

Sistemas Operativos

Gestión de la Entrada/Salida

[SGG]: Cap. 10 al
13

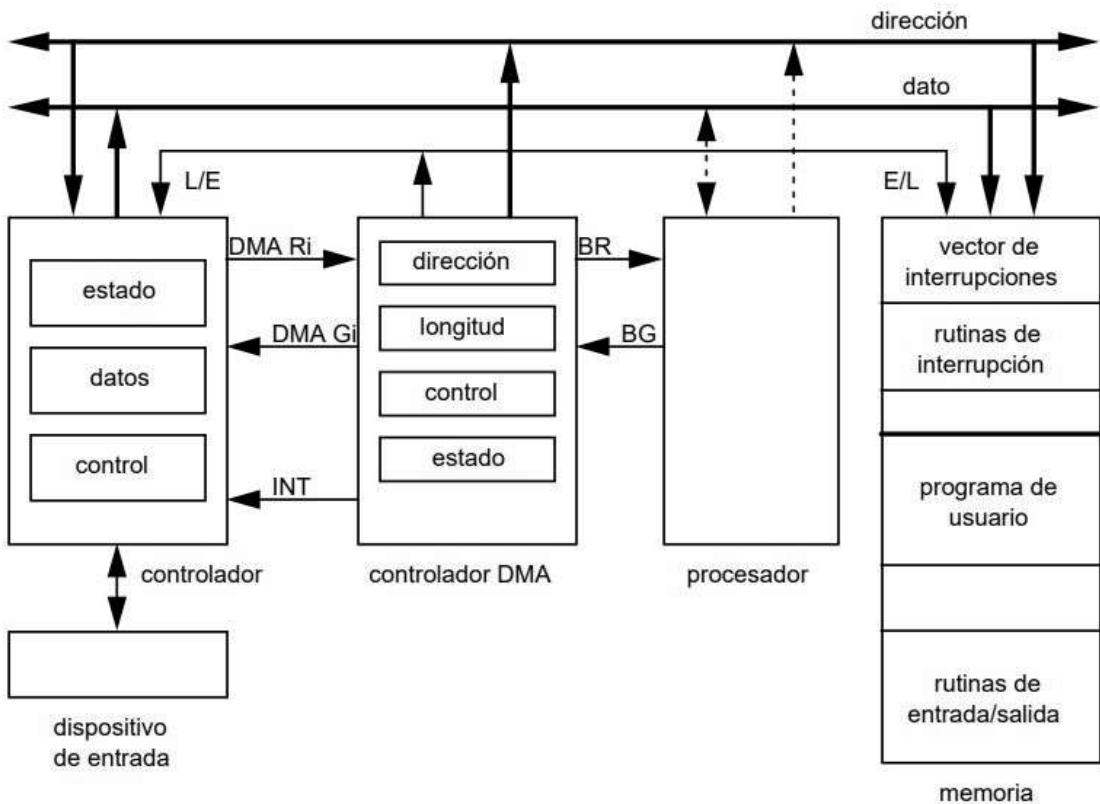
[Sta]: Cap 10 a 12

[Ste05]: Cap. 3 a 5

Gestión de la entrada/salida

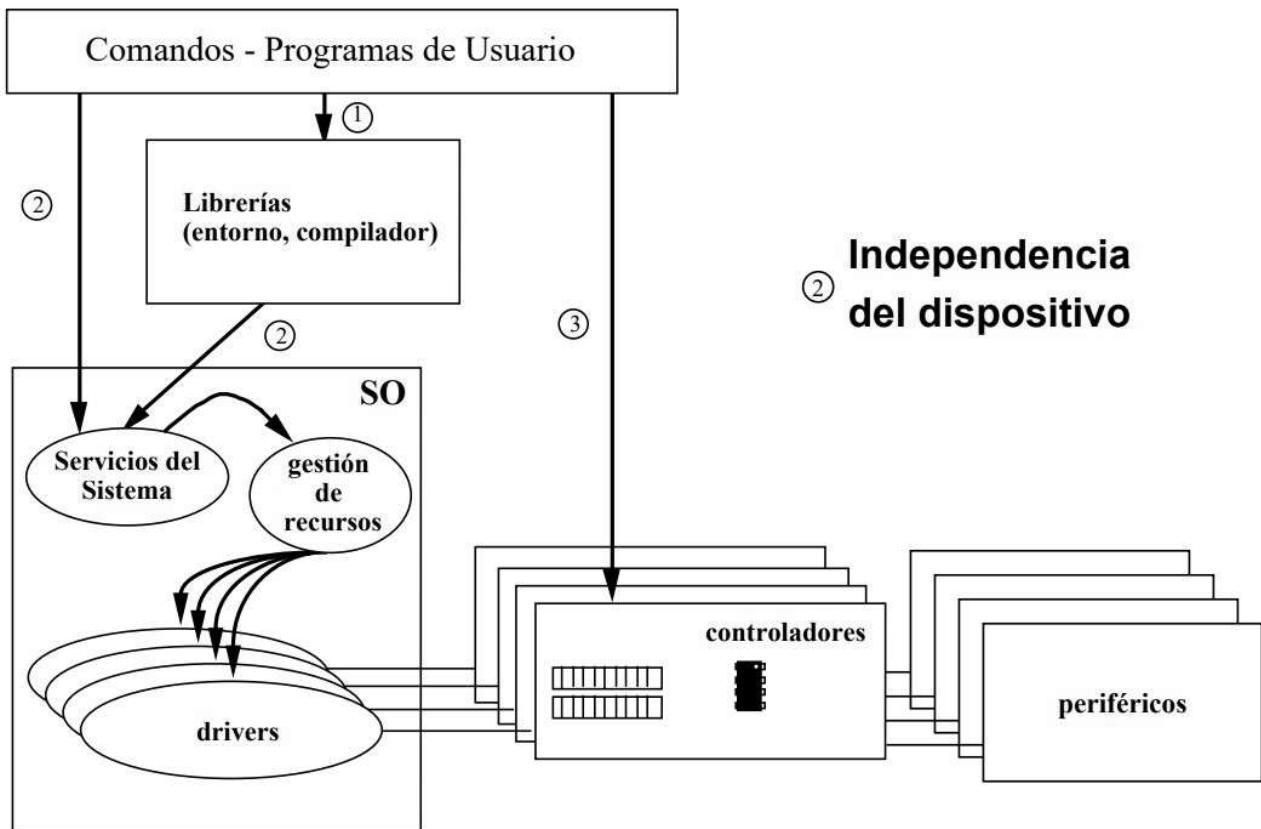
- *Hardware* de e/s. Recordatorio
- *Software* de la e/s. *Drivers*
- Tipos de dispositivos
- *Disco*: directorio, fichero, asignación de espacio
