Sistemas Operativos: Llamadas Al Sistema

Andrés Herrera Poyatos

13/10/2014

Introducción

Este manual contiene todas las llamadas al sistema del lenguaje *c* que se estudian en la asignatura **Sistemas Operativos** de segundo de carrera en Ingeniería Informática de la Universidad de Granada. El objetivo del mismo es proporcionar una de herramienta de consulta rápida, concisa y en Español, evitando tener que buscar en el **man** de **Linux** o en la multitud de páginas que conforman los guiones de **Sistemas Operativos**. Sin embargo, para una información más clara y detallada se recomiendan estos últimos medios.

Terminología

- ***fd* = *file descriptor* = *descriptor de archivo*** = Entero positivo que identifica un archivo. La entrada, salida y salida de error estándar están identificadas con los descriptores de archivo 0, 1 y 2 respectivamente. En la librería se encuentran las constantes símbolicas STDIN_FILENO, STDOUT_FILENO y STDERR_FILENO que identifican los descriptores de archivo anteriores.
- ***offset* = *current file offset*** = Entero no negativo característico de un archivo que mide mediante el cual se indica el próximo Byte en el que se va a leer o escribir. Contiene, por tanto, el número de Bytes ya leídos o escritor en el archivo. El ***offset*** aumenta mediante operaciones de lectura y escritura sobre el archivo. También se puede cambiar manualmente mediante las llamadas al sistema recogidas como **seek**. Por defecto, el ***offset*** es inicializado a 0 al abrir el archivo a no ser que se indique lo contrario.
- ***Archivo Regular*** = Archivo que contiene datos de cualquier tipo.
- ***Archivo de directorio*** = Archivo con nombres de otros archivos y punteros a los mismos.

- ***Pathname*** = nombre y dirección de un archivo o directorio. **Ejemplo:** `/home/andreshp/Universidad/`

Instrucciones Básicas

Se proporcionan algunas instrucciones básicas para el uso de llamadas al sistema con *c*.

perror

Resumen: Imprime un mensaje de error del sistema.

```
#include <stdio.h>

void perror(const char *s);

#include <errno.h>

const char *sys_errlist[];
int sys_nerr;
int errno;
```

La función **perror** produce un mensaje de error en la salida de error estándar. Este mensaje describe el último error que se a encontrado al realizar una llamada al sistema o utilizar una función de una librería.

El mensaje mostrado consiste de la cadena proporcionada al llamar a perror, *s*, más dos puntos seguidos del error encontrado y un salto de línea. Lo habitual es que la cadena *s* contenga un mensaje con el lugar del código donde se produjo el error para su posterior detención.

El error que tuvo lugar es identificado mediante la variable externa *errno*. Cuando se produce un error se asigna un entero a *errno* que permite identificarlo. Esta variable no se "limpia" aunque posteriores llamadas al sistema tengan éxito.

La variable global *sys_errlist[]* consiste en un array de cadenas de caracteres cada una de las cuales describe el error asociado a su índice. De esta forma, cuando se llama a **perror** se devuelve la cadena cuyo índice se corresponde con *errno* en ese momento. *sys_nerr* indica la longitud del array *sys_errlist[]* y, por tanto, el número de errores identificables. No se recomienda acceder a esta lista de forma independiente por posibles diferencias entre versiones.

Sesion 1: Entradas y Salidas de Datos

En esta sesión se trabajan aquellas llamadas al sistema que permiten el manejo de archivos y la entrada y salida de datos mediante los mismos.

open y creat

Resumen: Abren o crean un archivo.

```
# include <sys/types.h>
# include <sys/stat.h>
# include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);
```

Dado un *pathname* de un archivo, **open** devuelve el *fd* del mismo para poderlo utilizar con posteriores llamadas al sistema como **read**, **write**, **lseek**, etc. El *fd* sería el menor entero positivo no utilizado para esta labor.

Por defecto, el nuevo *fd* permanece abierto y el *offset* apunta al inicio del archivo.

Una llamada a **open** genera una entrada en la tabla del sistema de archivos abiertos, entrada indicada por el *fd* siendo éste una referencia a la misma. Esta entrada almacena el *offset* del archivo y las banderas con el estatus del mismo. El *fd* dado no es afectado por un cambio en el *pathname* del archivo. No es compartido con ningún otro proceso salvo vía **fork**.

Argumentos:

El argumento *flags* consiste en un entero con las diferentes opciones de apertura para el archivo. Estas opciones se indican mediante constantes símbolicas operadas mediante OR. Debe incluir obligatoriamente uno de los siguientes modos de acceso: O_RDONLY, O_WRONLY, o O_RDWR. Estos modos piden al archivo permisos de solo lectura, solo escritura o ambos, respectivamente.

Adicionalmente, 0 o más *flags* de creación de archivo y estatus pueden ser aplicados mediante una operación OR lógica. Algunos *flags* son:

- O_APPEND : El archivo se abre en modo “append”. Esto provoca el siguiente hecho: antes de cada **write** el *offset* se sitúa al final del archivo de forma que siempre se escribe al final del mismo.
- O_CREAT : Crea el archivo si este no existe al llamar a **open**. En caso de ser especificado se deben indicar los permisos con los que se creará el archivo en el argumento *mode*. Los permisos efectivos son modificados por **umask** de la forma: `mode & !umask`. Si O_CREAT no es especificado entonces *mode* es ignorado. Los posibles permisos son:

  - S_IRWXU = 00700 = *user* (dueño del archivo) tiene permisos de lectura, escritura y ejecución.
  - S_IRUSR = 00400 = *user* tiene permiso de lectura.
  - S_IWUSR = 00200 = *user* tiene permiso de escritura.
  - S_IXUSR = 00100 = *user*espondiente. tiene permiso de ejecución.
  - S_IRWXG = 00070 = *group* tiene permisos de lectura, escritura y ejecución.
  - S_IRGRP = 00040 = *group* tiene permiso de lectura.
  - S_IWGRP = 00020 = *group* tiene permiso de escritura.
  - S_IXGRP = 00010 = *group* tiene permiso de ejecución.
  - S_IRWXO = 00007 = *others* tienen permisos de lectura, escritura y ejecución.
  - S_IROTH = 00004 = *others* tiene permiso de lectura.
  - S_IWOTH = 00002 = *others* tiene permiso de escritura.
  - S_IXOTH = 00001 = *others* tiene permiso de ejecución.

