Módulo II

Sesión 1

Entrada/Salida de archivos regulares:

La mayor parte de las E/S en UNIX se pueden realizar utilizando open, read, write, lseek y close. Son funciones sin búfer, ya que cada read o write invoca una llamada al sistema en el núcleo y no se almacena en un búfer de la biblioteca.

- Para el núcleo, todos los archivos abiertos son identificados con descriptores de archivo.

Descriptor de archivo → entero no negativo por el que todos los archivos abiertos son identificados.

- **Entrada estándar** = 0 → Estándar POSIX 2.10 es STDIN_FILENO
- **Salida estándar** = 1 → Estándar POSIX 2.10 es STDOUT_FILENO
- **Salida error** = 2 → Estándar POSIX 2.10 es STDERR_FILENO

Current file offset: posición de lectura/escritura actual, lo tiene cada archivo abierto.

- Es un entero no negativo que mide el **número de bytes** desde el **comienzo** del archivo **hasta la posición**.
- Por defecto esa posición es **inicializada a 0** a no ser que usemos **O_APPEND**.
- Podemos **cambiar** explícitamente el **current offset** de un archivo ABIERTO usando **lseek**.

Llamadas al Sistema

1. **open**: puede abrir archivo ya existente o crear uno nuevo.

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

Argumentos

mode: especifica los permisos a emplear si se crea un nuevo archivo. **(OPCIONAL)**
pathname: es el archivo que utiliza el open, si no existe y está el flag **O_CREAT**, lo crea.
flags: especifica las diferentes funciones del open, deben incluir **O_RDONLY** (read only), **O_WRONLY** (write only) ó **O_RDWR** (read/write). Si ponemos varios flags debemos separarlos con "|" ~ Ver lista de flags en "man 2 open".

Return

La llamada open() devuelve el file descriptor (entero no negativo que hace referencia al archivo abierto) ó -1 si ha ocurrido algún error. El descriptor devuelto por una llamada exitosa será el número más bajo del fd que no esté abierto actualmente por el proceso.

Ejemplo

```
fd = open("F1", O_RDONLY);
        // En fd guarda el file descriptor del archivo F1 que se ha abierto sólo para lectura.
close(fd);      // Con close fd cerramos el archivo F1.
```

open("F1",O_CREAT, <permissions>) == **creat("F1", <permissions>)**

---

2. **lseek**: reposiciona el file offset de un archivo abierto, lo cambia.

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

Argumentos

fd: es el file descriptor del archivo de lectura / escritura
offset: es el número de bytes que contará.
whence: especifica desde dónde empieza a contar bytes.

- **SEEK_SET** → Inicio del fichero
- **SEEK_CUR** → Posición actual del file offset
- **SEEK_END** → Final del fichero

Return

La llamada lseek() devuelve la localización (en bytes) del offset midiendo desde el principio del archivo ó -1 si ha ocurrido algún error.

Ejemplo

```
lseek(fd, 40, SEEK_SET);  // Posiciona el file offset 40 bytes desde el inicio del fichero
```

---

3. **read**: lee desde un file descriptor.
   ssize_t read(int fd, void *buf, size_t count);

Argumentos

fd: es el file descriptor del archivo
buf: es el buffer o array donde meterá los bytes que lea.
count: son los bytes que intenta leer del archivo

Return

La llamada read() devuelve el número de bytes leídos, 0 si ha llegado al final del archivo ó -1 si ha ocurrido algún error.

Ejemplo

```
char caracter[1];
int filein = open(argv[1], O_RDONLY);
read(filein, caracter, 1);  // Lee un byte y lo guarda en "caracter" del archivo en argv[1]
```

---

4. **write**: escribe desde el buffer al archivo.
   ssize_t write(int fd, const void *buf, size_t count);

Argumentos

fd: es el file descriptor del archivo
buf: es el buffer o array de donde obtendrá los bytes a escribir.
count: son los bytes que intentará sacar del buffer para escribir en el archivo

Return

La llamada write() devuelve el número de bytes escritos, 0 si no ha escrito nada ó -1 si ha ocurrido algún error.

Ejemplo

```
char buf[]="abcdefghij";
int fd = open("archivo", O_CREAT | O_TRUNC | O_WRONLY, S_IRUSR | S_IWUSR);
write(fd, buf, 10);  // Escribe 10 bytes de buf en el archivo que indica fd "archivo": abcdefghij
```

Metadatos de un Archivo

Tipos de archivos en Linux:

- **Archivo regular:** Contiene datos de cualquier tipo. Para el núcleo no existe diferencia entre este tipo de archivo y el archivo de ficheros.
- **Archivo de directorio:** Un directorio es un archivo que contiene los nombres de otros archivos y punteros a la información de esos archivos.
- **Archivo especial de dispositivo de caracteres:** Representar ciertos dispositivos
- **Archivo especial de dispositivo de bloques:** Representar discos duros, CDROM...
- **FIFO o Cauce con Nombre:** Se usa en comunicación entre procesos (IPC)
- **Enlace simbólico:** Tipo de archivo que apunta a otro archivo
- **Socket:** Se usa para comunicación en red entre procesos y para comunicar procesos en un único nodo (host).