- Sistema de Ficheros UNIX
- UNIX: Estructura de un disco
- Tablas en memoria del Sistema de Ficheros

Fundamentos del *hardware* de e/s

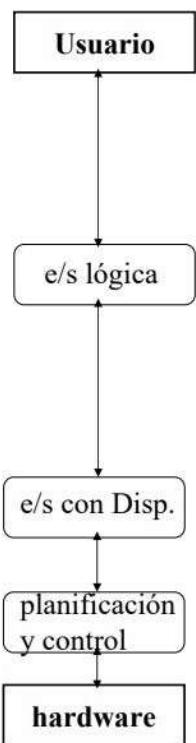


- **Sincronización por INTerrupciones**
- **Transferencia por Direct Memory Access**

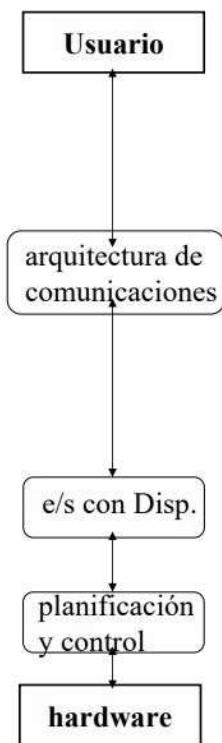
Análisis del software de la e/s



Software de la e/s (2 de 2)