- O_TRUNC : Si el archivo ya existe, es regular y se permite escritura, entonces se trunca a longitud 0, eliminado su contenido.

Otros posibles *flags* son: O_CLOEXEC, O_DIRECTORY, O_EXCL, O_NOCTTY, O_NOFOLLOW, y O_TTY_INIT. Más información sobre los *flags* en el **man**.

Llamada creat: **creat** es equivalente a **open** con *flags* igual a O_CREAT|O_WRONLY|O_TRUNC.

Return: En caso de éxito se devuelve el *fd* del archivo. Si tuvo lugar algún error se devuelve -1 y *errno* toma el valor correspondiente.

read

Resumen: Lee desde un descriptor de archivo.

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

Lee el número de bytes dado por *count* del archivo *fd* y lo asigna a *buf*. En archivos que soporten "seeking" (**lseek**) la operación de lectura comienza en el actual *offset* del archivo, y el *offset* del archivo es incrementado por el número de bytes leído. Si el offset actual se ha pasado el final del archivo no se leen bytes y **read** devuelve cero.

Posibles errores (en caso de haber un error se devuelve -1):

- Si *count* es 0 se puede producir algún error.
- Si *count* es mayor que SIZE_MAX el resultado no es especificado.

Return: En caso de éxito se devuelve el número de bytes leídos. Si el número devuelto es menor que el número de Bytes que se ha pedido leer no es un error

sino que o bien se ha alcanzado el final del fichero y no hay más bytes disponibles o bien se ha finalizado el proceso antes de tiempo. Si tuvo lugar algún error se devuelve -1 y *errno* toma el valor correspondiente.

write

Resumen: Escribe en una salida dada por un descriptor de archivo.

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

Escribe hasta *count* bytes desde el buffer del cual se proporciona el puntero *buf* en el archivo indicado por su descriptor de archivo *fd*.

El número de bytes escritos puede ser menor que *count* en caso de que no haya suficiente espacio en el medio físico o que la llamada haya sido interrumpida por una señal.

Para un archivo con seeking (**lseek**) se escribe el contenido de *buf* empezando en donde apunte el *offset*, siendo este incrementado por el número de bytes escritos tras la ejecución del *write*. Si el archivo fue abierto con *open* utilizando la opción O_APPEND, entonces el *offset* se sitúa al final del archivo antes de la escritura.

Return: En caso de éxito en la escritura, se devuelve el número de bytes que fueron escritos. Si por el contrario hubo un error, se devuelve -1 y a *errno* se le asigna el error correspondiente.

lseek

Resumen: Permite reposicionar el *offset* de un archivo.

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

La llamada al sistema **lseek** cambia el valor del *offset* del archivo indicado por *fd* al valor dado por el argumento *offset* de acuerdo con la directiva *whence*.

lseek permite asignar al *offset* del archivo un valor posterior al final del mismo. El valor *size* del archivo no cambia a no ser que se escriba con **write**, en cuyo caso se llenan los bytes entre *size* y el valor de *offset* con '\0' (esto se llama agujero o hole).

Argumento whence:

- SEEK_SET : El *offset* del archivo se cambia al valor dado por *offset*.

- SEEK_CUR : El *offset* del archivo se mantiene en el sitio actual más el número de Bytes indicado por *offset*.
- SEEK_END : The *offset* se cambia al final del archivo más el número de Bytes indicado por *offset*.
- SEEK_DATA : Adjusta el *offset* a la siguiente localización del archivo mayor o igual que el dato contenido por *offset*. Si *offset* apunta a datos, entonces se asigna su valor al *offset* del archivo.
- SEEK_HOLE : Asjusta el *offset* del archivo al siguiente agujero (hole, zona del archivo con '\0' consecutivos) en el archivo mayor o igual que *offset*. Si *offset* apunta a la mitad de un agujero entonces al *offset* del archivo se le asigna esta posción. Si no hay un agujero después de *offset* entonces el *offset* del archivo se sitúa al final del mismo (hay un agujero ímplicito al final de cualquier archivo).

Return: En caso de éxito, **lseek** devuelve la localización del *offset* medida en bytes desde el principio del archivo. En caso de error, se devuelve (off_t) -1 y *errno* indica el correspondiente error.

close

Resumen: Cierra un descriptor de archivo.

```
#include <unistd.h>

int close(int fd);
```

close cierra un descriptor de archivo de forma que este ya no refiere a ningún archivo. Cuaquier bloqueo en el archivo indicado por *fd* es removido.

Si el *fd* es el último descriptor de archivo apuntando a un archivo abierto por **open** los recursos asociados al mismo son liberados. Si, además, el archivo había sido objetivo de la llamda **unlink** se elimina tras **close**.

