Contenido

Resumen Modulo 2

Jose Antonio Ruiz Millan

INCLUDES NECESARIOS:

```
#include <fcntl.h>        //open
#include <sys/types.h> //open,lseek,stat,umask,chmod,directorios,procesos,fifo
#include <sys/stat.h>    //open,stat,umask,chmod,fifo
#include <unistd.h>      //close,write,read,lseek,stat,procesos,exec,pipe,dup2
#include <dirent.h>      //directories
#include <ftw.h>                  //nftw
#include <stdio.h>       //setvbuf,printf
#include <sys/wait.h>    //wait,waitpid
#include <stdlib.h>      //exit,strtol,atoi
#include <signal.h>      //señales
#include <fcntl.h>       //fcntl
#include <errno.h>
#include <string.h>      //strcmp,str…,memset
#include <stdbool.h>     //Cabeceras para poder usar el tipo booleano
```

Sistemas de ficheros:

Open:

```
int open(const char *path, int oflags);
int open(const char *path, int oflags, mode_t mode);
```

const char *path → Ruta relativa o absoluta al archivo que se va a abrir.

int oflags → Una lista de valores que determina el método en el que se va a abrir el archivo (si debe ser de sólo lectura, lectura / escritura, si se debe borrar cuando se abre, etc.).

POSIBLES VALORES:

O_RDONLY	Abre el fichero para solo lectura
O_WRONLY	Abre el fichero para solo escritura
O_RDWR	Abre el fichero para lectura y escritura
O_APPEND	Añade nueva informacion al final del fichero
O_TRUNC	Borra el contenido al abrir el fichero
O_CREAT	Si el archivo no existe, lo crea. Si se utiliza la opción O_CREAT, debe incluir el tercer parámetro.
O_EXCL	Combinado con la opción O_CREAT, garantiza que el llamador debe crear el archivo. Si el archivo ya existe, la llamada fallará.

mode_t mode → Una lista de valores separados por "|" que determinan los permisos del archivo si se crea.

CUIDADO CON LA MASCARA!!! Usar umask(0) antes para eliminarla.

S_IRUSR	Pone a verdadero lectura para propietario.
S_IWUSR	Pone a verdadero escritura para propietario.
S_IXUSR	Pone a verdadero ejecucion para propietario.
S_IRGRP	Pone a verdadero lectura para grupo.
S_IWGRP	Pone a verdadero escritura para grupo.
S_IXGRP	Pone a verdadero ejecucion para grupo.
S_IROTH	Pone a verdadero lectura para otros.
S_IWOTH	Pone a verdadero escritura para otros.
S_IXOTH	Pone a verdadero ejecucion para otros.

return value → Devuelve el descriptor al archivo que es un numero entero >= que 0.
Devuelve un numero negativo en caso de error.

Close:

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

int fildes	Descriptor del archivo a cerrar.
return value	Devuelve 0 en caso correcto y -1 en caso de fallo

Write:

```
size_t write(int fildes, const void *buf, size_t nbytes);
```

int fildes	El descriptor de archivo de dónde escribir la salida. Puede utilizar un descriptor de archivo obtenido de la llamada de sistema abierta o puede utilizar 0, 1 o 2 para referirse a la entrada estándar, la salida estándar o el error estándar, respectivamente.
const void *buf	Una cadena de caracteres terminada null del contenido a escribir.
size_t nbytes	El número de bytes a escribir. Si es menor que el búfer proporcionado, la salida se trunca.
return value	Devuelve el número de bytes que se escribieron. Si el valor es negativo, la llamada al sistema devolvió un error.

Read:

```
size_t read(int fildes, void *buf, size_t nbytes);
```

int fildes	El descriptor de archivo de dónde leer la entrada. Puede utilizar un descriptor de archivo obtenido de la llamada de sistema abierta o puede utilizar 0, 1 o 2 para referirse a la entrada estándar, la salida estándar o el error estándar, respectivamente.
const void *buf	Un array de caracteres donde se almacenará el contenido de lectura.
size_t nbytes	El número de bytes a leer. Si los datos a leer son menores que el buffer, se guardan todos.
return value	Devuelve el número de bytes que se leyeron. Si el valor es negativo, la llamada al sistema devolvió un error.

