
```

Llamadas para cambiar la máscara de modo de un fichero de acuerdo con el valor de `mode`. `chmod` trabaja con la ruta de fichero `path`; y `fchmod` lo hace con el descriptor de un fichero ya

abierto, `fildes`. En ambos casos, el argumento `mode` se puede formar a partir de las siguientes constantes:

`S_ISU1D`	`04000`	Cambiar el identificador de usuario al ejecutar.
`S_1SGID`	`02000`	Cambiar el identificador de grupo al ejecutar.
`S_ISVTX`	`01000`	Mantener el segmento de texto en memoria después de ejecutar.
`S_IRUSR`	`00400`	Permiso de lectura para el propietario.
`S_IWUSR`	`00200`	Permiso de escritura para el propietario.
`S_IXUSR`	`00100`	Permiso de ejecución (búsqueda) para el propietario.
`S_IRGRP`	`00040`	Permiso de lectura para el grupo.
`S_IWGRP`	`00020`	Permiso de escritura para el grupo.
`S_IXGRP`	`00010`	Permiso de ejecución (búsqueda) para el grupo.
`S_IROTH`	`00004`	Permiso de lectura para otros.
`S_IWOTH`	`00002`	Permiso de escritura para otros.
`S_IXOTH`	`00001`	Permiso de ejecución (búsqueda) para otros.

- **chown, fchown**

```
#include <sys/types.h>
chown (char *path, uid_t owner, gid_t group);
fchown (int fildes, uid_t owner, gid_t group);
```

Cambian el propietario y el grupo al que pertenece un fichero de acuerdo con los valores de `owner` (*uid* del nuevo propietario) y grupo (*gid* del nuevo grupo). `chown` trabaja con el nombre de un fichero y `fchown` lo hace con el descriptor de un fichero previamente abierto.

- **chroot**

```
int chroot(char *path);
```

Cambia el directorio raíz asociado a un proceso. El nuevo directorio raíz pasa a ser el referenciado por el puntero `path`.

- **close**

```
int close (int fildes);
```

Libera el descriptor de fichero `fd` y cierra su fichero asociado en el caso de que no haya más procesos que lo tengan abierto.

- **creat**

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int creat (char *path, mode_t mode);
```

Llamada para crear un nuevo fichero cuya ruta está referenciada por `path`. El fichero se creará con la máscara de modo que se especifica en `mode`. Si el fichero ya existe, `creat` trunca su longitud a 0 bytes. Si la llamada se ejecuta con éxito devolverá un descriptor de fichero.

- **dup**

```
int dup (int fildes);
```

Llamada para duplicar una entrada en la tabla de descriptores de ficheros de un proceso, devuelve un nuevo descriptor que va a tener en común con `fildes` los siguientes campos: apuntará al mismo objeto de fichero abierto, tendrá Igual modo de acceso (lectura, escritura, lectura/escritura) e igual indicador de estado. El descriptor devuelto es el menor de entre los que haya disponibles.

- **execl, exev, execle, execve, execlp, execvp**

```
int execl (path, arg0, argl, ... , argn, (char *)0)
char *path, *arg0, *argl, ..., *argn;

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

int execle (path, arg0, argl, ... , argn, (char *)0, envp)
char *path, *arg0, *argl, ...• *argn, *envp[];

int execve (path, argv, envp)
char *path, *argv[], *envp[];

int execlp (file, arg0, argl, ... , argn, (char *)0)
char *file, *arg0, *argl, ..., *argn;

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

La llamada al sistema `exec` sirve para invocar desde un proceso a otro programa ejecutable (programa compilado o shell script). Básicamente `exec` carga las regiones de código, datos y pila del nuevo programa en el contexto de usuario del proceso que la invoca. Existe toda una familia de funciones de librería asociadas a esta llamada al sistema: `execl`, `execv`, `execle`, `execve`, `execlp` y `execvp`.

En todas estas funciones, `path` es la ruta del fichero ejecutable que es invocado. `File` es el nombre de un fichero ejecutable, la ruta del fichero se construye buscando el fichero en los directorios indicados en la variable de entorno `PATH`.

`Arg0`, `arg1`,...,`argN` son punteros a cadenas de caracteres y constituyen la lista de argumentos o parámetros que se le pasa al nuevo programa. Por convenio, al menos `arg0` está presente siempre y apunta a una cadena idéntica a `path` o al ultimo componente de `path`. Para indicar el final de los argumentos siempre a continuación del último argumento `argN` se pasa un puntero nulo `NULL`.

`Envp` es un array de punteros a cadenas de caracteres terminado en un puntero nulo que constituyen el *entorno* en el que se va ejecutar el nuevo programa.

- **exit**