Return: En caso de éxito, **close** devuelve 0. En caso de error, se devuelve -1 y *errno* indica el correspondiente error.

stat

Resumen: Obtiene el estatus de un archivo.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int stat(const char *path, struct stat *buf);
int lstat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
```

Estas funciones devuelven información sobre un archivo. No se requieren permisos en el archivo para ejecutar las funciones (aunque en el caso de **lstat** y **stat** sí se requieren permisos en todos los directorios del *path* del archivo).

stat devuelve las características del archivo dado por *path*, almacenándolas en la estructura *stat* a la que apunta *buf*.

lstat es idéntica a **stat**, excepto que si *path* es un enlace simbólico entonces el link es el archivo del cual se obtienen sus características y no el archivo al que se refiere, como sí sucede con **stat**.

fstat es idéntica a **stat**, excepto que el archivo es indicado por *fd* en lugar de por su *path*.

Todas estas llamadas obtienen una estructura *stat* como la que sigue:

```
struct stat {
    dev_t     st_dev;     /* ID del dispositivo que contiene el archivo */
    ino_t     st_ino;     /* Número de inodo */
    mode_t    st_mode;    /* Protección del archivo */
    nlink_t   st_nlink;   /* Número de enlaces duros */
    uid_t     st_uid;     /* ID de usuario del dueño del archivo */
    gid_t     st_gid;     /* ID del grupo del dueño del archivo */
    dev_t     st_rdev;    /* ID del dispositivo  (si es un archivo especial) */
    off_t     st_size;    /* Tamaño total en bytes */
    blksize_t st_blksize; /* Tamaño de bloque para E/S del sitema */
    blkcnt_t  st_blocks;  /* Número de bloques de 512B del archivo */
    time_t    st_atime;   /* Tiempo del último acceso */
    time_t    st_mtime;   /* Tiempo de la última modificación */
    time_t    st_ctime;   /* Tiempo del último cambio de estado               */
};
```

Se proporcionan, además, las siguientes macros para comprobar el tipo de fichero utilizando *st_mode* del *stat* obtenido:

- S_ISLNK(st_mode) : Comprueba si es un enlace simbólico (soft).
- S_ISREG(st_mode) : Comprueba si es un archivo regular.
- S_ISDIR(st_mode) : Comprueba si es un directorio.
- S_ISCHR(st_mode) : Comprueba si es un dispositivo de caracteres.
- S_ISBLK(st_mode) : Comprueba si es un dispositivo de bloques.
- S_ISFIFO(st_mode) : Comprueba si es una cauce con nombre (FIFO).
- S_ISSOCK(st_mode) : Comprueba si es un socket.

El valor *st_mode* codifica además los permisos del archivo, independientemente del tipo de archivo que se trate, pudiendo comprobar si el archivo tiene determinados permisos mediante `st_mode & S_IPERMISO`, donde S_IPERMISO responde a un posible permiso de los indicados en **open**. *st_mode* utiliza 16 bits para esta codificación siendo los 12 últimos dedicados a permisos y los 4 anteriores al tipo de archivo.

Return: En caso de éxito, se devuelve 0. En caso de error, se devuelve -1 y *errno* indica el correspondiente error.

Sesion 2: Permisos y Directorios

umask

Resumen: Establece la máscara de creación de ficheros.

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

mode_t umask(mode_t mask);
```

La máscara de usuario es utilizada por **open(2)** o **mkdir(2)** para establecer los permisos iniciales del archivo o directorio que se va a crear. Específicamente, los permisos iniciales presentes en la máscara se desactivan del argumento *mode* de **open** de la siguiente forma:
`mode & ~umask`

Los posibles permisos se encuentran con la orden **open**.

Ejemplo:

mode = 0666 y *mask*=022. Entonces:

```
umask(mask); // Crea una máscara para la creación de archivos.

// El archivo se crea con los permisos 0666 & ~022 = 0644 = rw-r--r--
open("archivo", O_WRONLY | O_CREAT, mode);
```

Return: Siempre tiene éxito y devuelve el valor previo de *mask*.

chmod y fchmod

Resumen: Cambia los permisos dde un archivo.

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

int chmod(const char *path, mode_t mode);
int fchmod(int fildes, mode_t mode);
```

Cambia los permisos del archivo dado mediante *path* o referido por *fildes* (*fd*). Los permisos se pueden especificar mediante un or lógico de las variables constantes dadas en **open**.

Return: En caso de éxito devuelve 0. De haber un error se devuelve -1 y se asigna correspondientemente la variable *errno*.

opendir

Resumen: Abre un directorio.

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *name);
DIR *fdopendir(int fd);
```

Crea un directorio de estructura tipo *DIR*, llamada *stream* de directorio, correspondiente al *pathname* dado por *name*. Devuelve el puntero a la estructura creada. El *stream* está posicionado en la primera entrada del directorio. El tipo *DIR* se encuentra en ‘.

La función **fdopendir** es como **opendir** pero crea un *DIR* a partir del *fd* dado como argumento. EL *fd* es usadp internamente por la estructura y no debería ser modificado por el programa.

Return: En caso de éxito, se devuelve el puntero a DIR. En caso de error, NULL es devuelto como puntero en ambos casos.

readdir

Resumen: Lee de un directorio.

```
#include <dirent.h>

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

Devuelve un puntero a una *struct dirent* representando la siguiente entrada en el stream del directorio proporcionado por el puntero *dirp*. La lectura se realiza donde est situado el puntero de lectura del directorio dado. Dicho puntero pasa a la siguiente entrada tras el proceso. Devuelve NULL al final del stream del directorio o si tuvo lugar un error.

En Linux, la *struct dirent* está definida como sigue:

```
struct dirent {
    ino_t          d_ino;       /* número de i-nodo */
    off_t          d_off;       /* desplazamiento al siguiente dirent */
    unsigned short d_reclen;    /* longitud de esta entrada */
    unsigned char  d_type;      /* tipo de archivo */
    char           d_name[256]; /* nombre del archivo */
};
```

La estructura devuelta puede estar alojada estáticamente, luego no tiene sentido liberarla de la memoria.

El valor dad por *d_off* es equivalente al que proporciona **telldir**. Los posibles valores de d_type son:

- DT_BLK : This is a block device.
- DT_CHR : This is a character device.
- DT_DIR : This is a directory.
- DT_FIFO : This is a named pipe (FIFO).
- DT_LNK : This is a symbolic link.
- DT_REG : This is a regular file.
- DT_SOCK : This is a UNIX domain socket.
- DT_UNKNOWN : The file type is unknown.

Return: En caso de éxito, se el puntero a *dirent*. En caso de error, se devuelve -1 y *errno* indica el correspondiente error.

Sesion 3: Procesos Hijos

fork

Resumen: Crea un proceso hijo.