Lseek:

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

int fildes → El descriptor de archivo del puntero que se va a mover.

off_t offset → El desplazamiento del puntero (medido en bytes).

int whence → El método en el que se debe interpretar el desplazamiento (relativo, absoluto, etc.).

SEEK_SET	El desplazamiento se realiza desde el principio del fichero
SEEK_CUR	El desplazamiento debe medirse con respecto a la posición actual del puntero.
SEEK_END	El desplazamiento debe medirse con respecto al final del archivo.

return value → Devuelve el desplazamiento del puntero (en bytes) desde el principio del archivo. Si el valor devuelto es -1, se produjo un error al mover el puntero.

Stat:

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

const char *path	El descriptor de archivo del archivo que se está consultando.
struct stat *buf	**Estructura donde se almacenarán los datos del archivo.**
return value	Devuelve un numero negativo si falla

Estructura Stat:

```
struct stat {
      dev_t st_dev; /* nº de dispositivo (filesystem) */
      dev_t st_rdev; /* nº de dispositivo para archivos especiales */
      ino_t st_ino; /* nº de inodo */
      mode_t st_mode; /* tipo de archivo y mode (permisos) */
      nlink_t st_nlink; /* número de enlaces duros (hard) */
      uid_t st_uid; /* UID del usuario propietario (owner) */
      gid_t st_gid; /* GID del usuario propietario (owner) */
      off_t st_size; /* tamaño total en bytes para archivos regulares
      */
      unsigned long st_blksize; /* tamaño bloque E/S para el sistema
      de archivos*/
      unsigned long st_blocks; /* número de bloques asignados */
      time_t st_atime; /* hora último acceso */
      time_t st_mtime; /* hora última modificación */
      time_t st_ctime; /* hora último cambio */
};
```

Comprobar el tipo de fichero con st_mode:

```
S_ISLNK(st_mode) Verdadero si es un enlace simbólico (soft)
S_ISREG(st_mode) Verdadero si es un archivo regular
S_ISDIR(st_mode) Verdadero si es un directorio
S_ISCHR(st_mode) Verdadero si es un dispositivo de caracteres
S_ISBLK(st_mode) Verdadero si es un dispositivo de bloques
S_ISFIFO(st_mode) Verdadero si es una cauce con nombre (FIFO)
```

S_ISSOCK(st_mode) Verdadero si es un socket

EJEMPLO : if(S_ISREG(atributos.st_mode)){};

Flags para trabajar con ST_MODE:

S_IFMT	0170000	máscara de bits para los campos de bit del tipo de archivo (no POSIX)
S_IFSOCK	0140000	socket (no POSIX)
S_IFLNK	0120000	enlace simbólico (no POSIX)
S_IFREG	0100000	archivo regular (no POSIX)
S_IFBLK	0060000	dispositivo de bloques (no POSIX)
S_IFDIR	0040000	directorio (no POSIX)
S_IFCHR	0020000	dispositivo de caracteres (no POSIX)
S_IFIFO	0010000	cauce con nombre (FIFO) (no POSIX)
S_ISUID	0004000	bit SUID
S_ISGID	0002000	bit SGID
S_ISVTX	0001000	sticky bit (no POSIX)
S_IRWXU	0000700	**user** (propietario del archivo) tiene permisos de lectura, escritura y ejecución
S_IRUSR	0000400	user tiene permiso de lectura (igual que S_IREAD, no POSIX)
S_IWUSR	0000200	user tiene permiso de escritura (igual que S_IWRITE, no POSIX)
S_IXUSR	0000100	user tiene permiso de ejecución (igual que S_IEXEC, no POSIX)
S_IRWXG	0000070	**group** tiene permisos de lectura, escritura y ejecución
S_IRGRP	0000040	group tiene permiso de lectura
S_IWGRP	0000020	group tiene permiso de escritura
S_IXGRP	0000010	group tiene permiso de ejecución
S_IRWXO	0000007	**other** tienen permisos de lectura, escritura y ejecución
S_IROTH	0000004	other tienen permiso de lectura
S_IWOTH	0000002	other tienen permiso de escritura
S_IXOTH	0000001	other tienen permiso de ejecución