```
#include <stdlib.h>
void exit (int status);
```

Termina la ejecución del proceso que la invoca y devuelve `status` a su proceso padre para que lo pueda examinar para identificar la causa por la que finalizó el proceso hijo de acuerdo a unos criterios que haya previamente establecido el usuario.

- **fork**

```
#include <sys/types.h>
pid_t fork();
```

Crea un nuevo proceso. Si se ejecuta correctamente devuelve al proceso padre el *pid* que le asigne el sistema al proceso hijo. Asimismo devuelve `0` al proceso hijo.

- **getitimer, setitimer**

```
#include <time.h>
getitimer (int wich, struct itimerval *value);
setitimer (int wich, struct itimerval *value,
           struct itimerval *ovalue);
```

Estas llamadas se utilizan para controlar los tres temporizadores asociados a un proceso. `getitimer` se utiliza para leer el estado del temporizador especificado por `wich`. Los valores que puede tomar este argumento son:

`ITIMER_REAL`	Temporizador en tiempo real.
`ITIMER_VIRTUAL`	Temporizador que contabiliza el tiempo que el proceso se ejecuta en modo usuario (tiempo virtual).
`ITIMER_PROF`	Temporizador que contabiliza el tiempo que el proceso se ejecuta en modo usuario y en modo núcleo.

El valor del temporizador es devuelto a través de los campos de la estructura apuntada por `value`. Esta estructura se define como sigue:

```
struct itimerval {
      struct timeval it_interval; /*Intervalo del temporizador. */
      struct timeval it_value; /* Valor actual del temporizador. */
};
```

La estructura `timeval` se define así:

```
struct timeval {
      unsigned long tv_sec; /* Segundos transcurridos desde el día
                               1 de Enero de 1970 */
```

```
        long  tv_usec;    /* Microsegundos. Su rango está comprendido
                             entre O y 999.999 */
};
```

`setitimer` se utiliza para definir el valor del temporizador especificado por `wich`. `Value` es un puntero a una estructura con los nuevos valores del temporizador y `ovalue` es un puntero a una estructura donde se devuelven los antiguos valores del temporizador.

- **getpid, getppid**

```
<sys/types.h>
pid_t getpid ();
pid_t getppid ();
```

`getpid` devuelve el *pid* del proceso que la invoca. Por su parte, `getppid` devuelve el *pid* del proceso padre del proceso que realiza la llamada.

- **gettimeofday, settimeofday**

```
<time.h>
int gettimeofday (struct timeva *tp, struct timezone *tzp);
int settimeofday (struct timeval *tp, struct timezone *tzp);
```

Estas dos llamadas se utilizan para manipular el reloj interno del sistema. Con `gettimeofday` se puede leer la fecha del sistema expresada en segundos y microsegundos con respecto al día 1 de Enero de 1970 GMT. `settimeofday` se utiliza para fijar una nueva fecha del sistema.

`tp` y `tzp` son punteros a estructuras del tipo `timeval` y `timezone`, respectivamente. Estas estructuras se definen como sigue:

```
struct timeval {
        unsigned long tv_sec; /* Segundos desde el día 1 de Enero
                                 de 1970 */
        long  tv_usec;        /* Microsegundos. */
};
struct timezone {
        int tz_minuteswest; /* Corrección con respecto al Meridiano
                               de Greenwich */
        int tz_dsttime;     /* Corrección anual. */
};
```

- **getuid, geteuid, getgid, getegid**

```
<sys/types.h>
uid_tgetuid ();
uid_t geteuid ();
gid_t getgid ();
```

```
gid_t getegid ();
```

Las llamadas al sistema `getuid`, `geteuid`, `getgid` y `getegid` permiten determinar qué valores toman los identificadores *uid*, *euid*, *gid* y *egid*, respectivamente

- **kill**

