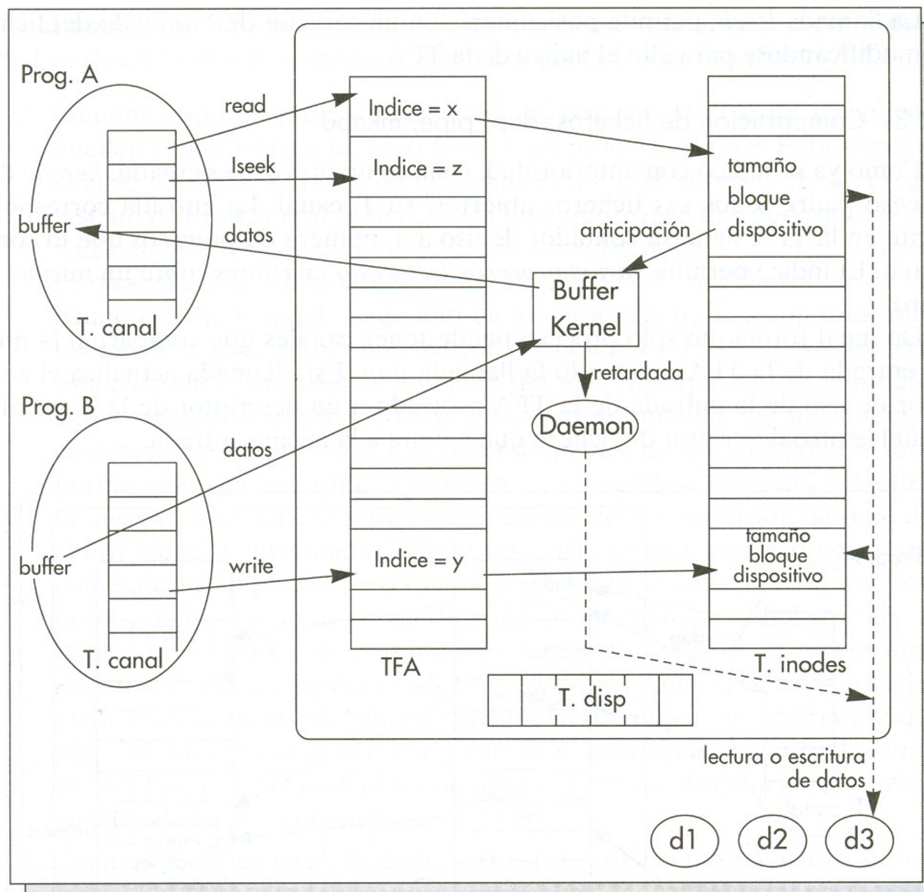

FIGURA 5.3. Llamadas read, write y lseek

a) Escritura *síncrona.* La escritura sobre el dispositivo empieza en el mismo momento de hacer la llamada *write*, cediéndose control al proceso una vez finalizada la escritura física.

b) Escritura *asíncrona.* La escritura sobre el dispositivo empieza en el mismo momento de hacer la llamada, no esperando a finalizar la escritura física para ceder el control al proceso.

c) Escritura *retardada, delayed.* El proceso no espera ni a que empiece la escritura física. UNIX actualiza en el buffer del kernel la información. Es responsabilidad de un proceso del sistema el escribir físicamente los datos actualizados en el buffer; regularmente este proceso se despierta para realizar la actualización. Esta forma de escritura, aunque más eficiente, está sujeta al riesgo de perder información de un fichero en caso de abortar el programa o de *crash* del UNIX.

La llamada *lseek*, permite posicionarse en un carácter determinado del fichero, modificándose para ello el índice de la TFA.

5.2.3. Compartición de ficheros: dup, pipe, mknod

Como ya se indicó con anterioridad, cuando un proceso es creado, *hereda* del proceso padre todos sus ficheros abiertos, su T. canal. La entrada correspondiente en la TFA tiene su contador de uso a *n*, número de procesos que lo comparten. El índice permite *varios procesos lectores o escritores* sobre un mismo fichero.

De igual forma, un solo proceso puede tener canales que compartan la misma entrada de la TFA utilizando la llamada *dup*. Esta llamada actualiza el contador de uso de la entrada de la TFA asociado a un descriptor de la T. canal y devuelve otro descriptor de fichero que apunta a la misma entrada.