Llamadas al sistema relacionadas con los permisos de archivos:

Umask:

```
mode_t umask(mode_t mask);
```

umask establece la máscara de usuario a mask & 0777.

Return value → Esta llamada al sistema siempre tiene éxito y devuelve el valor anterior de la máscara.

Chmod y fchmod:

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. Los permisos se pueden especificar mediante un OR lógico de los siguientes valores:

S_ISUID	04000	activar la asignación del UID del propietario al UID efectivo del proceso que ejecute el archivo.
S_ISGID	02000	activar la asignación del GID del propietario al GID efectivo del proceso que ejecute el archivo.
S_ISVTX	01000	activar *sticky* bit. En directorios significa un borrado restringido, es decir, un proceso no privilegiado no puede borrar o renombrar archivos del directorio salvo que tenga permiso de escritura y sea propietario. Por ejemplo se utiliza en el directorio **/tmp**.
S_IRWXU	00700	**user** (propietario del archivo) tiene permisos de lectura, escritura y ejecución
S_IRUSR	00400	lectura para el propietario (= S_IREAD no POSIX)
S_IWUSR	00200	escritura para el propietario (= S_IWRITE no POSIX)
S_IXUSR	00100	ejecución/búsqueda para el propietario (=S_IEXEC no POSIX)
S_IRWXG	00070	group tiene permisos de lectura, escritura y ejecución
S_IRGRP	00040	lectura para el grupo
S_IWGRP	00020	escritura para el grupo
S_IXGRP	00010	ejecución/búsqueda para el grupo
S_IRWXO	00007	other tienen permisos de lectura, escritura y ejecución
S_IROTH	00004	lectura para otros
S_IWOTH	00002	escritura para otros
S_IXOTH	00001	ejecución/búsqueda para otros

Return value → En caso de éxito, devuelve 0. En caso de error, -1 y se asigna a la variable errno un valor adecuado.

Manejo de directorios:

Opendir:

DIR *opendir(const char *name);

Return value → puntero a la estructura de tipo DIR. En caso de error devuelve NULL.

Readdir:

struct dirent *readdir(DIR *dirp);

lee la entrada donde esté situado el puntero de lectura de un directorio ya abierto cuyo stream se pasa a la función. Después de la lectura adelanta el puntero una posición.

return value → Devuelve la entrada leída a través de un puntero a una estructura (struct dirent), o devuelve NULL si llega al final del directorio o se produce un error.

Closedir:

int closedir(DIR *dirp);

return value → devolviendo 0 si tiene éxito, en caso contrario devuelve -1.

Seekdir:

void seekdir(DIR *dirp, long offset);

permite situar el puntero de lectura de un directorio (se tiene que usar en combinación con telldir).

Telldir:

long telldir(DIR *dirp);

devuelve la posición del puntero de lectura de un directorio. O -1 en caso de error.

Rewinddir:

void rewinddir(DIR *dirp);

posiciona el puntero de lectura al principio del directorio.