```
#inc1ude <signa1.h>
int kill (pid_t pid, int sig);
```

Permite a un proceso enviar una señal a otro proceso o a un grupo de procesos. `pid` es un número entero que permite identificar al proceso o conjunto de procesos a los que el núcleo va a enviar una señal. Si `pid` >0, el núcleo envía la señal al proceso cuyo *pid* sea igual a `pid`. Si `pid` = 0, el núcleo envía la señal a todos los procesos que pertenezcan al mismo grupo que el proceso emisor. Si `pid` = -1, el núcleo envía la señal a todos los procesos cuyo *uid* sea igual al *euid* del proceso emisor. Si el proceso emisor que lo envía tiene el *euid* del superusuario, entonces el núcleo envía la señal a todos los procesos, excepto al proceso intercambiador (*pid*=0) y al proceso inicial (*pid*=1). Si `pid` < -1, el núcleo envía la señal a todos los procesos cuyo *gid* sea igual al valor absoluto de `pid`. `sig` es una constante entera que identifica a la señal para la cual el proceso está especificando la acción. También se puede introducir directamente el número asociado a la señal.

- **link**

```
int link (char *path1, char *path2);
```

Crea una entrada de directorio cuya ruta será igual a la cadena apuntada por `path2`. Esta nueva entrada va a ser un enlace con el fichero cuyo nodo-i es el correspondiente a la ruta apuntada por `path1`.

- **lseek**

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek (int fildes, off_t offset, int whence);
```

Permite realizar accesos aleatorios mediante la configuración del puntero de lectura/escritura a un valor especifico. `fildes` es el descriptor del fichero, `offset` es el número de bytes que se va desplazar el puntero y `whence` es la posición desde donde se va desplazar el puntero, que puede tomar los siguientes valores constantes:

`SEEK_SET`	El puntero avanza offset bytes con respecto al inicio del fichero. El valor de esta constante es `0`.
`SEEK_CUR`	El puntero avanza offset bytes con respecto a su posición actual. El valor de esta constante es `1`.
`SEEK_END`	El puntero avanza offset bytes con respecto al final del fichero. El valor de esta constante es `2`.

Si `offset` es un número positivo, los avances deben entenderse en su sentido natural; es decir, desde el inicio del fichero hacia el final del mismo. Sin embargo, también se puede conseguir que el puntero retroceda pasándole a `lseek` un desplazamiento negativo.

- **mkdir**

```
int mkdir (char *path, mode_t mode);
```

Crea un fichero de directorio con una ruta igual a la cadena apuntada por `path`. `mode` codifica los la máscara de modo del directorio.

- **mknod**

```
#include <sys/types.h>
#include <sys/stat.h>
int mknod (char *path, mode_t mode, dev_t dev);
```

Crea un fichero de dispositivo, una tubería con nombre o fichero FIFO o un directorio. `path` apunta a una cadena de caracteres que contiene la ruta del fichero a crear. `mode` es la máscara de modo del fichero, en la que sólo uno de los siguientes bits deberá estar activo:

`S_IFIFO`	Crear una tubería con nombre o fichero FIFO.
`S_IFCHR`	Crear un fichero de dispositivo modo carácter.
`S_IFBLK`	Crear un fichero de dispositivo modo bloque.
`S_IFDIR`	Crear un directorio.

Los 9 bits menos significativos de `mode` codifican los permisos del fichero. Si el fichero a crear es un dispositivo, `dev` debe codificar los números principal y secundario del mismo.

- **mount**

```
int mount (char *spec, char *dir, int rwflag);
```

Monta un sistema de ficheros en un determinado directorio. `spec` es la ruta de acceso del fichero del dispositivo del disco donde se encuentra el sistema de ficheros que se va a montar, `dir` es la ruta de acceso del directorio sobre el que se va a montar el sistema de ficheros y `rwflags` es una máscara de bits que permite especificar diferentes opciones. En concreto el bit menos significativo de `flags` se utiliza para revisar los accesos de escritura sobre el sistema de ficheros. Si vale 1, la escritura estará prohibida, por lo que sólo se podrán hacer accesos de lectura; en caso contrario, la escritura estará permitida, pero de acuerdo a los permisos individuales de cada fichero.