```
#include <unistd.h>

      pid_t fork(void);
```

Return: Devuelve: - 0 al hijo (que se ejecuta en segundo lugar lugar). - El PID del hijo al padre.

setbuf

Resumen: Cambia el buffer de un flujo abierto.

```
#include <stdio.h>

void setbuf(FILE *stream, char *buf);

void setbuffer(FILE *stream, char *buf, size_t size);

void setlinebuf(FILE *stream);

int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

La función setvbuf() puede ser utilizada en cualquier flujo abierto para cambiar su buffer. El argumento *mode* debe ser una de las tres siguientes macros:

- _IONBF unbuffered
- _IOLBF line buffered
- _IOFBF fully buffered

wait

Resumen: Obtiene el pid del último hijo finalizado o espera hasta que uno nuevo finalice.

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);

pid_t waitpid(pid_t pid, int *status, int options);

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

Estas llamadas al sistema son utilizadas para esperar al cambio de estado en el proceso de un hijo y obtener información sobre el mismo. Un cambio de estado puede estar dado por: el hijo finalizó su ejecución, el hijo se paró por determinada señal... En el caso de que el hijo haya finalizado un *wait* permite al sistema liberar los recursos asociados con el hijo. Si no se realizó un *wait*, el hijjo finalizado permanecerá como proceso "zombie".

Si un hijo ya cambió de estado, las llamadas devuelven el pid de dicho hijo inmediatamente. En caso contrario, se bloquea el proceso padre hasta que el hijo cambie de estado o sea interrumpido.

wait

En el caso de *wait*, el sistema suspende la ejecución del proceso hasta que uno de sus hijos termine. La llamada `wait(&status)` es equivalente a waitpid(-1, &status, 0).

waitpid

En el caso de *waitpid* el sistema suspende la ejecución del proceso hasta que el hijo de argumento especificado cambie de estado. Por defecto, *waitpid* solo espera a que se terminen los procesos, pero este hecho es modificable a través de los argumentos.

El valor del *pid* puede ser:

- < -1 : Se espera a cualquier hijo cuyo ID de grupo (gid) sea igual al valor absoluto de *pid.
- -1 : Espera a que cualquier hijo finalice.
- 0 : Espera a cualquier hijo cuyo gid sea igual al de su padre.
- > 0 : Espera a cualquier hijo cuyo ID sea igual a *pid*.

En el caso de *options* se pueden dar las siguientes opciones constantes:

- WNOHANG : Devuelve inmediatamente el control si ningún hijo ha salido.
- WUNTRACED : Devuelve el control si un hijo a parado su ejecución. Es la opción por defecto.
- WCONTINUED : Devuelve el control si un hijo ha sido parado por culpa de SIGCONT.

Si *status* no es NULL, *wait* y *waitpid* almacenan la información del cambio de estatus en el entero al que apunta *status*. Este entero puede ser especificado por las siguientes macros, que toman el argumento como un entero y no un puntero a el mismo:

- WIFEXITED(status) : Devuelve **true** si el hijo terminó normalmente (exit(2), exit(3) o desde el main).
- WEXITSTATUS(status) : Devuelve el estatus de salida del hijo. Utiliza los 8 bits menos significativos del argumento de extatus para especificar si terminó por exit(2), exit(3) o en el main. Solo se debe emplear si WIFEXITED devolvió **true**.
- WIFSIGNALED(status) : Devuelve **true** si el hijo fue terminado por una señal.
- WTERMSIG(status) : Devuelve el número de la señal que causó al hijo terminar. Solo se emplea si WIFSIGNALED devolvió **true**.
- WCOREDUMP(status) : Devuelve **true** si el hijo produció un "core dump". Se emplea solo si WIFSIGANLED devolvió **true**.
- WIFSTOPPED(status) : Devuelve **true** si el proceso fue parado por pedido de una señal. Solo es posible utilizarlo si WUNTRACED fue usado o el hijo está trazado.
- WSTOPSIG(status) : Devuelve el número de la señal que causó parar al hijo. Solo puede ser usado si WIFSTOPPED devolvió **true**.
- WIFCONTINUED(status) : Devuelve **true** si el proceso fue "resumed" por pedido de SIGCONT.

Return: En caso de éxito, tanto **wait** como **waitpid** devuelven el PID del hijo finalizado. En caso de error se devuelve -1.

exec

Resumen: La familia de funciones exec() reemplaza la imagen del proceso actual con una imagen de un nuevo proceso.

```c
#include <unistd.h>

extern char **environ;
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ... , char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

El primer argumento es el nombre del archivo a ser ejecutado.

Por otro lado, el *const char* arg *y los subsequentes argumentos en* ***execl****,* ***execlp*** *y* ***execle*** *son interpretados como* arg0*,* arg1*, ... argumentos del programa a llamar dado por el primer argumento. Esta lista de argumentos debe finalizar mediante un puntero NULO.

En el caso de **execv**, **execvp** y **execvpe** se proporciona un array de cadenas de caracteres con los correspondientes argumentos y una última componente NULL indicando el final del array.

En aquellos **execs** con la letra p, el primer argumento es la dirección del archivo a ejecutar. El nombre del mismo se presupone dado como argumento. La dirección nula implica por defecto que el archivo está en el directorio actual.

Las funciones **execle** y **execvpe** permiten al usuario especificar el contexto en el que se ejecutará el programa vía el argumento *envp*. Dicho argumento es un array de punteros a cadenas con terminación en NULL. Las demás funciones toman como variable para el contexto aquel en el que se llama al proceso.

Return: Las funciones **exec** devuelven -1 si hay un error. En caso contrario, devuelven un valor distinto de -1.

Sesión 4: Archivos Especiales, Cauces y Dup

mknod

Resumen: Permite crear archivos especiales.