Periférico local



Puerto de comunicaciones



Sistema de Archivos

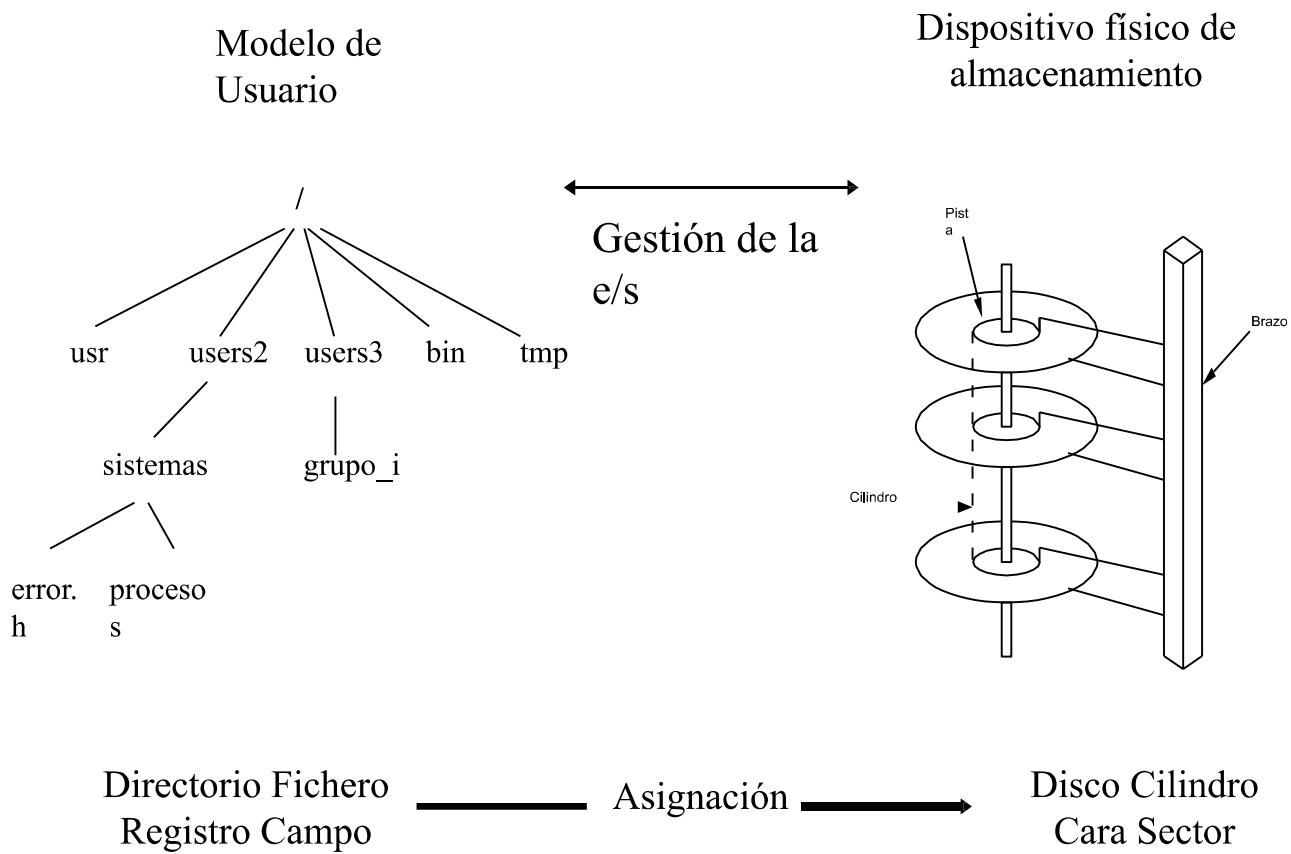
Drivers

- Dispositivos lógicos mantenidos por el SO
- *Software* que trabaja con la e/s dependiente del dispositivo
- Un *driver* por cada dispositivo o por una clase si están muy relacionados
 - UNIX: número mayor, número menor
- Emite los comandos a los registros del K_D
- Verifica que se ejecuten bien
- Conoce todos los detalles sobre el D y el K_D
 - @ de registros del K_D , utilidad de los registros, mecánica del D, sectores, pistas...