- **msgctl**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl (int msqid, int cmd, struct msqid_ds *buf);
```

Permite leer y modificar la información estadística y de control de una cola de mensajes. `msqid` es el identificador de la cola, `cmd` es un número entero o una constante simbólica que especifica la operación a efectuar y `buf` es una estructura del tipo predefinido `msqid_ds` que contiene los argumentos de la operación. Si la llamada `msgctl` tiene éxito, en `resultado` se almacenará un número entero cuyo valor depende del comando `cmd`. Si falla en `resultado` se almacenará el valor –1. (Más información relativa a esta llamada al sistema en la sección 7.3.3).

- **msgget**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget (key_t key, int msgflg);
```

Crea una cola de mensajes o bien permite acceder a una cola ya existente usando una clave dada. `key` es la clave de la cola de mensaje y `msgflg` es una máscara de indicadores (ver sección 7.3.1.4). Si la llamada al sistema `msgget` se ejecuta con éxito entonces en `msqid` se almacenará el identificador entero de una cola de mensajes asociada a la llave `key`. En caso contrario en `msqid` se almacenará el valor -1.

- **msgrcv**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgrcv (int msqid, void *msgp, int msgsz, long msgtyp,
            int msgflg);
```

Extrae un mensaje de una determinada cola de mensajes. `msqid` es un identificador de una cola de mensajes, `msgp` es un puntero a la variable del espacio de direcciones del usuario donde se va almacenar el mensaje, `msgsz` es la longitud del texto del mensaje en bytes, `msgtyp` indica el tipo del mensaje que se desea extraer. Si `msgtyp`=0 se extrae el primer mensaje que haya en la cola independientemente de su tipo. Corresponde al mensaje más viejo. Si `msgtyp` > 0 se extrae el primer mensaje del tipo `msgtyp` que haya en la cola. Por último si `msgtyp` < 0 se extrae el primer mensaje que cumpla que su tipo es menor o igual al valor absoluto de `msgtyp` y a la vez sea el más pequeño de los que hay. `msgflg` es una máscara de indicadores que permite especificar el comportamiento del proceso receptor en caso de que no pueda extraerse ningún mensaje del tipo especificado. Si la llamada al sistema tiene éxito en `resultado` se almacenará el número de bytes del mensaje recibido (este número no incluye los bytes asociados al tipo de mensaje). En caso de error en `resultado` se almacenará el valor -1.

- **msgsnd**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd (int msqid, void *msgp, int msgsz, int msgflg);
```

Envía un mensaje a una determinada cola de mensajes. `msqid` es un identificador de una cola de mensajes, `msgp` puntero a la variable del espacio de direcciones del usuario que contiene el mensaje que se desea enviar, `msgsz` es la longitud del texto del mensaje en bytes y `msgflg` es una máscara de indicadores que permite especificar el comportamiento del proceso emisor en caso de que no pueda enviarse el mensaje debido a una saturación del mecanismo de colas.

- **nice**

```
int nice (int incr);
```

Permite aumentar o disminuir el factor de amabilidad actual del proceso que la invoca. `incr` es una variable entera que puede tomar valores entre -20 y 19. El valor de incremento será sumado al valor del factor de amabilidad actual. Sólo el superusuario puede invocar a `nice` con valores de `incremento` negativos.

- **open**

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open (char *path, int oflag [, mode_t mode]);
```

Permite abrir un fichero ya existente. `path` es la ruta del fichero y `flags` puede ser o una máscara de modo octal o una máscara de bits que permiten especificar los permisos de apertura de dicho fichero. Cuando el argumento `flags` se especifica mediante una máscara de bits, ésta típicamente se implementa como una combinación de constantes enlazadas con el operador OR a nivel de bit (‘|’). Algunas de las constantes utilizadas más frecuentemente son:

`O_RDONLY`	Abrir en modo sólo lectura.
`O_WRONLY`	Abrir en modo sólo escritura.
`O_RDWR`	Abrir para leer y escribir.
`O_CREAT`	Crear el fichero si no existe con la máscara de modo especificada en `mode`.
`O_APPEND`	Situar el puntero de lectura/escritura al final del fichero para añadir datos.
`O_TRUNC`	Si el fichero existe, trunca su longitud a cero bytes, incluso si el fichero se abre para leer.

De las constantes `O_RDONLY`, `O_WRONLY` y `O_RDWR` solo una de ellas debe estar presente al componer la máscara `flags`, de lo contrario, el modo de apertura quedaría indefinido. Si `open` se ejecuta con éxito devuelve de un descriptor de fichero.

- **pause**