```c
# include <sys/types.h>
# include <sys/stat.h>
# include <fcntl.h>
# include <unistd.h>

int mknod(const char *pathname, mode_t mode, dev_t dev);
```

La llamada al sistema **mknod()** crea un nodo en el sistema de archivos (archivo, archivo especial de dispositivo, cauce con nombre) llamado *pathname*, cuyos atributos son especificados por *mode* y *dev*.

El argumento modo especifica los permisos de uso y el tipo de nodo a ser creado. Debería ser una combinación de (utilizando la operación OR) de uno de los tipos de archivos listado debajo y los permisos del nuevo nodo.

Los permisos son modificados por la máscara del proceso de forma usual, siendo los permisos del nodo creado: (mode & ~umask).

El tipo de archivo debe ser uno de los siguientes: S_IFREG, S_IFCHR, S_IFBLK, S_IFIFO or S_IFSOCK. Estos tipos generan: un archivo regular (creado vacío), un archivo especial de dispositivo orientado a caracteres, archivo especial de dispositivo orientado a bloques, FIFO (llamado pipe o cauce) o un "UNIX domain socket", respectivamente. El valor por defecto del tipo será S_IFREG.

Si el tipo de archivo es S_IFCHR o S_IFBLK entonces *dev* especifica a qué dispositivo se refiere el archivo especial. Para crear un cauce FIFO el valor de este elemento será 0. Un ejemplo de creación de cauce FIFO es el siguiente:

```
mknod("/tmp/FIFO", S_IFIFO|0666, 0);
```

Si *pathname* ya existe o es un enlace simbólico, la llamada falla generando un error.

Return: Devuelve 0 en caso de éxito y -1 si ha habido algún error.

mkfifo

Resumen: Crea un archivo FIFO.

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

int mkfifo(const char *pathname, mode_t mode);
```

La llamada **mknod** permite crear cualquier tipo de archivo especial. Sin embargo, en el caso particular de los archivos FIFO **mkfifo** es una llamada al sistema específica.

mkfifo crea un archivo FIFO de nombre *pathname*. El argumento *mode* especifica los permisos del FIFO. Es modificado por la máscara del proceso de la manera usual: (mode & ~umask)

Return: Devuelve 0 en caso de éxito y -1 si ha habido algún error.

unlink

Resumen: Elimina un nombre y el posible archivo al que se refiera.

```
#include <unistd.h>

int unlink(const char *pathname);
```

Elimina un nombre del sistema de archivos. Si el nombre está enlazado a un archivo (no utilizado en dicho momento por ningún proceso) dicho archivo es eliminado. Si el archivo está siendo utilizado por algún proceso, permanecerá en el sistema hasta que finalice el último proceso que lo utiliza.

Si el nombre es un *enlace simbólico* se remueve.

Si el nombre se refiere a un socket, fifo o dispositivo, el nombre es eliminado pero puede seguir siendo utilizado por los procesos que lo tienen abierto.

Return: Devuelve 0 en caso de éxito y -1 si ha habido algún error.

pipe

Resumen: Crea un cauce sin nombre.

```
#include <unistd.h>

int pipe(int pipefd[2]);

#define _GNU_SOURCE             /* See feature_test_macros(7) */
#include <fcntl.h>              /* Obtain O_* constant definitions */
#include <unistd.h>

int pipe2(int pipefd[2], int flags);
```

pipe crea un cauce, un camino de datos unidireccional que puede ser utilizado para la comunicación entre procesos. El array *pipefd* es usado para devolver dos descriptores de archivo (fd) que apuntan al final e inicio del cauce (es un FIFO). pipefd[1] apunta al final del cauce (escritura) y pipefd[0] al inicio del cauce (lectura). Los datos escritos en el cauce son mandados por el kernel al final de la zona de lectura del cauce.

En el caso de **pipe2** se disponen de las opciones dadas por *flags*. Si este es 0 se comporta como **pipe**. Los siguientes valores pueden ser usados con una operación OR lógica para el argumento *flags*:

- O_NONBLOCK : Pone el estatus de archivo O_NONBLOCK en los dos fd.
- O_CLOEXEC : Pone la bandera “close-on-exec” (FD_CLOEXEC) en los dos fd.

Return: En caso de éxito se devuelve 0. Si hay un error se devuelve -1.

dup

Resumen: Duplica un descriptor de archivo (fd).

```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);

#define _GNU_SOURCE             /* See feature_test_macros(7) */
#include <fcntl.h>              /* Obtain O_* constant definitions */
#include <unistd.h>

int dup3(int oldfd, int newfd, int flags);
```

Estas llamadas al sistema crean una copia del descriptor de archivo *oldfd*.

En el caso de **dup** se utiliza como nuevo descriptor de archivo el más pequeño no utilizado.

Por otro lado, **dup2** hace que *newfd* sea el fd en el que se copie *oldfd*. Nótese lo siguiente:

- Si *oldfd* no es un fd válido, entonces la llamada falla y *newfd* no es cerrado.
- Si *oldfd* es un fd válido y *newfd* tiene el mismo valor que *oldfd*, entonces **dup2** no hace nada y devuelve *newfd*.

Despúes de uuna llamada exitosa de algunas de estas llamadas al sistema, el antiguo y el nuevo desciptor de archivo pueden ser usados independientemente.

dup3 es lo mismo que **dup2** salvando las posibles banderas.

Return: En caso de éxito, estas llamadas al sistema devuelven el nuevo descriptor de archivo. En caso de error se devuelve -1.