Tipos de Dispositivos

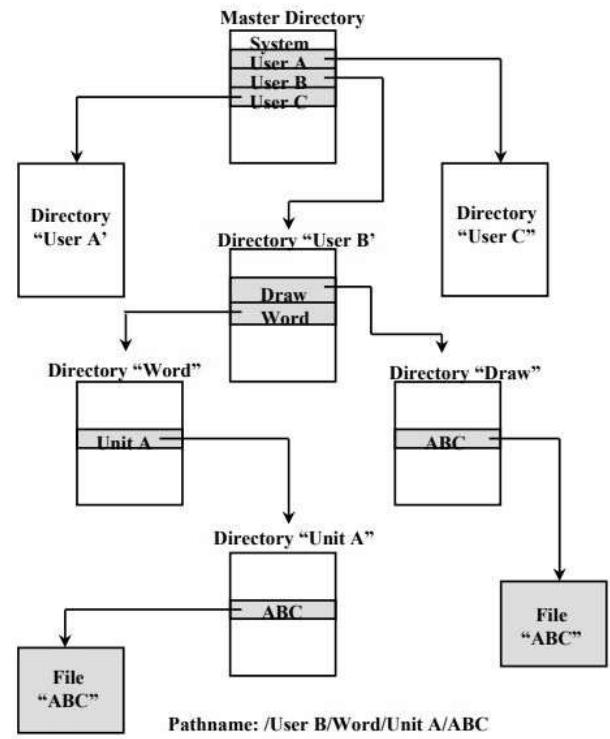
- De bloque. *Discos*
 - direccionable (bloque), con operaciones de localización
- De caracteres. *Terminales, impresoras, ratón...*
 - no direccionables, sin operaciones de localización

Abstracción del disco



Directorio: Espacio de nombres

- **Un nivel**
 - Todos los ficheros en un único directorio
- **Dos niveles**
 - Un directorio por usuario
- **Jerarquía**
 - Cada usuario puede organizar sus ficheros en subdirectorios



Fichero

- Unidad lógica de almacenamiento secundario
 - Tipo abstracto de dato def. e implementado por el S.O.
 - Colección de información relacionada
 - Para el usuario es la unidad mínima de almac. secund.
- Atributos :
 - nombre, identificador, tipo, propietario, protección, tamaño, fechas, localización, etc.
- Operaciones:
 - crear, borrar, leer, escribir, truncar, reposicionar
- Tipos de acceso a la información:
 - secuencial, directo, indexado, hash, ...

Fichero: operaciones

- Operaciones vs. Tipo de acceso
 - Secuencial:
 - rebobinar, leer_siguiente,
escribir_siguiente
 - Directo:
 - leer(n), escribir(n), posicionar(n)
 - leer_siguiente, escribir_siguiente
 - Indexado, Hash
 - Leer(clave), escribir(clave),
posicionar(clave)
 - leer_siguiente, escribir_siguiente

Asignación de espacio

- A cada fichero se le debe asignar espacio físico en un dispositivo
- El sistema debe guardar la información necesaria para recuperar el contenido del fichero
- El sistema debe controlar el espacio libre de cada dispositivo
- El espacio se asigna como una o varias porciones

Tamaño de las porciones

- Fijo/Variable
 - Variable minimiza el espacio perdido
 - Fijo simplifica la asignación de espacio
 - Se usa fijo: BLOQUE
- Grande/Pequeño
 - Grande minimiza tiempo de acceso
 - Grande minimiza información de acceso
 - Pequeño minimiza el espacio perdido

Métodos de Asignación

- Asignación contigua
- Asignación encadenada
- Asignación indexada

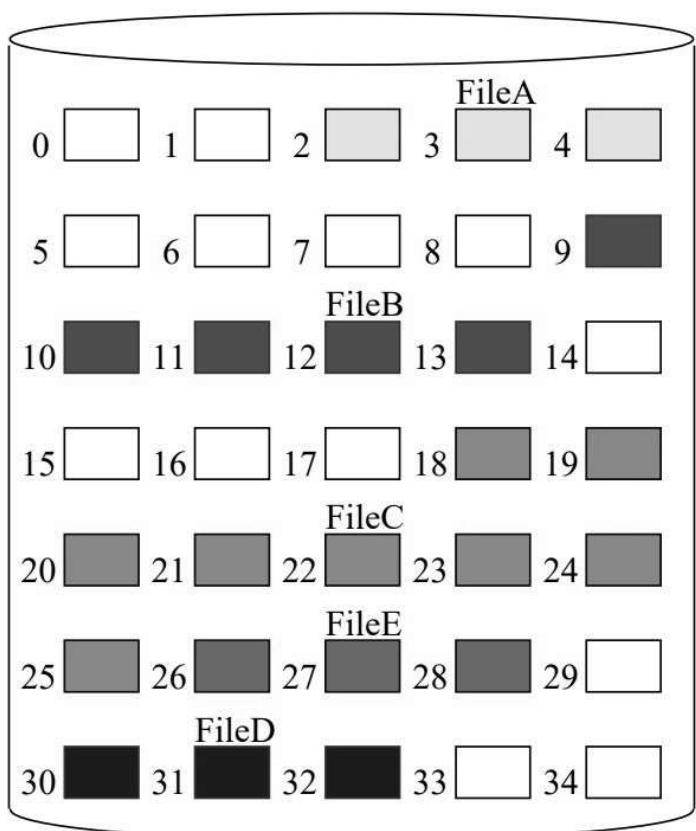


Tabla de localización

File Name	Start Block	Length
FileA	2	3
FileB	9	5
FileC	18	8
FileD	30	3
FileE	26	3