```
pause();
```

Hace que el proceso que la invoca quede a la espera de la recepción de una señal que no ignore o que no tenga bloqueada.

- **pipe**

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

Crea una tubería sin nombre y devuelve dos descriptores a través de los cuales se puede leer y escribir en la tubería. Para leer de la tubería hay que usar el descriptor `fildes[0]`, mientras que para escribir en la tubería hay que usar el descriptor `fildes[1]`.

- **ptrace**

```
#include <sys/ptrace.h>
int ptrace (int request, int pid, int addr, int data);
```

Permite a un proceso padre (proceso depurador) controlar la ejecución de un proceso hijo (proceso depurado). `pid` es el *pid* del proceso hijo, `addr` se refiere a una posición en el espacio de direcciones del hijo y la interpretación del argumento `data` depende de `request`. El argumento `request` permite al padre realizar las siguientes operaciones:

`0`	Habilita la depuración en el proceso hijo (éste parámetro lo utiliza sólo el proceso hijo).
`1, 2`	Devuelve el contenido de la dirección de memoria virtual referenciada por `addr` en el proceso hijo.
`3`	Devuelve el contenido de la posición `adrr` del área de usuario del proceso hijo.
`4, 5`	Escribe, con el valor `data`, el contenido de la dirección de memoria virtual referenciada por `addr` en el proceso hijo.
`6`	Escribe, con el valor `data`, en la posición `addr` del área de usuario del proceso hijo.
`7`	Le indica al proceso hijo que continúe con su ejecución. El proceso hijo estará parado debido a la recepción de una señal.
`8`	Fuerza a que el proceso hijo termine su ejecución con una llamada al sistema `exit`.
`9`	Le indica al proceso hijo que continúe su ejecución, pero activando el bit de traza del procesador. Esto hace que el proceso interrumpa su ejecución después de cada instrucción máquina. Esta orden se emplea para ejecutar un proceso paso a paso.

- **raise**

```
#include <signal.h>
int raise(int sig);
```

Permite a un proceso enviarse una señal a sí mismo. `sig` es una constante entera que identifica a la señal que se desea enviar. También se puede introducir directamente el número asociado a dicha señal.

- **read**

```
int read (int fildes, char *buf, unsigned nbyte);
```

Permite leer datos de un fichero. `fildes` es el descriptor de fichero, `buf` es el array de caracteres donde se almacenarán los datos que se lean en el fichero y `nbyte` es el número de bytes que se desea leer. Si la llamada al sistema se ejecuta devuelve el número de bytes leídos.

- **rename**

```
#include <stio.h>
rename (const char *source, const char *target);
```

Hace que el fichero cuya ruta indica `source` pase a llamarse como indica `target`.

- **rmdir**

```
rmdir (char *path);
```

Borra el directorio cuyo nombre viene indicado por `path`. Para que pueda ser borrado no debe contener ningún fichero.

- **semctl**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl (int semid, int semnum, int cmd,
      union semun
      {
            int val;
            struct semid ds *buf;
            ushort *array;
      }arg);
```

Permite acceder a la información administrativa y de control que posee el núcleo sobre un cierto conjunto de semáforos. `semid` es el identificador de un array o conjunto de semáforos, `semnum` es el identificador de un semáforo concreto dentro del array, `cmd` es un número entero o una constante simbólica (ver Tabla 7.1) que especifica la operación que va a realizar la llamada al sistema `semctl` y `arg` se utiliza para almacenar los argumentos o los resultados de la operación.

- **semget**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget (key_t key key, int nsems nsems, int semflg);
```

La llamada al sistema `semget` crea u obtiene un array o conjunto de semáforos. `key` es una llave numérica del tipo predefinido `key_t` o bien la constante IPC_PRIVATE que obliga a crear un nuevo identificador, `nsems` es el número entero de semáforos del conjunto o array asociados a `key` y `semflg` es una máscara de indicadores (máscara de bits). Estos indicadores permiten especificar, de forma similar a como se hace para los ficheros, los permisos de acceso al conjunto de semáforos. Si la llamada al sistema `semget` se ejecuta con éxito devolverá el identificador entero de un array o conjunto de `count` semáforos asociados a la llave `key`. Si no existe un conjunto de semáforos asociado a la llave la orden fallará y en `semid` se almacenará el valor –1 a menos que se haya realizado con el indicador IPC_CREAT de `flags` activo, lo que fuerza a crear un nuevo conjunto de semáforos. También se crea un nuevo conjunto de semáforos si el parámetro `key` se configura al valor IPC_PRIVATE.

- **semop**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop (int semid, struct sembuf *sops, int nsops);
```

Realiza operaciones sobre los elementos de un determinado conjunto de semáforos. `semid` es un identificador de un array o conjunto concreto de semáforos, `sops` es un puntero a un array de estructuras del tipo `sembuf` (ver sección 7.3.2.2) que indican las operaciones que se van a llevar a cabo sobre los semáforos y `nsops` es el número total de elementos que tiene el array de operaciones, es decir, el número total de operaciones.