Estructuras:

```
DIR *opendir(char *dirname)
struct dirent *readdir(DIR *dirp)
int closedir(DIR *dirp)
void seekdir(DIR *dirp, log loc)
long telldir(DIR *dirp)
void rewinddir(DIR *dirp)
typedef struct _dirdesc {
            int dd_fd;
            long dd_loc;
            long dd_size;
            long dd_bbase;
            long dd_entno;
            long dd_bsize;
            char *dd_buf;
} DIR;
//La estructura struct dirent conforme a POSIX 2.1 es la siguiente:
#include <sys/types.h>
#include <dirent.h>
struct dirent {
            long d_ino; /* número i-nodo */
            char d_name[256]; /* nombre del archivo */
};
```

Nftw:

int nftw(const char *dirpath, int (*fn) (const char *fpath, const struct stat *sb, int typeflag, struct FTW *ftwbuf), int nopenfd, int flags);

La función recorre el árbol de directorios especificado por dirpath y llama a la función func definida por el programador para cada archivo del árbol. Por defecto, ntfw realiza un recorrido no ordenado en preorden del árbol, procesando primero cada directorio antes de procesar los archivos y subdirectorios dentro del directorio.

El argumento **flags** es creado mediante un OR (|) con cero o más constantes, que modifican la operación de la función:

FTW_DIR	Realiza un **chdir** (cambia de directorio) en cada directorio antes de procesar su contenido. Se utiliza cuando **func** debe realizar algún trabajo en el directorio en el que el archivo especificado por su argumento **pathname** reside.
FTW_DEPTH	Realiza un recorrido postorden del árbol. Esto significa que **nftw** llama a **func** sobre todos los archivos (y subdirectorios) dentro del directorio antes de ejecutar **func** sobre el propio directorio.
FTW_MOUNT	No cruza un punto de montaje.
FTW_PHYS	Indica a **nftw** que nos desreferencie los enlaces simbólicos. En su lugar, un enlace simbólico se pasa a **func** como un valor **typedflag** de FTW_SL.

El tercer argumento, **typeflag**, suministra información adicional sobre el archivo, y tiene uno de los siguientes nombres simbólicos:

FTW_D	Es un directorio
FTW_DNR	Es un directorio que no puede leerse (no se lee sus descendientes).
FTW_DP	Estamos haciendo un recorrido posorden de un directorio, y el ítem actual es un directorio cuyos archivos y subdirectorios han sido recorridos.
FTW_F	Es un archivo de cualquier tipo diferente de un directorio o enlace simbólico.
FTW_NS	**`stat`** ha fallado sobre este archivo, probablemente debido a restricciones de permisos. El valor **`statbuf`** es indefinido.
FTW_SL	Es un enlace simbólico. Este valor se retorno solo si **`nftw`** se invoca con FTW_PHYS
FTW_SLN	El ítem es un enlace simbólico perdido. Este se da cuando no se especifica FTW_PHYS.

El cuarto elemento de func **es ftwbuf**, es decir, un puntero a una estructura que se define de la forma:

```
struct FTW {
       int base;/* Desplazamiento de la parte base del pathname */
       int level;/*Profundidad del archivo dentro recorrido del arbol */
};
```

El campo **base** es un desplazamiento entero de la parte del nombre del archivo (el componente después del último /) del pathname pasado a func.

Return value → Esta llamada al sistema devuelve 0 si recorre completamente el árbol; -1 si error o el primer valor no cero devuelto por func.

EJEMPLO:

```
int visitar(const char* path, const struct stat* stat, int flags, struct FTW* ftw) {

        if ((S_ISREG(stat->st_mode)) && (stat->st_mode & S_IXGRP) && (stat->st_mode &
S_IXOTH)) {

            printf("%s %llu\n",path, stat->st_ino);

            size += stat->st_size;
            n_files++;

        }

        return 0;
}
```

Main:

```
if (nftw(argc >= 2 ? argv[1] : ".", visitar, 10, 0) != 0) {
      perror("nftw");
  }
```

Llamadas al sistema para procesos:

Llamadas a procesos:

```
pid_t getpid(void); // devuelve el PID del proceso que la invoca.

pid_t getppid(void); // devuelve el PID del proceso padre del proceso
que // la invoca.

uid_t getuid(void); // devuelve el identificador de usuario real del
// proceso que la invoca.

uid_t geteuid(void); // devuelve el identificador de usuario efectivo
del // proceso que la invoca.

gid_t getgid(void); // devuelve el identificador de grupo real del
proceso // que la invoca.

gid_t getegid(void); // devuelve el identificador de grupo efectivo
del // proceso que la invoca.
```

Fork:

```
pid_t fork(void);
```

Crea un proceso hijo.