1. stat: se usa para obtener los metadatos de un archivo.

int stat(const char *pathname, struct stat *statbuf);

La estructura de stat es:

Estructura stat:

```
dev_t st_dev;              /* número de dispositivo (filesystem) */
dev_t st_rdev;             /* número de dispositivo para archivos especiales */
ino_t st_ino;              /* número de inodo */
mode_t st_mode;            /* tipo de archivo y mode (permisos) */ consultar flags
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 */ bloque = 512B
time_t st_atime;           /* hora último acceso */
time_t st_mtime;           /* hora última modificación */
time_t st_ctime;           /* hora último cambio */
```

Lstat hace lo mismo que stat pero si se ejecuta sobre un **enlace simbólico** devuelve **información sobre el enlace, no sobre el archivo** al que hace referencia.

Macros POSIX para comprobar el tipo de fichero:

- **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

Sesión 2

Llamadas al sistema (II):

1. **umask:** máscara que modifica el mode, permisos (**mode & umask** (donde & es AND)). Si el directorio padre tiene un ACL por defecto, el umask se ignora y se hereda el ACL. ACL es access control lists, te dice quién tiene acceso y los permisos asociados a ese directorio.

   mode_t umask(mode_t mask);

   Argumentos

   mask: 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 por defecto de **umask=022**, este archivo se creará con permisos: **0666 & ~022 = 0644 = rw- r-- r--**).
   - Si no ponemos valores referentes a los 3 campos, estos se rellenan con todos los permisos

   Return

   La llamada umask() devuelve siempre (nunca da error) el valor previo de la máscara.

   Ejemplo

```
fd1 = open("archivo1", O_CREAT|O_TRUNC|O_WRONLY, S_IRGRP|S_IWGRP|S_IXGRP)
umask(0)                  // El umask 0 hace que no se desactive ninguno de los permisos
fd2 = open("archivo2", O_CREAT|O_TRUNC|O_WRONLY, S_IRGRP|S_IWGRP|S_IXGRP)

--- -r-x --- 1     user user    0 dic 14 10:11  archivo1
--- -rwx--- 1     user user    0 dic 14 10:11  archivo2
```

---

2. **chmod**: cambia los permisos de un archivo dado.

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

   Argumentos

   path: especifica el archivo
   mode: especifica los permisos, podemos usar | .

---

3. **fchmod**: igual que chmod pero para archivos que han sido abiertos antes con open.

   int fchmod(int fildes, mode_t mode);

   Argumentos

   fildes: especifica el file descriptor del archivo abierto

   Return

   La llamada chmod() devuelve 0 en caso de éxito ó -1 si ha ocurrido algún error.

   Ejemplo

```
int
fd=open("archivo",O_CREAT|O_TRUNC|O_WRONLY,S_IRGRP|S_IWGRP|S_IXGRP);
chmod( "archivo", S_IRWXU | S_IRGRP | S_IWGRP | S_IROTH );      // - rwx rw- r--
fchmod( fd, S_IRWXU | S_IRWXG | S_IRWXO );                      // - rwx rwx rwx
```

Funciones de manejo de directorios

1. **opendir:** abre el stream del directorio correspondiente. Puede usarse **fdopendir(int fd)**.
   DIR *opendir(const char *name);

Argumentos

name: especifica el nombre del directorio

Return

Devuelve un puntero al stream del directorio o NULL si ha ocurrido algún error.

---

2. **readdir**: cambia los permisos de un archivo dado.
   struct dirent *readdir(DIR *dirp);

Argumentos

dirp: puntero al stream del directorio que va a leer.

Return

Devuelve un puntero a la siguiente entrada en el stream del directorio o NULL. Lo que devuelve readdir puede ser sobreescrito si hacemos varias llamadas con el mismo dirp.

```
struct dirent {
        ino_t  d_ino;              /* Inode number */
        off_t  d_off;              /* Not an offset; see below */
        unsigned short  d_reclen;  /* Length of this record */
        unsigned char  d_type;     /* Type of file; not supported by all filesystem types */
        char  d_name[256];         /* Null-terminated filename */
    };
```

Ejemplo

```
DIR *dirp = opendir(pathname);          // abre el stream de pathname
struct dirent *ed = readdir(dirp);      // ed señala a la siguiente entrada en pathname
closedir(dirp);                         // cierra el stream
```

---

3. **closedir**: cierra un directorio.
   int closedir(DIR *dirp);

Return

Devuelve 0 si tiene éxito o -1 si ha ocurrido algún error.

---

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

---

5. **telldir**: devuelve la posición actual del puntero de lectura de un directorio.
   long telldir(DIR *dirp);

Return

Devuelve la localización actual en el stream del directorio ó -1 en caso de error.

---

6. **seekdir**: permite cambiar la posición del puntero de lectura de un directorio.
   void seekdir(DIR *dirp, long loc);

Argumentos

dirp: puntero al stream del directorio que va a leer.
loc: valor que ha retornado previamente la función tell dir

Sesión 3

Identificadores de procesos

Identificador de Proceso (**PID**): entero no negativo que identifica inequívocamente a un proceso. El proceso init (encargado de inicializar el sistema) tiene PID = 1, es la raíz.