- **setuid, setgeid**

```
#include <sys/types.h>
int setuid (uid_t uid);
int setgid (gid_t gid);
```

La llamada al sistema `setuid` permite asignar el valor `uid` al *euid* y al *uid* del proceso que invoca a la llamada. Si el identificador de usuario efectivo del proceso que efectúa la llamada es el del superusuario, entonces en este caso *uid*=`uid` y *euid*=`uid`. Si el identificador del usuario efectivo del proceso que efectúa la llamada no es el del superusuario, entonces en este caso *euid*=`uid` si el valor del parámetro `uid` coincide con el valor del *uid* del proceso o si esta llamada se está invocando dentro de la ejecución de un programa que tiene su bit `S_ISUID` activado y el valor del parámetro `uid` coincide con el valor del *uid* del propietario del programa.

La explicación del funcionamiento de la llamada al sistema `setguid` es análoga a la explicada para `setuid` pero referido a los identificadores *gid* y *egid*.

- **shmctl**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl (int shmid, int cmd, struct shmin_ds *buf);
```

Permite realizar operaciones de control sobre una zona de memoria compartida creada previamente por `shmget`. `shmid` es el identificador de una región de memoria compartida, `cmd` es un número entero o una constante simbólica que especifica la operación a efectuar y `buf` es un puntero a una estructura del tipo predefinido `shmid_ds` que contiene los argumentos de la operación. (Más información en la sección 7.3.4.5).

- **shmget**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget (key_t key, int size, int shmflg);
```

Crea un segmento de memoria compartida o accede a uno que ya existe. `key` es la clave de acceso a un segmento de memoria compartida, `size` especifica el tamaño en bytes del segmento de memoria solicitado y `shmflg` es una máscara de indicadores (ver sección 7.3.1.4). Si la llamada al sistema se ejecuta con éxito devolverá el identificador entero de la zona de memoria compartida asociada a la llave `key`.

- **shmat**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
char *shmat (int shmid, char *shmaddr, int shmflg);
```

Asigna un espacio de direcciones virtuales al segmento de memoria cuyo identificador `shmid` ha sido dado por `shmget`. Por lo tanto `shmat` enlaza una región de memoria compartida de la tabla de regiones con el espacio de direcciones de un proceso. `shmid` es un identificador de una región de memoria compartida, `shmaddr` es la dirección virtual del proceso donde se desea que empiece la región de memoria compartida. Si `shmaddr` = 0, el sistema selecciona la dirección. Es la opción más adecuada si se desea conseguir portabilidad. Si `shmaddr` $\neq$ 0, el valor de la dirección devuelto depende si se especificó o no el bit SHM_RND del parámetro `shmflg`. Si se especificó el segmento de memoria es enlazada en la dirección especificada por el parámetro `shmaddr` redondeada por la constante SHMLBA (SHare Memory Lower Boundary Address). En caso

contrario el segmento de memoria es enlazado en la dirección especificada por el parámetro `shmaddr`. `shmflg`, es una máscara de bits que indica la forma de acceso a la memoria. Si el bit SHM_RDONLY está activo, la memoria será accesible para leer, pero no para escribir. Por defecto un segmento de memoria se comparte para lectura y escritura. Si la llamada al sistema `shmat` tiene éxito devuelve la dirección a la que está unido el segmento de memoria compartida `shmid`.

- **shmdt**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmdt (char *shmaddr);
```

Desenlaza un segmento de memoria compartida del espacio de direcciones de un proceso. `shmaddr` es la dirección virtual del segmento de memoria compartida que se quiere separar del proceso.