Return value → devuelve 0 al proceso hijo. Ambos procesos continuarán ejecutándose desde la función fork (). De lo contrario, -1 se devolverá al proceso padre, no se creará ningún proceso hijo, y se establecerá errno para indicar el error.

Wait, waitpid :

pid_t wait(int *stat_loc);

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

EJEMPLO:

```
if( waitpid( childpid[i] , &estado , 0 ) > 0 ){
    printf("Acaba de finalizar mi hijo con PID:%d\n", childpid[i] );
        printf("Solo me quedan %d hijos vivos\n", nprocs - (i+1) );
  }
```

Exec:

```
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[]);
```

return value → Devuelve -1 si se produce algun error

Comunicación entre procesos con cauces:

Con nombre:

Mkfifo:

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

Crea un fichero fifo para la comunicación

TENER ENCUENTA UMASK. (umask(0)).

Return value → En caso de éxito mkfifo () devuelve 0. En el caso de un error, se devuelve -1 (en cuyo caso, errno se establece correctamente).

Utilizacion:

Creamos el fifo con mkfifo, luego lo abrimos con open y usamos write y read para hacer lo que necesitemos. Finalmente lo cerramos conclose y listo.

Sin nombre:

Pipe:

int pipe(int fildes[2]);

Crea un pipe pasandole un array de 2 enteros. Donde fd[0] es la lectura y fd[1] es la escritura.

NO OLVIDAR CERRAR EL DESCRIPTOR QUE NO VAYAMOS A UTILIZAR

Return value → Una vez completada con éxito, se devolverá 0; De lo contrario, se devolverá -1 y se establecerá errno para indicar el error.

Dup2:

int dup2(int fildes, int fildes2);

El primer parametro se duplica en el segundo, cerrandose anteriormente el Segundo.

Return value → Una vez completada con éxito, se devolverá un entero no negativo, es decir, el descriptor de archivo; De lo contrario, se devolverá -1 y se establecerá errno para indicar el error.

Gestion y control de señales:

Para saber las señales ejecutar **kill -l**

O ejecutar man 7 signal

Kill:

int kill(pid_t pid, int sig);

Envia una señal al proceso pid.

- Si pid es positivo, entonces se envía la señal sig al proceso con identificador de proceso igual a pid. En este caso, se devuelve 0 si hay éxito, o un valor negativo si se produce un error.
- Si pid es 0, entonces sig se envía a cada proceso en el grupo de procesos del proceso actual.
- Si pid es igual a -1, entonces se envía la señal sig a cada proceso, excepto al primero, desde los números más altos en la tabla de procesos hasta los más bajos.
- Si pid es menor que -1, entonces se envía sig a cada proceso en el grupo de procesos -pid.
- Si sig es 0, entonces no se envía ninguna señal, pero sí se realiza la comprobación de errores.

Return value → Una vez completada con éxito, se devolverá 0. De lo contrario, se devolverá -1 y se establecerá errno para indicar el error.

Sigaction:

int sigaction(int sig, const struct sigaction *restrict act, struct sigaction *restrict oact);

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

Return value → 0 en caso de éxito y -1 en caso de error

Estructura sigaction:

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

- **sa_handler** especifica la acción que se va a asociar con la señal signum pudiendo ser:
  - SIG_DFL para la acción predeterminada,
  - SIG_IGN para ignorar la señal
  - o un puntero a una función manejadora para la señal.