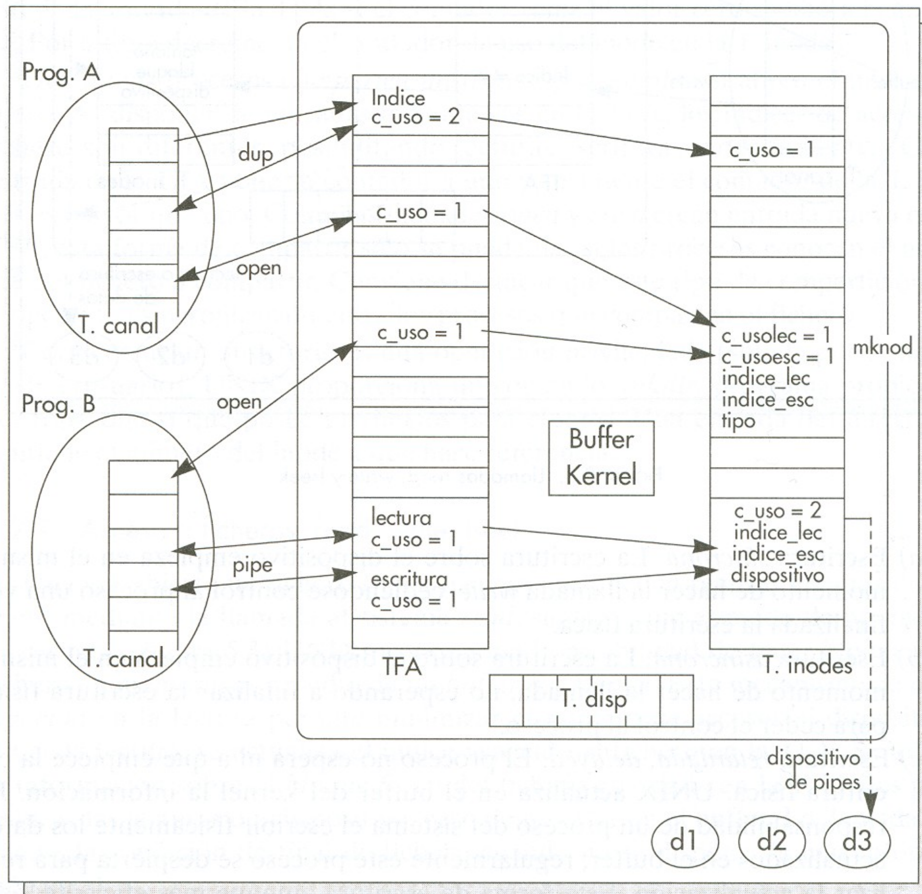

FIGURA 5.4. Llamadas dup, pipe y mknod

Directiva	Llamada
#include <errno.h>	todas
#include <fcntl.h>	todas
#include <signal.h>	*signal*
#include <stdio.h>	todas
#include <sys/ipc.h>	*msgctl, msgget, msgsnd, msgrcv, semctl, semget, semop, shmctl, shmget, shmop*
#include <sys/locking.h>	*lockf, plock*
#include <sys/msg.h>	*msgctl, msgget, msgsnd, msgrcv, semctl, semget, semop, shmctl, shmget, shmop*
#include <sys/stat.h>	*mknod, stat, fstat*
#include <sys/times.h>	*times*
#include <sys/types.h>	*mknod, stat, fstat, times, ustat, msgctl, msgget, msgsnd, msgrcv, semctl, semget, semop, shmctl, shmget, shmop*
#include <sys/utsname.h>	*uname*
#include <termio.h>	*ioctl*
#include <ustat.h>	*ustat*

Figura 5.15. Definiciones que se deben incluir

5.5.2. Llamadas básicas de E/S: creat, open, close, read, write

• *Llamada creat*

Prototipo: `int creat (char *path, int perm)`

Función: La llamada *creat* sirve para crear un nuevo fichero. Si el usuario tiene permiso de escritura, se crea un nuevo *inode* y se le asigna un apuntador en el directorio en que se crea. Si el fichero ya existía se mantiene su información de control, pero se pierden los datos que tenía (su longitud se pone a 0). *perm* especifica los permisos de acceso para este nuevo fichero: 3 grupos (usuario: *o*, grupo: *g* y mundo: *w*) de 3 bits (lectura: *r*, escritura: *w* y ejecución: *x*)

Valores que devuelve:
- el descriptor del fichero abierto para escritura.

• *Llamada open*

Prototipo: `int open (char *path, int flags, int perm)`

Función: La forma más extendida de *open* permite abrir un fichero existente o crear uno inexistente (lo que equivale a *creat*). El segundo parámetro especifica los flags. Su significado se puede ver en la figura 5.16.