- **sigblock**

```
#include <signal.h>
long sigblock (long mask);
```

Permite añadir nuevas señales bloqueadas a la máscara actual de señales. `mask` que es un entero largo que se utilizará como operando junto con la máscara actual de señales para realizar una operación lógica de tipo OR a nivel de bits. Se considera que la señal número *j* está bloqueada si el j-ésimo bit de `mask` está a 1. Este bit puede ser fijado con la macro `sigmask(j)`. Si la llamada se ejecuta con éxito devuelve la máscara de señales que se tenía especificada antes de ejecutar esta llamada al sistema

- **signal**

```
#include <signal.h>
void (*signal (int sig, void (*action)()))();
```

Permite especificar el tratamiento de una determinada señal recibida por un proceso. `sig` es una constante entera que identifica a la señal para la cual el proceso está especificando la acción. También se puede introducir directamente el número asociado a la señal. `action` este parámetro especifica la acción que se debe realizar cuando se trate la señal, puede tomar los siguientes valores:

`SIG_DFL`	Constante entera que indica que la acción a realizar es la acción por defecto asociada a dicha señal.
`SIG_IGN`	Constante entera que indica que la señal se debe ignorar.
`dirección`	Es la dirección del manejador de la señal definido por el usuario.

Si `signal` se ejecuta con éxito devuelve la acción que tenía asignada dicha señal antes de ejecutar esta llamada al sistema. Este valor puede ser útil para restaurarlo en cualquier instante

posterior. Por otra parte, si se produce algún error durante la ejecución de la llamada al sistema `resultado` tomará el valor `SIG_ERR` (constante entera asociada al valor -1).

- **sigpause**

```
#include <signal.h>
long sigpause (long mask);
```

Bloquea la recepción de señales de acuerdo con el valor de `mask`, de la misma forma que hace `sigsetmask`. A continuación se pone a esperar a que llegue alguna de las señales no bloqueadas. Si no se desea que `sigpause` bloquee ninguna señal, se le debe pasar la máscara `OL`.

Cuando `sigpause` termina su ejecución, restaura la máscara de señales que había antes de llamarla. La ejecución de `sigpause` termina cuando es interrumpida por una señal. Después de tratar la señal, `sigpause` hace que `errno` tome el valor EINTR y devuelve el valor -1.

- **sigsetmask**

```
#include <signal.h>
long sigsetmask (long mask);
```

Fija la máscara actual de señales, es decir, permite especificar qué señales van a estar bloqueadas. Obviamente, aquellas señales que no pueden ser ignoradas ni capturadas, tampoco van a poder ser bloqueadas. `mask` que es un entero largo asociado a la máscara de señales. Se considera que la señal número *j* está bloqueada si el j-ésimo bit de `mask` está a 1. Este bit puede ser fijado con la macro `sigmask(j)`. Si la llamada se ejecuta con éxito devuelve la máscara de señales que se tenía especificada antes de ejecutar esta llamada al sistema.

- **sigvector**

```
#include <signal.h>
sigvector (int sig, struct sigvec *vec, struct sigvec *ovec);
```

`sigvector` se utiliza para especificar la forma de tratar una señal. `sig` es una constante entera que identifica a la señal para la cual el proceso está especificando la acción. También se puede introducir directamente el número asociado a la señal.

Tanto `vec` como `ovec` son punteros a estructuras del tipo `sigvec`. Esta estructura está definida con los siguientes campos:

```
struct sigvec
{
       void (*sv_handler) ();
       long sv_mask;
       long sv_flags;
};
```

El campo `sv_handler` es un puntero a una función que devuelve `void` y tiene el mismo significado que el parámetro `action` de la llamada `signal`. Este campo se utiliza para indicar cuál será la rutina de tratamiento de la señal. Al igual que en el caso de `signal`, puede tomar tres tipos de valores con diferente significado:

`SIG_DFL`	Constante entera que indica que la acción a realizar es la acción por defecto asociada a dicha señal.
`SIG_IGN`	Constante entera que indica que la señal se debe ignorar.
`dirección`	Es la dirección del manejador de la señal definido por el usuario.

El campo `sv_mask` codifica, en cada uno de sus bits, las señales que no se desen que sean tratadas si son recibidas mientras se está ejecutando la rutina de tratamiento actual. Normalmente, este campo vale 0, lo que indica que mientras se está tratando una señal, cualquier otra señal puede interrumpir. Si alguno de los bits de `svmask` vale 1, se impide el anidamiento cuando se recibe esa señal.

