ESQUEMA SO MÓDULO 2.

Índice

Código vario

perror

Ejemplo:

```
if( 3 != 4){
        perror("Error");
        exit(-1);
}
```

strtol

Transforma cualquier string en un long int (vale int)
Ejemplo:

```
int num = strtol("223",NULL,10);
```

Opción inversa: sprintf(str, "%d", INT); - donde str es un string

exit

Lo mejor es hacer exit(EXIT_FAILURE) o exit(EXIT_SUCCES)

Interacción con archivos

Entrada/salida sin búfer: open, read, write, lseek, close

Devuelven un fd, o -1 si hay error

Por defecto, la entrada estándar tiene fd=0; la salida estándar, fd=1; y la salida de error estándar, fd=2. STDIN_FILENO, STDOUT_FILENO y STDERR_FILENO, respectivamente, definidas en <unistd.h>.

Cada archivo tiene una posición de lectura/escritura actual (current file offset). Está representado por un entero no negativo que mide el número de bytes desde el comienzo del archivo. Por defecto, esta posición es inicializada a 0 cuando se abre un archivo, a menos que se especifique al opción O_APPEND. La posición actual (current_offset) de un archivo abierto puede cambiarse explícitamente utilizando la llamada al sistema lseek.

Open

int open(const char *pathname, int flags, mode_t mode);
Devuelve el nuevo fd, o -1 si ha habido error
Open puede abrir un archivo o crear otro. Si se desea crear otro, mirar *umask.

Donde:

*pathname:	fd
flags:	O_RDONLY, O_WRONLY, O_RDWR
file creation flags:	O_CLOEXEC, O_CREAT, O_DIRECTORY, O_EXCL, O_NOCTTY, O_NOFOLLOW, O_TMPFILE, O_TRUNC, siendo opcionales y añadibles con "|"
file status flags:	listado completo en man 2 open
mode:	Sólo se usa si en flags está O_CREAT. Aquí se ponen los permisos del nuevo archivo creado.

Close

int close(int fd);
Devuelve 0 si éxito, -1 si error
Close cierra un fd

Read

ssize_t read(int fd, void *buf, size_t count);
Devuelve el número de bytes leídos, -1 si hay error.

Lee de un fd un número count de bytes en el búffer pasado por *buf.

Read comienza a leer desde el current size offset, que aumenta después de la lectura. Si el current size offset sobrepasa el final del archivo, read devuelve 0.

Write

ssize_t write(int fd, const void *buf, size_t count);
Devuelve el n.º de bytes escritos, -1 si error
Escribe en el archivo que corresponda al fd lo que esté almacenado en *buf con un máximo de count bytes

Lseek

off_t lseek(int fd, off_t offset, int whence);
Devuelve el file offset resultante

Avanza o retrocede el current file offset según queramos. Whence indica si avanza desde el inicio del archivo, el current file offset, o el final del archivo (SEEK_SET, SEEK_CUR, SEEK_END) respectivamente. En offset se introduce el número de bytes a avanzar desde whence (se admiten números negativos).

Atributos de archivos

stat

int stat(const char *pathname, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *pathname, struct stat *buf);
Devuelven 0 si éxito, -1 si error
Guarda los atributos del archivo pasado (fd o pathname) en un struct de tipo stat antes creado. Los atributos se pueden consultar en man 2 stat.

Por ejemplo, para consultar los permisos, miramos st_mode:

```
stat(pathname, &sb);
        if ((sb.st_mode & S_IFMT) == S_IFREG) {
           /* Handle regular file */
        }
```

La lista completa de banderas (como S_IFREG) se puede ver en man 2 stat.

También se pueden consultar de la siguiente forma:

```
stat(pathname, &sb);
        if (S_ISREG(sb.st_mode)) {
           /* Handle regular file */
        }
```

La lista completa de las macros estilo S_ISREG también se consultan en man 2 stat

Llamadas al sistema

Llamadas al sistema relacionadas con permisos de archivos:

umask

mode_t umask(mode_t mask)
Devuelve el antiguo valor de la máscara

umask establece la máscara de usuario a mask & 0777. Específicamente, los permisos presentes en la máscara se desactivan del argumento mode de open (así pues, por ejemplo, si creamos un archivo con campo mode= 0666 y tenemos el valor común por defecto de umask=022, este archivo se creará con permisos: 0666 & ~022 = 0644 = rw-r--r--, que es el caso más normal).