O_RDONLY	Sólo lectura
O_WRONLY	Sólo escritura
O_RDWR	Lectura y escritura
O_NDELAY	Sincronización para FIFOs
O_CREAT	Crear si no existe
O_TRUNC	Borrar si existe
O_EXCL	Devolver error si existe (combinado con O_CREAT)
O_APPEND	Escribir en modo adición

Estos flags pueden combinarse usando los operadores lógicos de C | (or), & (and) y ~ (not). El valor O_WRONLY|O_CREAT|O_TRUNC en la llamada *open* hace que se efectúe el equivalente a la llamada *creat*. El flag O_EXCL permite el uso de ficheros como semáforos de exclusión mutua.

Figura 5.16. Flags usados por open

• *Llamada close*

Prototipo: `int close (int fd)`

Función: *close* cierra el fichero cuyo descriptor se pasa como argumen liberando su descriptor y salvando anteriormente los datos q estuvieran sólo modificados en memoria. Hay que tener cuenta que la llamada *exit* cierra todos los ficheros que ter abierto el proceso que acaba.

• *Llamadas read y write*

Prototipos: `int read (int fd, char *buf, unsigned lon)`
`int write (int fd, char *buf, unsigned lon)`

Función: Lectura y escritura en un dispositivo o fichero, generalmente través del buffer kernel. Se pasan como argumentos el descript de fichero, el buffer destino para lectura o fuente para escritura la longitud a leer o escribir

Valores que devuelve:
- un valor entero equivalente al número de bytes leídos o esc tos (que puede no coincidir con la longitud indicada).
- Si es 0 en lectura indica el fin de fichero.
- Si el valor devuelto es -1 es que se ha producido un error

En las figuras 5.17 y 5.18 pueden verse ejemplos de su uso.


```
#define BUFSIZE 512

main (argc, argv)   /* copiar.c */
      int argc;
      char *argv[];
{
      int n;
      char buf[BUFSIZE];

      /* ciclo de lectura y escritura */
      while ((n= read(0, buf, BUFSIZE)) > 0)
      if (write(1,buf,n) != n) error("write");

      if (n==-1) error("read");
}
```

Figura 5.17. Ejemplo de E/S. Programa copiar

• *Llamada lseek*

Prototipo: `long lseek (int fd, long despl, int cod_origen)`

Función: En UNIX la lectura/escritura siempre se hace sobre la posición en curso, que se inicializa en la *open* al principio del fichero en general, o al final si se especifica el flag O_APPEND; y se actualiza en cada lectura/escritura. La llamada *lseek* hace posible el posicionamiento en cualquier parte del fichero, con lo que combinada con *read/write* posibilita el acceso directo. Como argumentos se pasan el descriptor de fichero, la posición o desplazamiento y código de origen desde donde contar el desplazamiento (0: desplazamiento sobre posición inicial —posición absoluta—, 1: sobre posición actual y 2: sobre fin de fichero).

Valores que devuelve:
- la nueva posición.

Se puede ver un ejemplo de su utilización en la figura 5.18 a través del programa ejemplo de *posicionar* definido en el capítulo 4, que escribe los 10 últimos caracteres de un fichero (en este ejemplo y en los siguientes, el tratamiento de errores no es exhaustivo).


```
#include <stdio.h>
#include <fcntl.h>

main(argc, argv)    /* cola.c */
  int argc;
  char *argv[];
{
  int fd, l;
  char buf[80];

  if ((fd= open(argv[1], 0666, O_RDONLY)) == -1) {
    error("open");
    exit(1);
  }
  if (lseek(fd, -10L, 2) == -1) {
    error("lseek");
    exit(1)
  }
  l= read(fd, buf, 10);
  write(1, buf, l);
}
```

Figura 5.18. Ejemplo de lseek

5.5.3. Compartición de ficheros y estructura del directorio: dup, pipe, mknod, link, unlink, mount, umount

• *Llamada dup*

Prototipo: `int dup(int fd)`

Función: *Dup* duplica un descriptor de fichero creando un nuevo descriptor para el fichero o pipe. El nuevo descriptor es copia exacta del anterior. La llamada sólo falla si el original no está abierto o la tabla de descriptores está llena. UNIX garantiza que se asigna el descriptor de más bajo índice que no esté ocupado. Si antes de *dup* hacemos *close(0)* tenemos garantizado que el descriptor se duplica sobre la entrada estándar. Lo mismo se puede hacer con cualquier otro canal.

Valores que devuelve:
- nuevo descriptor

O_RDONLY
O_WRONLY
O_RDWR
O_TRUNC

```
numdescriptordefichero = open ("fich.c", O_WRONLY|O_CREAT|O_TRUNC, 0666);
close(numdescriptordefichero);

fd = creat("fich.c", 0666);

int read  (int fd, char *buf, unsigned lon)
int write (int fd, char *buf, unsigned lon)
```

```
rwx rwx rwx
110 110 110
 6   6   6

111 100 000
 7   4   0

00001000
00000010
00000001

00001011
```