El campo `sv_flags` codifica cuál va a ser la semántica (significado) que se emplee en la recepción de la señal. Los siguientes bits están definidos para este campo:

`SV_BSDSIG`	Usar la semántica de BSD. Esto significa, entre otras cosas, que cuando se instala una rutina de tratamiento, permanecerá instalada hasta que se haga otra llamada a `sigvector` que instale una rutina nueva.
`SV_RESETHAND`	Impone las misma semántica que la llamada `signal` del UNIX System V. Es decir, siempre se restaura la rutina de tratamiento por defecto antes de entrar en la rutina de tratamiento suministrada por el usuario.

`vec` apunta al que va a ser el nuevo vector de señal y `ovec` devuelve **un** puntero al vector que había, por si desea restaurarlo posteriormente.

- **stat, fstat**

```
#include <sys/types.h>
#include <sys/stat.h>
int stat (char *path, struct stat *buf);
int fstat (int fildes, struct stat *buf);
```

`stat` devuelve, a través de `buf`, información sobre el estado del fichero cuya ruta es `path`. `fstat` hace lo mismo con el fichero descrito por `fildes`. La estructura de `buf` es la siguiente:·

```
struct stat {
      dev_t st_dev     /* Número de dispositivo del disco donde
                          se encuentra el fichero. * /
      ino_t st_ino     /* Nodo-i del fichero. */
      ushort st_mode   /* Máscara de modo*/
      ushort st_nlink* /* Número de enlaces al fichero. */
      uid_t st_uid     /* Identificador de usuario (uid) del
                          propietario del fichero. */
```

```
gid_t st_gid     /* Identificador del grupo (gid) del
                    propietario del fichero. */
dev_t st_rdev    /* Número principal y número secundario.
                    Tiene significado únicamente para los
                    ficheros especiales. */
off_t st_size    /* Tamaño, en bytes, de un fichero.*/
time_t st_atime  /* Fecha del último acceso al fichero
                    (lectura).*/
time_t st_mtime  /*Fecha de la última modificación del
                   fichero.*/
time_t st_ctime  /*Fecha del último cambio de la información
                   administrativa del fichero*/
}
```

- **stime**

```
int stime (long *tp);
```

Permite fijar la fecha y la hora actuales del sistema con el valor apuntado por `tp` contiene los segundos transcurridos desde las 00:00:00 GMT del día 1 de enero de 1970.

- **sync**

```
void sync();
```

Copia en el disco aquellos bloques de la caché de buffers de bloques de disco cuyo contenido ha sido modificado. Se asegura de este modo la consistencia de los datos del sistema.

- **time**

```
#include <time.h>
time_t time (time_t *tloc);
```

Permite leer la fecha y la hora actuales que almacena el sistema. Si la llamada se ejecuta con éxito en `tloc` se almacenarán los segundos transcurridos desde las 00:00:00 GMT del día 1 de enero de 1970. Además esta llamada también devuelve esta misma información.

- **times**

```
#include <sys/times.h>
clock_t times (struct tms *buffer);
```

Permite conocer el tiempo empleado por un proceso en su ejecución. Si `times` se ejecuta con éxito almacena en `tbuffer` la información estadística relativa a los tiempos de ejecución empleados por el proceso, desde su inicio hasta el momento de invocar a `times`. (Más información en sección 6.2.4.3).

- **umount**

```
int umount(char *name);
```

Desmonta el sistema de ficheros que se encuentra montado sobre el fichero de dispositivo indicado en `name`.

- **unlink**

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

Borra la entrada del directorio especificada en la ruta apuntada por `path`.

- **wait**

```
pid_t wait (int *stat_loc);
```

Suspende la ejecución del proceso actual hasta que alguno de sus procesos hijos finalice. `stat_loc` es la dirección de una variable entera donde se almacenará el *código de retorno para el proceso padre* generado por el algoritmo `exit()` al terminar un proceso hijo. Si la llamada se ejecuta con éxito devuelve el *pid* del proceso hijo que ha terminado.

- **write**

```
int write (int fildes, ehar *buf, unsigned nbyte);
```

Esta llamada permite escribir datos en un fichero. `fildes` es el descriptor de fichero, `buf` es el array de caracteres donde se encuentran almacenados los datos que se van a escribir en el fichero y `nbyte` es el número de bytes que se desea escribir. Si `write` se ejecuta con éxito, devuelve el número de bytes escritos.