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

Identificador de Usuario Real (**UID**): comprobación que realiza el programa logon proporcionándole un login y un password para identificar la línea del archivo /etc/passwd que corresponde al usuario para comprobar la clave de shadow.

- **UID efectivo** (**euid**): se corresponde con el UID real salvo que ejecute un programa con el bit SUID activo, en este caso se corresponderá con el UID del propietario del ejecutable.

Identificador de Grupo (**GID**): similar al UID pero comprobando el grupo principal en el archivo de passwords.

- **GID efectivo** (**egid**): igual que con el UID efectivo.

getuid(); **geteuid();** **getgid();** **getegid();**

Llamada al Sistema (lll):

1. **fork**: crea un proceso hijo duplicando el proceso que llama a fork que pasa a ser el proceso padre. Con fork es la única forma que tiene el núcleo de crear un nuevo proceso. El hijo será una copia idéntica al padre y ambos ejecutarán desde el fork de forma concurrente.

```
pid_t fork( );
```

Return

La llamada fork( ), si tiene éxito, el padre el padre tendrá el PID del proceso hijo y devuelve 0 en el hijo, si no ha salido bien el PID del padre será -1 y no se creará ningún hijo.

Advertencia: Zombies

Si el padre finaliza antes que el hijo, el hijo tendrá de padre el proceso init, el **init ya hará los wait** necesarios para que no se queden zombies, en caso de que el hijo termine antes que el padre, el **padre DEBE hacer wait** para liberar y que no quede zombie, si no se quedará ocupando recursos.
Si el padre tiene más de un hijo hay que hacer un **wait para cada hijo**.
Si queremos esperar a un hijo en concreto podemos hacer un **waitpid**(PID del hijo concreto)

Ejemplo

```
pid_t pid;
if( (pid = fork( )) < 0)                // Llamada ha salido mal
        perror("Error en el fork, no se ha podido crear proceso hijo");
else if( pid==0 )                       // Proceso hijo ejecutando el programa
        printf("Ejecutando el proceso hijo...");
else                                    // Proceso padre ejecutando el programa
        printf("Ejecutando el proceso padre...");
```

wait y waitpid

wait | waitpid: se usan para esperar hasta que se produzca un cambio de estado en un hijo del proceso que llama y obtiene información sobre el hijo cuyo estado ha cambiado (el hijo finaliza [devuelve **0**], se ha parado o ha sido reanudado [devuelve **nº señal**]).

```
pid_t wait(int *wstatus);                                  // Tiene un estado
pid_t waitpid(pid_t pid, int *wstatus, int options);       // Tiene además el pid
```

---

2. exec: familia de llamadas que reemplazan completamente el espacio de direcciones de usuario del proceso que ejecuta la llamada por el del programa que se le pasa como argumento, y este, empieza a ejecutarse en el contexto del proceso hijo empezando en el main. El proceso hijo puede ejecutar un programa distinto al padre

```
int execl(const char *path, const char *arg, …, NULL);
int execlp (const char *file, const char *arg, ..., NULL);
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[]);
```

Argumentos

file / path: camino del archivo ejecutable.
const char *arg: argumentos (incluido el nombre) del programa a ejecutar, acaba con NULL
char *const argv[]: vector de punteros a cadenas terminadas en cero, representan la lista de argumentos disponibles para el nuevo programa (el primero apunta al nombre).

Return

Sólo devuelven un valor (-1) si ha ocurrido un error.

Ejemplo

```
if( (execl("/usr/bin/ldd","ldd","./tarea5",NULL)<0)) {
        perror("\nError en el execl");
        exit(-1);
}
```

// ldd es un programa, que se encuentra en /usr/bin, al que hay que pasarle otro archivo (./tarea5) como argumento. Como último argumento escribimos NULL indicando que acaban

---

3. clone: crea un nuevo proceso, a diferencia de fork, permite al hijo compartir partes del contexto de ejecución. Con fork hacemos copia de las estructuras de datos y con clone las compartimos.
Si con fork el padre tenía abierto un archivo, el hijo lo tendrá abierto, pero si el hijo lo cierra, al padre no se le cerrará. Esto con clone no ocurre, si lo cierra uno lo cierra para todos porque está compartido.

```
int clone(int (*func) (void *), void *child_stack, int flags, void *func_arg, …  );
```

Argumentos

func: función que comienza ejecutando el hijo.
child_stack: puntero a pila que debe usar el hijo y que se tiene que haber reservado antes.
flags: indicadores de clonación (ver el man).
func_arg: argumentos para la función (func) donde va a empezar a ejecutar el hijo.

Return

Devuelve un identificador de una hebra (tid) si ha salido bien o -1 si ha ocurrido algún error.