- Fácil localización: inicio y tamaño
- Fácil acceso secuencial y directo
- Difícil asignación
- Pérdida de espacio
- Problemas con cambios de tamaño

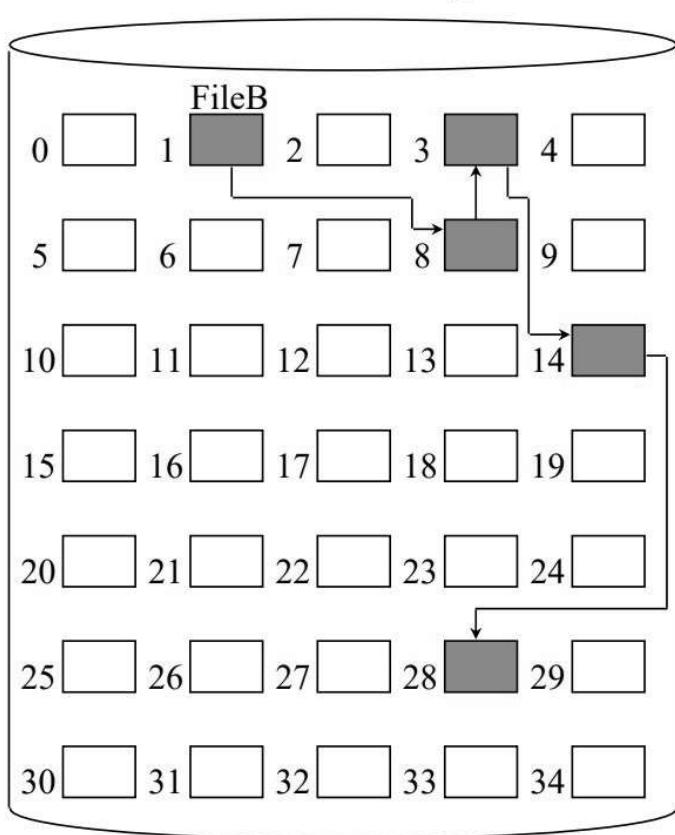


Tabla de localización

File Name	Start Block	Length
...
FileB	1	5
...

- Localización: inicio y tamaño
- Acceso secuencial
- Imposible acceso directo
- Fácil asignación
- Sin pérdida de espacio