Nota: Para que funcione bien, hay que poner un 0 primero en mask. Por ejemplo: umask(0777) o umask(022).

chmod

int chmod(const char *pathname, mode_t mode);
int fchmod(int fd, mode_t mode);
Uso clásico, devuelve 0, -1 si hay error.

Ejemplo:
chmod(fd1, S_IRGRP|S_IXGRP)

Llamadas al sistema relacionadas con manejo de directorios

opendir, readdir, closedir, seekdir, telldir, rewinddir

opendir

DIR *opendir(const char *name);
DIR *fdopendir(int fd);
Devuelven un puntero a una estructura DIR
Se le pasa el pathname del directorio a abrir, y devuelve un puntero a la estructura de tipo DIR. El tipo DIR está definido en <dirent.h>, y se declara con el *.

readdir

struct dirent *readdir(DIR *dirp);
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.
NOTA: man 3 readdir, no man 2 readdir
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.

La struct dirent es esta:

```
struct dirent {
        long d_ino; /* número i-nodo */
        char d_name[256]; /* nombre del archivo */
};
```

Se declaran con el *

closedir

int closedir(DIR *dirp);
Devuelve 0 si éxito, -1 si error
Cierra un directorio.

seekdir

void seekdir(DIR *dirp, long loc);
permite situar el puntero de lectura de un directorio (loc se obtiene usando telldir).

Telldir

long telldir(DIR *dirp);
Devuelve la posición del puntero de lectura de un directorio.

Rewinddir

void rewinddir(DIR *dirp);
posiciona el puntero de lectura al principio del directorio.

DIR es un struct, que contiene lo siguiente:

```
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;
```

Ejemplo de uso de opendir, readdir y stat

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

        DIR* fddir = opendir("carpeta");
        struct stat atrib;
        struct dirent* puntero;
        int x = 0;
        while( (puntero = readdir(fddir)) != NULL ){
                stat(puntero->d_name,&atrib);
                x++;
                if(!S_ISDIR(atrib.st_mode)){
                                printf("\n%s",puntero->d_name);
                }
                atrib.st_mode = 0;
        }
        printf("%i",x);

}
```

nfwt

int nftw (const char *dirpath, int (*func) (const char *pathname, const struct stat *statbuf,
int typeflag, struct FTW *ftwbuf), int nopenfd, int flags);
Devuelve 0 si éxito, -1 si error
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. Mientras se recorre el árbol, la función nftw abre al menos un descriptor de archivo por nivel del árbol. El parámetro nopenfd especifica el máximo número de descriptores que puede usar. Si la profundidad del árbol es mayor que el número de descriptores, la función evita abrir más cerrando y reabriendo descriptores.

Ejemplo: Ejercicio 3 sesión 2, github de erseco.

Llamadas al sistema para control de procesos

Llamadas para obtener pid:

getpid, getppid

pid_t getpid(); // devuelve el PID del proceso que la invoca.
pid_t getppid(); // devuelve el PID del proceso padre del proceso que la invoca
uid_t getuid(); // devuelve el identificador de usuario real del proceso que la invoca.
uid_t geteuid(); // devuelve el identificador de usuario efectivo del proceso que la invoca.
gid_t getgid(); // devuelve el identificador de grupo real del proceso que la invoca.
gid_t getegid(); // devuelve el identificador de grupo efectivo del proceso que la invoca.

Fork

pid_t fork(void);
Devuelve:
Desde el proceso padre, el pid del hijo.
Desde el proceso hijo, 0.
Crea una copia del proceso, llamada proceso hijo.

Ejemplo:
pid_t pid = fork();