Ejemplo:

```
// Asigna fd[1] a la salida estándar:
dup2(fd[1],STDOUT_FILENO);
```

Sesión 5: Señales

En esta sección se tratan las llamadas al sistema relacionadas con el control y gestión de señales. De forma previa se dan las siguientes definiciones:

- Una señal es *depositada* cuando el proceso inicia una acción en base a ella.
- Una señal está *pendiente* si ha sido generada pero todavía no depositada.
- Una señal está *bloqueada* si se encuentra en la denominada *máscara de bloqueo de señales* del proceso, en cuyo caso permanece pendiente hasta que es desbloqueada, deja de pertenecer a la máscara. Si una señal es recibida varias veces cuando estaba bloqueada se trata como una única señal al desbloquearse.
- Una señal se dice *ignorada* si el proceso no le presta atención cuando es generada.

Una señal se representa mediante un entero positivo. El símbolo de la misma empieza con SIG y sigue con letras características de su acción. Consultar las posibles señales en el **man** o en la consola con `kill -l`.

kill

Resumen: Envía una señal a un proceso.

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

La llamada al sistema **kill** puede ser usada para enviar una señal a cualquier proceso o grupo de procesos.

Argumentos:

- Si *pid* es positivo, entonces se envía la señal *sig* al proceso identificado por *pid*.
- Si *pid* es 0 entonces, se envía la señal a cada proceso del grupo de procesos del proceso actual.
- Si *pid* es igual a -1, entonces *sig* es enviado a cada proceso con permiso para mandarle una señal. En la práctica esto se traduce a enviar la señal a cada proceso en la tabla de procesos desde los números más altos hasta los más bajos (sin contar el proceso *init* cuyo ID es 1).
- Si *pid* es menor que -1, entones se envía *sig* a cualcuier proceso en el grupo de procesos -*pid*.
- Si *sig* es 0, entonces no se envía ninnguna señal, pero si se comprueban los posibles errores. Esto puede ser utilizado para comprobar la existencia de un proceso o grupo de procesos con el *pid* dado .

Return: En caso de éxito (al menos una señal fue enviada) se devuelve 0. Si tuvo lugar algún error se devuelve -1 y *errno* toma el valor correspondiente.

sigaction

Resumen: Examina y cambia la acción tomada por un proceso ante determinada señal.

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

La llamada al sistema **sigaction** es utilizada para cambiar la acción tomada por un proceso en la recepción de una señal específica.

Argumentos:

- *signum* especifica la señal y puede ser cualquier señal válida exceptuando SIGKILL y SIGSTOP.
- Si *act* no es NULL, la nueva acción a efectuar para la señal *signum* es instala como *act*.
- Si *oldact* no es NULL, la acción previa se guarda en *oldact*.

La estructura *sigaction* se define de la siguiente forma:

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
};
```

Los datos miembros de la misma se explican a continuación:

- El elemento *sa_restorer* está obsoleto y no debería utilizarse.
- *sa_handler* especifica la acción a ser asociada con la señal *signum* pudiendo ser:
  - SIG_DFL para a acción prefeterminada (DFL = default).
  - SIG_IGN para ignorar la señal (IGN = ignore).
  - Un puntero a una función manejadora de la señal.
- *sa_mask* especifica una máscara de señales que deberían ser bloqueadas durante la ejecución del manejador de la señal (estas señales se añaden a la máscara de señal de la hebra en el que el manejador de señal es invocado). Además, la señal que lance el manejador será bloqueada, a menos que se activen las opciones SA_NODEFER o SA_NOMASK.
  Para asignar valores a *sa_mask* se utilizan las siguientes funciones:

  ```
  int sigemptyset(sigset_t *set);
  ```

  Inicializa a vacío un conjunto de señales (devuelve 0 si tiene éxito y –1 en caso contrario).

```
int sigfillset(sigset_t *set);
```

Inicializa un conjunto con todas las señales (devuelve 0 si tiene éxito y –1 en caso contrario).

```
int sigismember(const sigset_t *set, int senyal);
```

Determina si una señal senyal pertenece a un conjunto de señales set (devuelve 1 si la señal se encuentra dentro del conjunto, y 0 en caso contrario).

```
int sigaddset(sigset_t *set, int signo);
```

Añade una señal a un conjunto de señales set previamente inicializado (devuelve 0 si tiene éxito y –1 en caso contrario).

```
int sigdelset(sigset_t *set, int signo);
```

Elimina una señal signo de un conjunto de señales set (devuelve 0 si tiene éxito y –1 en caso contrario).

- *sa_flags* especifica el conjunto de opciones que modifican el comportamiento de la señal. Es formado por el uso de la operación lógica OR de cero o más de las siguientes constantes:
  - SA_NOCLDSTOP:
    Si signum es SIGCHLD (en caso contrario no hace nada), indica al núcleo que el proceso no desea recibir notificación cuando los procesos hijos se paren (esto es, cuando los procesos hijos reciban una de las señales: SIGTSTP, SIGTTIN o SIGTTOU).
  - SA_ONESHOT o SA_RESETHAND:
    Indica al núcleo que restaure la acción para la señal al estado predeterminado una vez que el manejador de señal haya sido llamado.
  - SA_RESTART:
    Proporciona un comportamiento compatible con la semántica de señales de BSD haciendo que ciertas llamadas al sistema reinicien su ejecución cuando son interrumpidas por la recepción de una señal.
  - SA_NOMASK o SA_NODEFER:
    Se pide al núcleo que no impida la recepción de la señal desde el propio manejador de la señal.
  - SA_SIGINFO:
    El manejador de señal toma 3 argumentos, no uno. En este caso, se debe configurar sa_sigaction en lugar de sa_handler.

El argumento siginfo_t para sa_sigaction es una estructura con los siguientes elementos:

```
siginfo_t {
    int      si_signo;    /* Número de señal */
    int      si_errno;    /* Un valor errno */
    int      si_code;     /* Código de señal */
```