Ejemplo

```
int thread(void *p) {  }

void **stack;
int i;
stack = (void **)malloc(15000);
if (!stack)
        return -1;
i = clone(thread, (char*) stack + 15000, CLONE_VM|CLONE_FILES, NULL);    // flags
if (i == -1)
        perror("clone");
```

Información sobre bits SUID, SGID y STICKY BIT

SUID (setuid)

En vez de la 'x' en el grupo del usuario encontramos ahora una 's' (suid). Al activar este bit obliga al archivo ejecutable binario a ejecutarse como si lo hubiera lanzado el usuario propietario y no realmente quien lo lanzó o ejecutó. Es decir, es poder invocar un comando propiedad de otro usuario (generalmente de root) como si uno fuera el propietario.

SGID (setgid)

El bit SGID funciona exactamente igual que el anterior solo que aplica al grupo del archivo. El usuario que pertenezca al grupo podrá ejecutarlo. También se muestra como una 's' en vez del bit 'x' en los permisos del grupo.

STICKY BIT (Bit de persistencia)

Este bit se aplica para directorios como en el caso de /tmp y se indica con una 't' en vez de la 'x' en los permisos de otros. Lo que hace el bit de persistencia en directorios compartidos por varios usuarios, es que el sólo el propietario del archivo pueda eliminarlo del directorio. Es decir cualquier otro usuario va a poder leer el contenido de un archivo o ejecutarlo, pero sólo el propietario original podrá eliminarlo o modificarlo.

Sesión 4

Concepto y tipos de cauce

Cauce: mecanismo para la comunicación de información y sincronización entre procesos. Los datos pueden ser escritos por varios procesos al cauce y a su vez leídos por otros. Los datos se tratan en orden FIFO (First In First Out). Hay cauces sin nombre y con nombre.

- **Cauces sin nombre:** se crean con la llamada al sistema **pipe**, devuelve dos descriptores (lectura y escritura). No es necesario hacer un open.
- **Cauces con nombre (archivos FIFO):** se crean con **mknod** y **mkfifo** como un archivo especial que consta de un nombre. Los procesos abren y cierran un archivo FIFO usando su nombre mediante **open** y **close**. Cualquier proceso puede compartir datos utilizando **read** y **write**. Hay que borrarlo explícitamente con **unlink**.

Cauces con Nombre

1. **mknod**: permite crear archivos especiales como FIFO o archivos de dispositivo.
   int mknod (const char *FILENAME, mode_t MODE, dev_t DEV);

Argumentos

filename: nombre del archivo creado.
mode: especifica los valores almacenados en el st_mode del i-nodo correspondiente:

- **S_IFCHR:** archivo de dispositivo orientado a caracteres.
- **S_IFBLK:** archivo de dispositivo orientado a bloques.
- **S_IFSOCK:** archivo para un socket.
- **S_IFIFO:** archivo para un FIFO

dev: especifica a qué dispositivo se refiere. Para un cauce FIFO el valor del argumento es 0.

Return

Devuelve 0 si ha ido bien o -1 si ha ocurrido un error.

Ejemplo

```
mknod("/tmp/FIFO", S_IFIFO|0666,0);
        // Ha creado un cauce FIFO con permisos (0666 & ~umaskInicial)
```

---

2. **mkfifo**: llamada específica para crear archivos FIFO (funciona igual que mknod).
   int mkfifo (const char *FILENAME, mode_t MODE));

Argumentos

filename: nombre del archivo creado.
mode: especifica los permisos del archivo FIFO.

Ejemplo

```
mkfifo("/tmp/FIFO", 0666);
```

Utilización de un cauce FIFO: las operaciones de E/S son las mismas salvo por **lseek** que ahora carece de sentido. La llamada **read** es bloqueante para los consumidores cuando no hay datos que leer en el cauce y desbloquea devolviendo 0 cuando todos los procesos que tenían el cauce abierto para escritura, lo cierran o terminan.

Cauces sin Nombre

1. **pipe**: permite crear un cauce sin nombre. Si luego hacemos fork, el priceso hijo heredará los descriprores del cauce.

int pipe(int pipefd[2]);

Argumentos

pipefd: vector de enteros que contiene dos file descriptor fd[0] para el read y fd[1] write.

Return

Devuelve 0 si ha ido bien o -1 si ha ocurrido un error.

Ejemplo

```
int fd[2], numBytes;
char buffer[80], mensaje[]= "\nEl mensaje transmitido por un cauce!!\n";
pipe(fd);        // Crea el cauce sin nombre y mete en fd[0] el fd de read y en fd[1] de write
numBytes= read(fd[0],buffer,sizeof(buffer));
write(fd[1],mensaje,strlen(mensaje)+1);
```

---

2. **dup**: duplica un file descriptor. Utiliza el número más bajo de file descriptor que no se esté usando para el nuevo descriptor. Puede ser útil usar close antes

int dup(int oldfd);

Argumentos

oldfd: file descriptor que queremos duplicar.

Return

Devuelve 0 si ha ido bien o -1 si ha ocurrido un error.

---

3. **dup2**: igual que dup pero en lugar de usar el número más bajo de fd usa el newfd para duplicarlo. No hay que usar close antes, lo hace dup2 automáticamente.

int dup2(int oldfd, int newfd);

Argumentos

oldfd: file descriptor que queremos duplicar.
newfd: file descriptor que almacenará el duplicado. Si estaba abierto antes, lo cierra.