- **sa_mask** permite establecer una máscara de señales que deberían bloquearse durante la ejecución del manejador de la señal. 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 usan 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 un conjunto de opciones que modifican el comportamiento del proceso de manejo de señales. Se forma por la aplicación del operador de bits OR a cero o más de las siguientes constantes:
  - ◦SA_NOCLDSTOP

    Si signum es SIGCHLD, 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 **parámetro 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 */
            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 */
            void * si_addr; /* Dirección de memoria que ha producido el
            fallo */
            int si_band; /* Evento de conjunto */
            int si_fd; /* Descriptor de fichero */
}
```

EJEMPLO:

```
struct sigaction sa;
sa.sa_handler = SIG_IGN; // ignora la señal
sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
```

sigprocmask:

int sigprocmask(int how, const sigset_t *restrict set, sigset_t *restrict oset);

La llamada sigprocmask se emplea para examinar y cambiar la máscara de señales.

- El argumento how indica el tipo de cambio. Los valores que puede tomar son los siguientes:
  - 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 del conjunto actual de señales bloqueadas. Es posible intentar el desbloqueo de una señal que no está bloqueada.
  - SIG_SETMASK: El conjunto de señales bloqueadas se pone según el argumento set.
- set representa el puntero al nuevo conjunto de señales enmascaradas. Si set es diferente de NULL, apunta a un conjunto de señales, en caso contrario sigprocmask se utiliza para consulta.
- oldset representa el conjunto anterior de señales enmascaradas. Si oldset no es NULL, el valor anterior de la máscara de señal se guarda en oldset. En caso contrario no se retorna la máscara la anterior.

Return value → 0 en caso de éxito y -1 en caso de error

Sigpending:

int sigpending(sigset_t *set);

La llamada sigpending permite examinar el conjunto de señales bloqueadas y/o pendientes de entrega. La máscara de señal de las señales pendientes se guarda en set.

- set representa un puntero al conjunto de señales pendientes

return value → 0 en caso de éxito y -1 en caso de error

sigsuspend:

int sigsuspend(const sigset_t *mask);

La llamada sigsuspend reemplaza temporalmente la máscara de señal para el proceso con la dada por el argumento mask y luego suspende el proceso hasta que se recibe una señal.

- mask representa el puntero al nuevo conjunto de señales enmascaradas

return value → -1 si sigsuspend es interrumpida por una señal capturada (no está definida la terminación correcta)

EJEMPLO:

```
sigset_t new_mask;
/* inicializar la nueva mascara de señales */
sigemptyset(&new_mask);
sigaddset(&new_mask, SIGUSR1);
/*esperar a cualquier señal excepto SIGUSR1 */
sigsuspend(&new_mask);
```

Control de archivos (cerrosjos, etc):

Fcntl:

int fcntl(int fildes, int cmd, ...);

permite consultar o ajustar las banderas de control de acceso de un descriptor, es decir, de un archivo abierto. Además, permite realizar la duplicación de descriptores de archivos y bloqueo de un archivo para acceso exclusivo.

Argumento orden:

F_GETFL	Retorna las banderas de control de acceso asociadas al descriptor de archivo.
F_SETFL	Ajusta o limpia las banderas de acceso que se especifican como tercer argumento.
F_GETFD	Devuelve la bandera close-on-exec10 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.
F_DUPFD	Duplica el descriptor de archivo especificado por fd en otro descriptor. El tercer argumento es un valor entero que especifica que el descriptor duplicado debe ser mayor o igual que dicho valor entero. En este caso, el valor devuelto por la llamada es el descriptor de archivo duplicado (nuevoFD = fcntl(viejoFD, F_DUPFD, inicialFD).
F_SETLK	Establece un cerrojo sobre un archivo. No bloquea si no tiene éxito inmediatamente.
F_SETLKW	Establece un cerrojo y bloquea al proceso llamador hasta que se adquiere el cerrojo.
F_GETLK	Consulta si existe un bloqueo sobre una región del archivo.

EJEMPLO PERMISOS:

MododAcceso=banderas & O_ACCMODE;

```
if (MododAcceso == O_WRONLY || MododAcceso == O_RDWR)
        printf (”El archivo permite la escritura \n);
```

EJEMPLO DE DUPLICAR DESCRIPTOR:

```
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 fcntl:

Si bien a coste de eficiencia, deshabilitar el búfering de stdio con setbuf() o similar. O utilizar write y read.