```
    int      si_trapno;   /* Número de la trampa que causó la señal */
    pid_t    si_pid;      /* ID del proceso emisor */
    uid_t    si_uid;      /* ID del usuario real del proceso emisor */
    int      si_status;   /* Valor de salida o señal */
    clock_t  si_utime;    /* Tiempo de usuario consumido */
    clock_t  si_stime;    /* Tiempo de sistema consumido */
    sigval_t si_value;    /* Valor de señal */
    int      si_int;      /* señal POSIX.1b */
    void    *si_ptr;      /* señal POSIX.1b */
    int      si_overrun;  /* Contador overrun: POSIX.1b timers */
    int      si_timerid;  /* Contador ID: POSIX.1b timers */
    void    *si_addr;     /* Dirección de memoria que causo el fallo */
    long     si_band;     /* Evento de conjunto */
    int      si_fd;       /* Descriptor de archivo */
    short    si_addr_lsb; /* Bits menos significativos de la dirección */
}
```

Return: En caso de éxito **sigaction** devuelve 0. Si tuvo lugar algún error se devuelve -1 y *errno* toma el valor correspondiente.

sigprocmask

Resumen: Examina y cambia la máscara de señales.

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

sigprocmask es usado para buscar y/o cambiar la máscara de señal de la hebra que lo ejecuta. La máscara de señal es un conjunto de señales cuyo procesamiento está bloqueado.

El comportamiento de la señal depende del valor de *how*:

- SIG_BLOCK:
  El conjunto de señales bloqueadas es la unión del conjunto actual y el argumento *set*.
- SIG_UNBLOCK:
  Las señales que hay en *set* se eliminan de la máscara. Es posible intentar la eliminación de una señal no bloqueada.
- SIG_SETMASK:
  Al conjunto de señales bloqueadas se le asigna el argumento *set*.

Si el *oldset* no es NULL, el previo valor de la máscara de señal es almacenado en *oldset*. En caso de ser NULL no interviene en nada.

Si *set* es NULL, entonces no se cambia la máscara de señal (*how* es ignorado), pero el valor actual de la máscara de señal es en cualquier caso almacenado en *oldset* (si no es NULL).

Return: En caso de éxito se devuelve 0. Si tuvo lugar algún error se devuelve -1 y *errno* toma el valor correspondiente.

sigpending

Resumen: Examina las señales pendientes.

```
#include <signal.h>
int sigpending(sigset_t *set);
```

sigpending devuelve el conjunto de señales que están pendientes de procesamiento en la hebra que utiliza la llamada al sistema (las señales que han surgido estando bloqueadas). Por tanto, la máscara de señal aplicada a las señales pendientes se almacena en *set*.

Return: En caso de éxito se devuelve 0. Si tuvo lugar algún error se devuelve -1 y *errno* toma el valor correspondiente.

sigsuspend

Resumen: Espera a una señal.

```
#include <signal.h>
int sigsuspend(const sigset_t *mask);
```

sigsuspend reemplaza temporalmente la máscara de señal del proceso que ejecuta la llamada por la máscara dada en *mask* y suspende el proceso hasta que se recibe una señal que invoca el manejador de señales o termina un proceso.

Si la señal termina el proceso **sigsuspend** no devuelve nada. Si la señal es captada, entonces **sigsuspend** sí ejecuta la devolución, tras la ejecución del manejador de señales, y la máscara de señal se restaura a los valores que tenía antes de **sigsuspend**.

No es posible bloquear SIGKILL o SIGSTOP, especificar estas señales en la máscara no tiene efecto alguno.

Return: **sigsuspend** siempre devuelve -1 con *errno* indicando el error correspondiente. (Habitualmente EINTR que indica que la llamada fue interrumpida por una señal).

Sesión 6: fcntl y Proyecciones de Memoria

fcntl

Resumen: Manipula un descriptor de archivo. Proviene de *file control*.

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd, ... /* arg */ );
```

fcntl realiza una de las operaciones descritas debajo en el descriptor de archivo *fd*. La operación es determinada por *cmd*. Puede tomar un tercer argumento adicional. Si es o no requerido este argumento es determinado por *cmd*. El tipo de argumento requerido es indicado en paréntesis despues de cada *cmd* en la siguiente explicación. Nótese que en la mayoría de los casos el argumento requerido es un *int* y que dicho argumento se identifica mediante el nombre *arg*. En caso de no requerir un argumento lo denotamos con (*void*):

Banderas de estado de un archivo abierto

Los siguientes comandos manipulan las banderas asociadas con un descriptor de archivo.

- F_GETFL (void):
  Devuelve las banderas de estado del archivo de *fd* dado.
- F_SETFD (int):
  Asigna las banderas del archivo de *fd* dado al valor especificado por *arg*.

- F_GETFD:
  Devuelve la bandera **close-on-exec** 10 del archivo indicado. Si devuelve un 0, la bandera está desactivada, en caso contrario devuelve un valor distinto de cero. La bandera close-on-exec de un archivo recién abierto esta desactivada por defecto.
- F_SETFD:
  Activa o desactiva la bandera **close-on-exec** del descriptor especificado. En este caso, el tercer argumento de la función es un valor entero que es 0 para limpiar la bandera, y 1 para activarlo.