Return

Devuelve el nuevo file descriptor si ha ido bien o -1 si ha ocurrido un error.

Ejemplo: implementar con cauce sin nombre "ls | sort -r". Parecido a tarea7.c

```
int main(int argc, char *argv[])
{
    int fd[2];
    pid_t PID;
    pipe(fd);                          // crea un pipe
    if ( (PID= fork())<0) {            // ahora el proceso hijo y el padre comparten los fd
        perror("fork");
        exit(EXIT_FAILURE);
    }
    if(PID == 0) {                     // el proceso hijo hará el ls y se lo enviará al padre
        close(fd[0]);                  // establecemos la dirección del flujo cerrando la lectura
    close(STDOUT_FILENO);                  // cerrar primero la salida estándar del proceso
    hijo
    dup(fd[1]);                        // hacemos el duplicado del file descriptor del write
    execlp("ls","ls",NULL);            // orden ls
    }
    else {                 // el proceso padre hace el sort -r con la información del ls que envía
    el hijo
    close(fd[1]);                       // establecemos la dirección del flujo cerrando la
    escritura
        dup2(fd[0],STDIN_FILENO); // duplicamos el fd del read a la vez que cerramos la
    entrada
        execlp("sort","sort","-r",NULL);          // orden sort -r
    }
    return EXIT_SUCCESS;
}
```

Sesión 5

Señales

Señal: mecanismo básico de sincronización que usa el núcleo para indicar a los procesos que han ocurrido determinados eventos asíncronos/síncronos con su ejecución. Los procesos pueden enviarse señales para notificar algún evento, invocando a un manejador de señales.
La invocación al manejador de la señal puede interrumpir el flujo de control del proceso. Cuando se entrega la señal, el kernel invoca al manejador, y cuando este retorna, la ejecución sigue por donde iba.
Las señales bloqueadas de un proceso se almacenan en la máscara de bloqueo de señales.

Llamada al Sistema para Señales:

1. **kill**: se utiliza para enviar una señal a un proceso o conjunto de procesos.
   int kill(pid_t pid, int sig);

Argumentos

pid: valor según el cuál la llamada hará una cosa u otra:
- **pid > 0** → envía la señal **sig** al proceso con identificador de proceso igual a **pid**.
- **pid = 0** → envía **sig** a cada proceso en el grupo de procesos del proceso actual.
- **pid = -**1 → se envía **sig** a cada proceso excepto al primero (desde los números más altos de la tabla hasta los más bajos).
- **pid < -1** → se envía **sig** a cada proceso en el grupo de procesos **-pid**.

sig: señal que se envía.
- Si **sig = 0**, no se envía ninguna señal, pero se hace la comprobación de errores.

Return

Devuelve 0 si ha ido bien o -1 si ha ocurrido un error.

---

2. **sigaction**: permite establecer la acción que realizará un proceso como respuesta a la recepción de una señal. Las únicas señales que no pueden cambiar su acción por defecto son: SIGKILL y SIGSTOP.
   int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);

Argumentos

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

Return

Devuelve 0 si ha ido bien o -1 si ha ocurrido un error.

Struct sigaction

```
struct sigaction {
        void (*sa_handler)(int);
        void (*sa_sigaction)(int, siginfo_t *, void *);   // siginfo_t es un struct
        sigset_t sa_mask;
        int sa_flags;
        void (*sa_restorer)(void);       // OBSOLETO, no usar
}
```

Parámetros del Struct Sigaction

sa_handler: especifica la acción que se asocia a signum: SIG_DFL acción predeterminada, SIG_IGN ignorar la señal o un puntero a una función manejadora de la señal.
sa_mask: establece una máscara de señales que se bloquean durante la ejecución del manejador. A menos que se activen las opciones SA_NODEFER o SA_NOMASK la señal que **lance el manejador será bloqueada**. Para asignar valores se usan las funciones:

- **int sigemptyset(sigset_t *set)**: inicializa a un conjunto vacío de señales.
- **int sigfillset(sigset_t *set)**: inicializa un conjunto con todas las señales.
- **int sigismember(const sigset_t *set, int senyal)**: determina si senyal pertenece a set.
- **int sigaddset(sigset_t *set, int signo)**: añade una señal a set (debe estar inicializado).
- **int sigdelset(sigset_t *set, int signo)**: elimina una señal signo de set.

sa_flags: flags que modifican el comportamiento del manejo de señales:

- **SA_NOCLDSTOP**: si signum es SIGCHLD, no desea recibir notificación cuando los procesos hijos se paren (reciben señales SIGTSTP, SIGTTIN o SIGTTOU).
- **SA_ONESHOT o SA_RESETHAND**: restaura la acción para la señal al estado predeterminado una vez que el manejador de señal haya sido llamado.
- **SA_RESTART**: hace que ciertas llamadas al sistema reinicien su ejecución cuando son interrumpidas por la recepción de una señal.
- **SA_NOMASK o SA_NODEFER**: no impide recibir la señal desde el propio manejador
- **SA_SIGINFO**: el manejador toma 3 argumentos, no uno. Hay que configurar sa_sigaction en lugar de sa_handler.