```
if( pid == 0 ){
        if( num%2 == 0 )
                printf(" %d Es divisible entre 2\n",num);
        else printf(" %d NO es divisible entre 2\n",num);
} else{
        if( num%4 == 0 )
                printf(" %d Es divisible entre 4\n",num);
        else printf(" %d NO es divisible entre 4\n",num);
}
```

Wait

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
Devuelve el ID del proceso hijo que termina o 0 en caso de no terminar ningún proceso, -1 en caso de error.

Wait(NULL) equivale a waitpid(-1, NULL, 0);
Waitpid suspende la ejecución actual del proceso en curso hasta que un proceso hijo, especificado en el argumento pid ha terminado, o hasta que se produce una señal cuya acción es terminar con el proceso actual.

El valor de pid puede ser uno de los siguientes:
< -1 Espera a que cualquier proceso hijo cuyo PID del proceso es igual al valor absoluto de pid.
-1 Espera por cualquier proceso hijo; este es el mismo comportamiento que tiene wait.
0 Espera por cualquier proceso hijo cuyo PID es igual al del proceso padre.
> 0 Espera por el proceso hijo cuyo PID es igual al valor de pid (por un proceso determinado).

Opciones:

- WNOHANG: Vuelve inmediatamente si ningún hijo ha terminado.
- WUNTRACED: El padre obtiene información adicional si el hijo recibe alguna de las señales SIGINT, SIGTTOU, SIGSSTP, SIGTSTOP.

- **Status:**

  Si status no es NULL, waitpid almacena la información de estado en la memoria apuntada por status.

- WIFEXITED(status): Es distinto de cero si el hijo terminó normalmente.
- WEXITSTATUS(status): Permite obtener el valor devuelto por el proceso hijo en la llamada exit o el valor devuelto en la función main, utilizando la sentencia return. Esta macro solamente puede ser tenida en cuenta si WIFEXITED devuelve un valor distinto de cero.
- WIFSIGNALED(status): Devuelve true si el proceso hijo terminó a causa de una señal no capturada.

- WTERMSIG(status): Devuelve el número de la señal que provocó la muerte del proceso hijo. Esta macro sólo puede ser evaluada si WIFSIGNALED devolvió un valor distinto de cero.
- WIFSTOPPED(status): Devuelve true si el proceso hijo que provocó el retorno está actualmente parado; esto solamente es posible si la llamada se hizo usando WUNTRACED o cuando el hijo está siendo rastreado.
- WSTOPSIG(status): Devuelve el número de la señal que provocó la parada del hijo. Esta macro solamente puede ser evaluada si WIFSTOPPED devolvió un valor distinto de cero.

Exec

```
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[]);
```

Devuelve -1 si hay error, nada si no lo hay

Es como fork, solo que la nueva hebra ejecuta un programa distinto.

Ejemplo:

```
execl(''/usr/bin/ldd'',''ldd'',''./tarea5'',NULL)
```

Cuando un proceso ejecuta una llamada exec, el espacio de direcciones de usuario del proceso se reemplaza completamente por un nuevo espacio de direcciones; el del programa que se le pasa como argumento, y este programa comienza a ejecutarse en el contexto del proceso hijo empezando en la función main. El PID del proceso no cambia ya que no se crea ningún proceso nuevo.

El primer argumento, por convenio, debe apuntar al nombre del archivo asociado con el programa que se esté ejecutando. La lista de argumentos debe ser terminada por un puntero NULL.

El const char *arg y los puntos suspensivos siguientes, en las funciones execl, execlp, y execle, son los argumentos (incluyendo su propio nombre) del programa a ejecutar: arg0, arg1, ..., argn. Todos juntos, describen una lista de uno o más punteros a cadenas de caracteres terminadas en cero. El primer argumento, por convenio, debe apuntar al nombre de archivo asociado con el programa que se esté ejecutando. La lista de argumentos debe ser terminada por un puntero NULL.

Las funciones execv y execvp proporcionan un vector de punteros a cadenas de caracteres terminadas en cero, que representan la lista de argumentos disponible para el nuevo programa. El primer argumento, por convenio, debe apuntar al nombre de archivo asociado con el programa que se esté ejecutando. El vector de punteros debe ser terminado por un puntero NULL.

La función execle especifica el entorno del proceso que ejecutará el programa mediante un parámetro adicional que va detrás del puntero NULL que termina la lista de argumentos de la lista de parámetros o el puntero al vector argv. Este parámetro adicional es un vector de punteros a cadenas de caracteres acabadas en cero y debe ser terminada por un puntero NULL.

Si a un archivo se le deniega el permiso (execve devuelve EACCES), estas funciones continuarán buscando en el resto de la lista de búsqueda. Si no se encuentra otro archivo devolverán el valor EACCES en la variable global errno .

Cauces

Cauces con nombre: FIFO

mknod, mkfifo, unlink

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

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

int unlink(const char *pathname);

Devuelven 0 si éxito, -1 si error.

Crean un archivo fifo con el nombre *pathname, y permisos mode.

A mknod no hacerle mucho caso. Se puede ver su uso en man 2 mknod.

Mkfifo crea un cauce de forma similar a pipe, solo que con un archivo físico creado de forma manual. Mode son los permisos del archivo. En los archivos fifo no podemos usar lseek. Se usan read y write para leer y escribir a través del cauce. Read se bloquea mientras no haya datos para leer, y devuelve 0 si todos los productores (write) han cerrado. Ejemplo en la guía de prácticas, pag 110/111.

Unlink borra el cauce.

Los cauces con nombre se crean (llamadas al sistema mknod y mkfifo) en el sistema de archivos en disco como un archivo especial, es decir, consta de un nombre que ayuda a denominarlo exactamente igual que a cualquier otro archivo en el sistema de archivos, y por tanto aparecen contenidos/asociados de forma permanente a los directorios donde se crearon.

Los procesos abren y cierran un archivo FIFO usando su nombre mediante las ya conocidas llamadas al sistema open y close, con el fin de hacer uso de él.

Cualesquiera procesos pueden compartir datos utilizando las ya conocidas llamadas al sistema read y write sobre el cauce con nombre previamente abierto. Es decir, los cauces con nombre permiten comunicar a procesos que no tienen un antecesor común en la jerarquía de procesos de UNIX.

El archivo FIFO permanece en el sistema de archivos una vez realizadas todas las E/S de los procesos que lo han utilizado como mecanismo de comunicación, hasta que se borre explícitamente (llamada al sistema unlink) como cualquier archivo.

Cauce sin nombre:

pipe

int pipe(int pipefd[2]);

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

Devuelve 0 si éxito, -1 si error

USAR ANTES DE FORK

Crea un cauce. Por convenio, pipefd[0] es el consumidor, y pipefd[1] el productor. Las flags se pueden consultar en man pipe. Se suelen usar en procesos padre-hijo, donde los pipefd se clonan también, por lo que están duplicados. Si el sentido de flujo de inf es de padre a hijo, el padre hace close en pipefd[1] e hijo en pipefd[0]. Si el sentido es al contrario, pues viceversa. Mirar pág 113/114 de la guía.

No tienen un archivo asociado en el sistema de archivos en disco, sólo existe el archivo temporalmente y en memoria principal.

Al crear un cauce sin nombre utilizando la llamada al sistema pipe, automáticamente se devuelven dos descriptores, uno de lectura y otro de escritura, para trabajar con el cauce. Por consiguiente no es necesario realizar una llamada open.