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

```
struct flock {
      short l_type; /* Tipo de cerrojo: F_RDLCK, F_WRLCK, F_UNLCK */
      short l_whence;/*Interpretarl_start:SEEK_SET,SEEK_CURR,SEEK_END*/
      off_t l_start; /* Desplazamiento donde se inicia el bloqueo */
      off_t l_len; /* Numero bytes bloqueados: 0 significa “hasta EOF”
      */
      pid_t  l_pid;  /*  Proceso  que  previene  nuestro  bloqueo(solo
      F_GETLK)*/
};
```

El elemento **l_type** indica el tipo de bloqueo que queremos utilizar y puede tomar uno de los siguientes valores: F**_RDLCK** para un cerrojo de lectura**, F_WRLCK** para un cerrojo de escritura, **y F_UNLCK** que elimina un cerrojo.

EJEMPLO:

```
struct flock mi_bloqueo;

. .  /* ajustar campos de mi_bloqueo para describir el cerrojo a usar*/
```

fcntl(fd, orden, &mi_bloqueo);

El conjunto de campos **l_whence, l_start y l_len** especifica el rango de bytes que deseamos bloquear. Los primeros dos valores son similares a los campos whence y offset de lseek(), es decir, l_start especifica un desplazamiento dentro del archivo relativo a valor l_whence (**SEEK_SET** para el inicio del archivo, **SEEK_CUR** para la posición actual y **SEEK_END** para el final del archivo). El valor de **l_start** puede ser negativo siempre que la posición definida no caiga por delante del inicio del archivo.

El campo **l_len** es un entero que especifica el número de bytes a bloquear a partir de la posición definida por los otros dos campos. Es posible bloquear bytes no existentes posteriores al fin de archivo, pero no es posible bloquearlos antes del inicio del archivo. A partir del kernel de Linux 2.4.21 es posible especificar un valor de l_len negativo. Esta solicitud se aplica al rango (l_start – abs(l_len), l_start -1).

El valor 0 de l_len tiene un significado especial "bloquear todos los bytes del archivo, desde la posición especificada por l_whence y l_start hasta el fin de archivo sin importar cuanto crezca el archivo". Esto es conveniente si no conocemos de antemano cuantos bytes vamos a añadir al archivo. Para bloquear el archivo completo, podemos especificar l_whence como SEEK_SET y l_start y l_len a 0.

Funciones adicionales:

Setvbuf:

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

_IONBF → unbuffered

_IOLBF → line buffered

_IOFBF → fully buffered

return value → La función setvbuf () devuelve 0 en caso de éxito. Devuelve distinto de cero en caso de fallo.

```
EJEMPLO : setvbuf(stdout,NULL,_IONBF,0)
```

Strtol:

long int strtol(const char *nptr, char **endptr, int base);

Nos permite pasar un string a long int indicando la base.

Atoi:

int atoi(const char *nptr);

pasar un string a entero.

Strcmp:

int strcmp(const char *s1, const char *s2);

compara dos cadenas.

Return value → Devuelve 0 si s1 y s2 son iguales, <0 si s1 es menor que s2 y >0 si s1 es mayor que s2.

Strcpy:

char *strcpy(char *dest, const char *src);

copia la cadena src en dest.

Return value → Devuelve un puntero a la cadena dest

Strlen:

size_t strlen(const char *s);

return value → Devuelve el tamaño de la cadena

memset:

void *memset(void *s, int c, size_t n);

inicializa a c todos los parametros de s teniendo en cuenta su tamaño en n.

EJEMPLOS:

```
struct sigaction act;


//Iniciamos a 0 todos los elementos de la estructura act
memset (&act, 0, sizeof(act));
```

/* Elimina el contenido del buffer */

memset(buffer,0,sizeof(buffer));