Ejemplo

```
struct sigaction sa;                          // crea la estructura sigaction
sa.sa_handler = SIG_IGN;                      // ignora la señal
sigemptyset(&sa.sa_mask);                     // inicializa un conjunto vacío de señales
sa.sa_flags = SA_RESTART;          // reinicia las funciones interrumpidas por un manejador
if (sigaction(SIGINT, &sa, NULL) == -1){    // especifica la señal SIGINT, con la estructura sa
        printf("error en el manejador"); }        // y sin guardar la estructura anterior
```

---

3. **sigprocmask**: se emplea para cambiar la lista de señales bloqueadas actualmente.
   int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);

Argumentos

how: indica el tipo de cambio. Los valores que puede tomar son:

- **SIG_BLOCK**: El conjunto de bloqueadas es la unión del conjunto actual y el set.
- **SIG_UNBLOCK**: Las señales que hay en set se eliminan del conjunto actual de señales bloqueadas. Se puede intentar desbloquear una señal que no bloqueada.
- **SIG_SETMASK**: El conjunto de bloqueadas se pone según el argumento set.

set: puntero al nuevo conjunto de señales enmascaradas.

- **set != NULL**: apunta a un conjunto de señales.
- **set == NULL**: se usa sigprocmask para consulta.

oldset: es el conjunto anterior de señales enmascaradas.

- **oldset != NULL**: se guarda el valor anterior de la máscara de señal en oldset.

Return

Devuelve 0 si ha ido bien o -1 si ha ocurrido un error.

---

4. **sigpending**: permite el examen de señales examinadas y/o pendientes de entrega (las que se han producido mientras estaban bloqueadas).

int sigpending(sigset_t *set);

Argumentos

set: puntero al conjunto de señales pendientes.

Return

Devuelve 0 si ha ido bien o -1 si ha ocurrido un error.

---

5. **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.

int sigsuspend(const sigset_t *mask);

Argumentos

mask: puntero al nuevo conjunto de señales enmascaradas.

Return

Devuelve -1 si sigsuspend es interrumpida por una señal capturada.

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);
```

---

6. **signal**: igual que dup pero en lugar de usar el número más bajo de fd usa el newfd para duplicarlo. No hay que usar close antes, lo hace dup2 automáticamente.

sighandler_t signal(int signum, sighandler_t handler);

Argumentos

signum: especifica la señal y puede ser cualquier señal válida salvo SIGKILL o SIGSTOP.
handler: función que vamos a enviar. Debe tener como parámetro de la función un int.

Return

Devuelve el el valor anterior del signal handler si ha ido bien o SIG_ERR si ha ocurrido un error.

Sesión 6

La función fcntl

1. **fcntl**: (file control) permite consultar o ajustar las banderas de control de acceso de un archivo abierto. Además, permite bloquear el archivo para un uso exclusivo.

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

Argumentos

fd: file descriptor del archivo.
orden: operación que se ejecuta.
argumento_orden: según orden necesite parámetros o no.

Return

Devuelve -1 si ha ocurrido un error o un valor dependiendo de la orden.

Órdenes

F_GETFL: retorna las banderas de control de acceso asociadas al descriptor de archivo.
F_SETFL (int): ajusta o limpia las banderas de acceso que se especifican como argumento.
F_GETFD: devuelve la bandera close-on-exec (si está activa en un descriptor, al ejecutar exec, el hijo no heredará ese descriptor) 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 está desactivada por defecto.
F_SETFD(int): activa o desactiva la bandera close-on-exec del descriptor especificado. En este caso, el tercer argumento de la función es 0 para limpiar la bandera, y 1 para activarlo.
F_DUPFD (int): duplica el descriptor de archivo especificado por fd en otro descriptor. El argumento especifica que el descriptor duplicado debe ser mayor o igual que él. Devuelve el descriptor de archivo duplicado **(nuevoFD = fcntl(viejoFD, F_DUPFD, inicialFD).**
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.

Banderas de estado de un archivo abierto

La llamada fcntl permite recuperar/modificar el modo de acceso y las banderas de estado de un archivo abierto usando **F_GETFL**.
Con **O_SYNC** podemos comprobar si el archivo fue abierto para escrituras sincronizadas.
Ya que para comprobar las constantes O_RDONLY (0), O_WRONLY (1) y O_RDWR (2) del modo de acceso no podemos hacerlo con un único bit, usamos la máscara **O_ACCMODE**.

```
int banderas, ModoAcceso;
banderas = fcntl(fd, F_GETFL);
if (banderas == -1)                    //Si ha ocurrido algún error
        perror("fcntl error");
if (banderas & O_SYNC)                 //Si lo han abierto para escrituras sincronizadas
        printf ("Las escrituras son sincronizadas \n");
ModoAcceso = banderas & O_ACCMODE;  //Usamos máscara
if (ModoAcceso == O_WRONLY || ModoAcceso == O_RDWR)
        printf ("El archivo permite la escritura \n);
```

F_SETFL permite modificar algunas banderas de estado, son: **O_APPEND**, **O_NONBLOCK**, **O_NOATIME**, **O_ASYNC** y **O_DIRECT**. Si intentamos modificar cualquier otra, se ignora. Para modificar las banderas hacemos lo siguiente:

```
int bandera;
bandera = fcntl(fd, F_GETFL);                  //Obtenemos la bandera
if (bandera == -1)
        perror("fcntl");
bandera |= O_APPEND;                           //Habilitamos bandera O_APPEND
if (fcntl(fd, F_SETFL, bandera) == -1)         //Modificación invocando a fcntl
        perror("fcntl");
```

Duplicar descriptores de archivos con fcntl

F_DUPFD permite duplicar un descriptor (si tiene éxito tendremos a dos descriptores que apuntan al mismo archivo abierto compartiendo la misma sesión de trabajo).