Los cauces sin nombre sólo pueden ser utilizados como mecanismo de comunicación entre el proceso que crea el cauce sin nombre y los procesos descendientes creados a partir de la creación del cauce.

El cauce sin nombre se cierra y elimina automáticamente por el núcleo cuando los contadores asociados de números de productores y consumidores que lo tienen en uso valen simultáneamente 0.

Duplicar fd:

Dup

int dup(int oldfd);

int dup2(int oldfd, int newfd);

Devuelve el nuevo fd si éxito, -1 si error

Dub copia el fd pasado al último fd cerrado. Por lo que si hacemos close(1); dub(32), el resultado práctico sería el mismo que el explicado aquí abajo para dub2.

Las llamadas al sistema dup proporcionan un modo para duplicar un descriptor de archivos, presentando dos o más descriptores diferentes que acceden al mismo archivo. Se pueden usar para leer y escribir en diferentes partes del archivo. La llamada al sistema dup duplica un descriptor de archivo, fd, enviando un nuevo descriptor. La llamada al sistema dup2 copia eficazmente un descriptor de archivo a otro especificando qué descriptor usar para la copia.

Supongamos la siguiente tabla de descriptores de archivos:

0	→	Entrada estándar (teclado)
1	→	Salida estándar (pantalla)
2	→	Salida estándar errores (pantalla)
32	→	datos.txt en modo escritura

Fíjese que ya existe un descriptor fd para manejar el fichero datos.txt, que se obtuvo mediante la llamada open, por tanto, el valor del descriptor es seleccionado por el sistema operativo (en nuestro caso, hemos seleccionado el valor 32 arbitrariamente).

A partir de dicha situación, tras realizar la siguiente llamada:

dup2(fd, STDOUT_FILENO);

El resultado sobre la tabla de descriptores es la siguiente:

0	→	Entrada estándar (teclado)
1	→	datos.txt en modo escritura
2	→	Salida estándar errores (pantalla)
32	→	datos.txt en modo escritura

Por tanto, tras la llamada dup2 se genera el duplicado, de manera que el descriptor 1 (STDOUT_FILENO) ya no apunta a la pantalla sino al fichero datos.txt:

De esta manera, cuando invoquemos a printf el mensaje que se pase como parámetro no se imprimirá en pantalla, sino que se almacenará en el fichero datos.txt.

Señales

man 7 signal, tabla de señales y acciones por defecto. 121/122 guia prácticas para verlas en español.

Cada señal posee un nombre que comienza por SIG. Realmente, cada señal lleva asociado un número entero positivo, que es el que se entrega al proceso cuando éste recibe la señal. Se puede usar indistintamente el número o la constante que representa a la señal.

Kill

int kill(pid_t pid, int sig);

Devuelve 0 si éxito, -1 si error

No sólo mata, sirve para mandar cualquier señal.

→ 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.

Sigaction

int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);

Devuelve 0 si éxito, -1 si error

La llamada al sistema sigaction se emplea para cambiar la acción que por defecto toma un proceso cuando recibe una determinada señal.

Ejemplo en pág 128 guía prácticas

El significado de los parámetros de la llamada es el siguiente:

• 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.

Devuelve 0 en caso de éxito.

La estructura sigaction es la siguiente:

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

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 /* señal POSIX.1b */
        si_int;
        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 */
```

Los posibles valores para cualquier señal se pueden consultar con man sigaction.
Nota: El elemento sa_restorer está obsoleto y no debería utilizarse. POSIX no especifica un elemento sa_restorer.

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.

Sigprocmask

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);

Se emplea para examinar y cambiar la máscara de señales. More in pág 133 guía prácticas

Sigpending

int sigpending(sigset_t *set);

Permite examinar el conjunto de señales bloqueadas y/o pendientes de entrega. La máscara de señal de las señales pendientes se guarda en set. More in pág 134 guía prácticas

Sigsuspend

int sigsuspend(const sigset_t *mask);

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. More in pág 134 guía prácticas

Control de archivos: fcntl

fcntl

int fcntl(int fd, int orden, /* argumento_orden */);

La llamada al sistema fcntl (file control) es una función multipropósito que, de forma general, 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.

El argumento orden admite un rango muy diferente de operaciones a realizar sobre el fd. El tercer argumento, que es opcional, va a depender de la orden indicada. A continuación, mostramos las órdenes admitidas:

→ 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-exec del archivo indicado. (Si la bandera close-on-exec está activa en un descriptor, al ejecutar la llamada exec() el proceso hijo no heredará este descriptor.). 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.

Más info en la guía de prácticas

FIN.