Ejemplo:

```
int bandera;
bandera = fcntl(fd, F_GETFL);            // Se obtienen las banderas.
if (bandera == -1)
   perror("fcntl");
bandera |= O_APPEND;                     // Se añade O_APPEND
if (fcntl(fd, F_SETFL, bandera) == -1)   // Se asigna la nueva bandera
    perror("fcntl");
```

Duplicando un descriptor de archivo

- F_DUPFD (int):
  Busca el menor número disponible en la tabla de descriptores de archivo y le asigna *fd* (ver **dup** y **dup2**). En caso de éxito se devuelve el nuevo descriptor de archivo y en caso de error -1.
  Ejemplo:

```
// Rediccionamiento de la salida estándar:
int fd = open ("temporal", O_WRONLY);
close (1);
if (fcntl(fd, F_DUPFD, 1) == -1 ) perror ("Fallo en fcntl");
char bufer[256];
int cont = write (1, bufer, 256);
```

Bloqueo de archivos

- F_SETLK (struct flock *):
  Establece un cerrojo sobre un archivo. No bloquea si no tiene éxito inmediatamente.
- F_SETLKW (struct flock *):
  Establece un cerrojo y bloquea al proceso llamador hasta que se adquiere el cerrojo.
- F_GETLK (struct flock *):
  Consulta si existe un bloqueo sobre una región del archivo.

El tercer argumento, *lock*, es un puntero a una estructura que tiene al menos los siguientes campos:

```
struct flock {
    ...
    short l_type;    /* Type of lock: F_RDLCK,
                        F_WRLCK, F_UNLCK */
    short l_whence;  /* How to interpret l_start:
                        SEEK_SET, SEEK_CUR, SEEK_END */
    off_t l_start;   /* Starting offset for lock */
    off_t l_len;     /* Number of bytes to lock */
    pid_t l_pid;     /* PID of process blocking our lock
                        (F_GETLK only) */
    ...
};
```

mmap

Resumen: Hace un map (proyección) de un archivo u objeto en memoria compartida del espacio de direcciones del proceso.

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
int munmap(void *addr, size_t length);
```

mmap crea una nueva proyección en memoria virtual del proceso correspondiente.

Argumentos:

La dirección de inicio para proyectar (o mapear) el descriptor es proporcionada por *addr*. Usualmente se proporciona un NULL que indica que el kernel decide la dirección de inicio.

El argumento *length* indica el número de bytes a proyectar, empezando con un desplazamiento desde el inicio del archivo dado por *offset*, que normalmente es 0.

El argumento *fd* indica el descriptor de archivo que se debe proyectar.

Las proyecciones se realizan mediante páginas pues son la unidad mínima de gestión de memoria. Por tanto, los argumentos deberían estar alineados correspondientemente. En caso de no adaptarse al tamaño de página, la proyección se redondea por exceso hasta completar la página. Los bytes añadidos en tal caso toman el valor 0.

El tipo de proyección viene dado por la máscara de bits *prot* que puede estar formado por una OR lógica de las siguientes constantes:

- PROT_READ: Los datos se pueden leer.
- PROT_WRITE: Los datos se pueden escribir.
- PROT_EXEC: Podemos ejecutar los datos.
- PROT_NONE: No podemos acceder a los datos.

El argumento *flags* contiene uno de los siguientes valores:

- MAP_SHARED: Los cambios son compartidos.
- MAP_PRIVATE: Los cambios son privados.
- MAP_FIXED: Interpreta exactamente el argumento addres.
- MAP_ANONYMOUS: Crea un mapeo anónimo (Apartado 3.2).
- MAP_LOCKED: Bloquea las páginas en memoria (al estilo mlock).
- MAP_NORESERVE: Controla la reserva de espacio de intercambio.
- MAP_POPULATE: Realiza una lectura adelantada del contenido del archivo.
- MAP_UNITIALIZED: No limpia (poner a cero) las proyecciones anónimas.

Return: En caso de éxito se devuelve la dirección de inicio adignada. Si tuvo lugar algún error se devuelve -1 y *errno* toma el valor correspondiente.

memcpy

Resumen: Copia una área de memoria.

```
#include <string.h>
void *memcpy(void *dest, const void *src, size_t n);
```

Copia *n* Bytes de la área de memoria dada por *src* al área de memoria *dest*. Las áreas no deben solaparse.

Return: Devuelve un puntero a *dest*.

ftruncate

Resumen: Trunca un archivo a determinada longitud.

```
#include <unistd.h>
#include <sys/types.h>
int truncate(const char *path, off_t length);
int ftruncate(int fd, off_t length);
```

Las funcionas **truncate** y **ftruncate** causan al archivo regular dado por *path* o referenciado por *fd* ser truncado a determinada longitud en *Bytes*.

Si el archivo era previamente más largo que este nuevo tamaño, los datos extras se pierden. Si por el contrario era más corto, se extiende y la parte extendida toma el valor '\0' en cada Byte.

El *offset* del archivo no varía.

Con **ftruncate**, el archivo debe estar abierto para escritura, con **truncate** debe tener permisos de escritura.

Return: En caso de éxito se devuelve la dirección de inicio adignada. Si tuvo lugar algún error se devuelve -1 y *errno* toma el valor correspondiente.

Sesión 7: Cliente-Servidor

getpid

Resumen: Obtiene el pid de del proceso que ha efectuado la llamada.

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
```

Return: Siempre tiene éxito.

tmpfile

Resumen: Crea un archivo temporal.

```
#include <stdio.h>
FILE *tmpfile(void);
```

La función **tmpfile** crea un archivo temporal de lectura y escritura para datos en binario que se eliminará de forma automática cuando termine el proceso.

Return: Devuelve un descriptor de stream o NULL si no se puede generar el archivo, en cuyo caso errno se asigna adecuadamente.

fread, fwrite

Resumen: Lectura y escritura en binario de un stream.

```
#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

La función **fread** lee *nmemb* elementos de tamaño *size* Bytes del stream dado por *stream*, almacenándolos en la dirección dada por *ptr*.

La función **fwrite** escribe *nmemb* elementos de tamaño *size* Bytes en el stream dado por *stream*, obteniéndolos de la dirección dada por *ptr*.

Return: En caso de éxito, **fread** y **fwrite** devuelven el número de elementos leídos o escritos. Si un error tiene lugar o se ha alcanzado el final del archivo se devuelve los bytes leídos o escritos hasta el momento del mismo.