```
int fd = open ("temporal", O_WRONLY);      //Abre el archivo temporal
//Cerramos la salida estándar para asegurar que donde se va a duplicar está libre
close (1);
if (fcntl(fd, F_DUPFD, 1) == -1 )      //Realiza duplicación en el descriptor 1
        perror ("Fallo en fcntl");
char bufer[256];
//Escribe en el archivo temporal ya que el descriptor apunta al archivo abierto
int cont = write (1, bufer, 256);
```

Bloqueo de archivos con fcntl

Para utilizar fcntl en el bloqueo de archivos, debemos usar una estructura flock que define el cerrojo que deseamos adquirir o liberar. Se define como:

```
struct flock {
    short l_type;       // Tipo de cerrojo
    short l_whence;     //Interpretar l_start
    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)
};
```

Parámetros del Struct flock

l_type: indica el tipo de bloqueo que queremos usar, puede tomar los valores: **F_RDLCK** (cerrojo de lectura), **F_WRLCK** (cerrojo de escritura) y **F_UNLCK** (elimina el cerrojo).

l_whence: establece desde donde empieza a contar el desplazamiento l_start:

- **SEEK_SET**: empieza desde el principio.
- **SEEK_CUR**: el actual file offset.
- **SEEK_END**: empieza desde el final.

l_start: desde donde empieza el offset tomando como referencia el valor de l_whence. Si l_whence es SEEK_CUR o SEEK_END, el valor de l_start puede ser negativo.

l_len: indica el número de bytes bloqueados:

- **l_len > 0**: cubre un intervalo de l_len bytes desde l_start → **[l_start, l_start+l_len-1]**
- **l_len < 0**: el intervalo cubre **[l_start - abs(l_len), l_start-1]**
- **l_len = 0**: bloquea todos los bytes desde la posición dada por l_whence y l_start

Ejemplo

```
struct flock cerrojo;
int fd = open(*++argv, O_RDWR)
cerrojo.l_type =F_WRLCK;
cerrojo.l_whence =SEEK_SET;
cerrojo.l_start =0;
cerrojo.l_len =0;

while (fcntl(fd, F_SETLK, &cerrojo) == -1) {  /*si el cerrojo falla, vemos quien lo bloquea*/
        while(fcntl(fd, F_GETLK, &cerrojo) != -1 && cerrojo.l_type != F_UNLCK ) {
                printf("%s bloqueado por %d desde %d hasta %d para %c", *argv,
                cerrojo.l_pid, cerrojo.l_start, cerrojo.l_len,
                cerrojo.l_type==F_WRLCK ? 'w':'r'));
                if (!cerrojo.l_len) break;
                cerrojo.l_start +=cerrojo.l_len;
                cerrojo.l_len=0;
        } /*mientras existan cerrojos de otros procesos*/
} /*mientras el bloqueo no tenga exito */
/* Ahora el bloqueo tiene exito y podemos procesar el archivo */

/* Una vez finalizado el trabajo, desbloqueamos el archivo entero */
cerrojo.l_type =F_UNLCK;
cerrojo.l_whence =SEEK_SET;
cerrojo.l_start =0;
cerrojo.l_len =0;
if (fcntl(fd, F_SETLKW, &cerrojo) == -1) perror("Desbloqueo");
```

El archivo /proc/locks

En este archivo podemos ver los cerrojos que están actualmente en uso, aparecen los cerrojos creados tanto por fcntl como por flock. Ejemplo de su contenido:

Nº Cerrojo	Tipo1	Modo	Tipo2	PID	ID archivo	Ini. bloq	Fin bloq
1:	POSIX	ADVISORY	READ	8581	08:08:2100091	128	128
2:	POSIX	ADVISORY	WRITE	8581	08:08:2097524	0	EOF

…

- **Tipo1:** POSIX indica que se creó con fcntl() y FLOCK que se creó con flock().
- **Modo:** puede ser ADVISORY(consultivo) o MANDATORY (obligatorio).
- **Tipo2:** puede ser WRITE o READ.
- **Identificador del archivo:** tres números separados por ":", el número principal y secundario del dispositivo donde reside el sistema de archivos y el número de i-nodo
- **Ini. bloq:** byte de inicio del bloqueo. Para flock() siempre es 0.
- **Fin bloq:** byte final del bloqueo, EOF indica el final del archivo. Para flock() es EOF.

Archivos proyectados en memoria

Un archivo proyectado en memoria es una técnica de los SO para acceder a archivos. Crea una nueva región de memoria en el espacio de direcciones del proceso del tamaño de la zona a acceder del archivo (una parte o todo el archivo) y cargar en ella el contenido de esa parte del archivo.

1. **mmap**: proyecta bien un archivo bien un objeto memoria compartida en el espacio de direcciones del proceso. Opera sobre páginas, el área proyectada es múltiplo del tamaño de página.

void *mmap(void *address, size_t length, int prot, int flags, int fd, off_t offset);

Argumentos

address: especifica la dirección de inicio dentro del proceso donde debe mapearse el descriptor. Si especificamos un nulo, le indica al kernel que elija la dirección de inicio.
length: número de bytes a proyectar.

- **addres** y **length** deben ser múltiplos del tamaño de página para que estén alineados al límite de páginas, si no la proyección se redondeará por exceso hasta completar.

prot: tipo de protección de la proyección, usamos las siguientes constantes:

- **PROT_READ**: los datos se pueden leer.
- **PROT_WRITE**: los datos se pueden escribir.
- **PROT_EXEC**: los datos se pueden ejecutar.
- **PROT_NONE**: no podemos acceder a los datos.

flags: debemos indicar el indicador MAP_SHARED o MAP_PRIVATE, estos son:

- **MAP_SHARED**: los cambios son compartidos. Los dos usos principales son realizar entradas/salidas proyectadas en memoria ocompartir memoria entre procesos.
- **MAP_PRIVATE**: los cambios son privados. Puede usarse para que múltiples procesos compartan el código/datos de un ejecutable o biblioteca, de forma que las modificaciones que realicen no se guarden en el archivo.
- **MAP_FIXED**: hace que la dirección addres sea un requisito, no un consejo. No debería especificarse por razones de portabilidad.
- **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
- **MAP_UNITIALIZED**: no limpia(poner a cero) las proyecciones anónimas

fd: el descriptor del archivo a proyectar. Una vez creada la proyección, lo podemos cerrar.
offset: desplazamiento desde el inicio del archivo desde el que se empiezan a contar los byes a proyectar (length). Suele ser 0.

Return

Devuelve dirección inicial de la proyección si ha ido bien o MAP_FAILED en caso de error.

Ejemplo

```
const int MMAPSIZE=1024;
char *memoria;
fd = open("Archivo", O_RDWR|O_CREAT|O_EXCL, S_IRWXU);
memoria = (char *)mmap(0, MMAPSIZE, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
```

---

2. munmap: elimina una protección del espacio de un proceso. Una vez desmapeada, cualquier referencia a la proyección generará la señal SIGSEGV. Si una región se declaró MAP_PRIVATE, los cambios que se realizaron en ella son descartados.

int munmap(void *address, size_t length);

Argumentos

address: el valor que devuelve mmap.
length: tamaño de la región mapeada.

Return

Devuelve 0 si ha ido bien o -1 en caso de error.

Otras funciones relacionadas con la protección de archivos son:

- **mremap()**: se utiliza para extender una proyección existente.
- **mprotect()**: cambia la protección de una proyección.
- **madvise()**: establece consejos sobre el uso de memoria, es decir, como manejar las entradas/salidas de páginas de una proyección.
- **remap_file_pages()**: permite crear mapeos no-lineales, es decir, mapeos donde las páginas del archivo aparecen en un orden diferente dentro de la memoria contigua.
- **mlock()**: permite bloquear (anclar) páginas en memoria.
- **mincore()**: informa sobre las páginas que están actualmente en RAM

Compartición de memoria

Una forma de compartir memoria entre un proceso padre y un hijo es invocar mmap() con MAP_SHARED en el padre antes de invocar a fork().
La compartición de memoria no está restringida a procesos emparentados, cualesquiera procesos que mapeen la misma región de un archivo, comparten las mismas páginas de memoria física.

Proyecciones anónimas

Una proyección anónima es similar una proyección de archivo salvo que no existe el correspondiente archivo de respaldo. En su lugar, las páginas de la proyección son inicializadas a cero.
Una de las utilidades de las proyecciones anónimas es que varios procesos emparentados compartan memoria. También se puede utilizar como mecanismo de reserva y asignación de memoria en un proceso.

Proyección anónima con MAP_ANON

```
char *p;
p = (char *)mmap (0, sizeof(char), PROT_READ , MAP_SHARED |MAP_ANON, -1, 0);
```

Proyección anónima con /dev/zero.

```
int fd;
char *p;
fd = open("/dev/zero", O_RDONLY);
p = (char *) mmap (0, sizeof(int), PROT_READ , MAP_SHARED, fd, 0);
```

Tamaño de la proyección y del archivo proyectado

Normalmente, el tamaño del mapeo es múltiplo del tamaño de página, y cae completamente dentro de los límites del archivo proyectado. Sin embargo, no es necesariamente así, si el tamaño de la proyección no es múltiplo del tamaño de página, ésta se redondea por exceso. Estos bytes de redondeo son accesibles pero no se mapean en el archivo de respaldo, y se inicializan a 0.

Para conocer el tamaño del archivo origen podemos utilizar **stat**() y para establecer el tamaño del archivo destino se puede usar **ftruncate**().