La información se puede centralizar
Modelo DOS: FAT

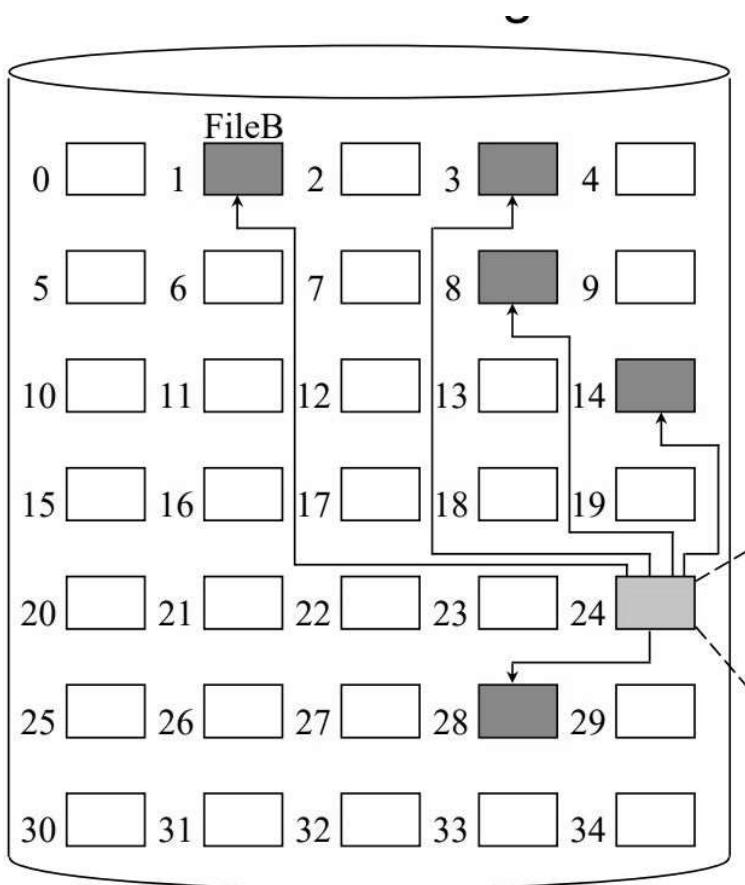


Tabla de localización

File Name	Index Block
...	...
FileB	24
...	...

- Localización: índice
- Acceso secuencial
- Acceso directo
- Fácil asignación
- Sin pérdida de espacio

Problema: tamaño del índice

Sistema de ficheros UNIX

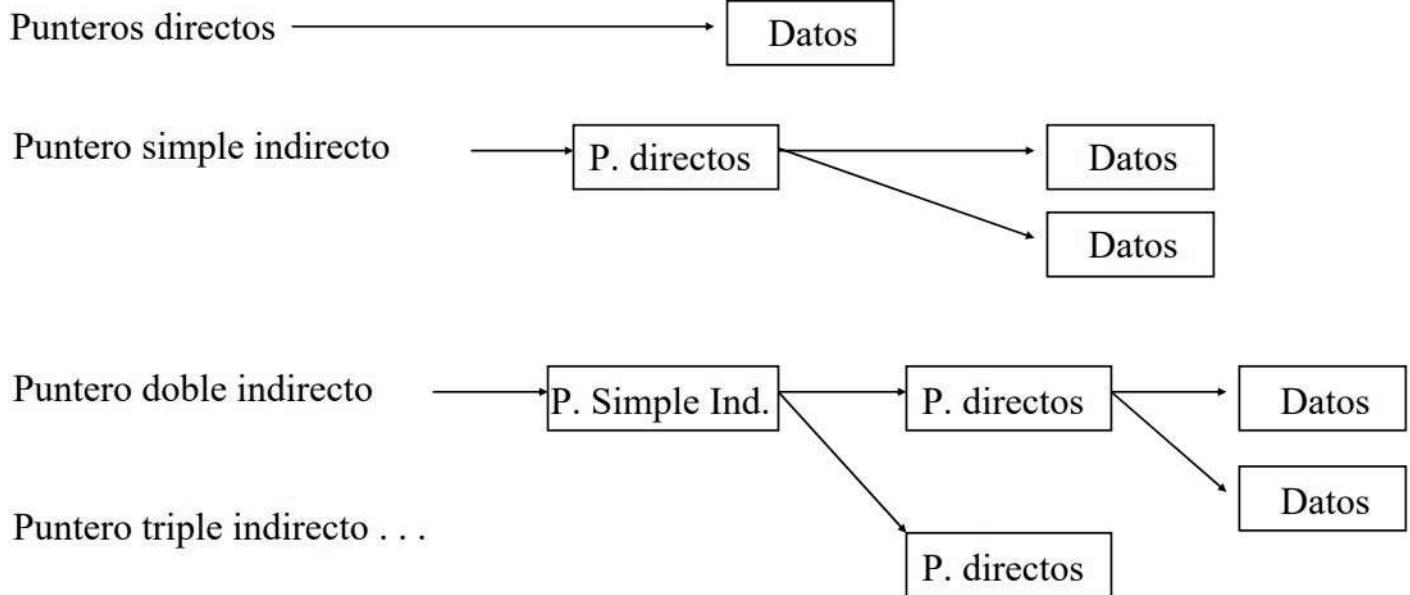
- Ficheros sin estructura
 - Son secuencias de bytes
- Tipos de ficheros
 - Ordinario: definido por el usuario
 - Directorio: contiene una lista de parejas nombres de fichero -- Inodo
 - Dispositivo: usado para el acceso a dispositivos de entrada/salida
 - FIFO, pipes o “named pipes”: Ficheros para comunicación entre procesos

UNIX: I-node

- Estructura de datos que almacena toda la información relativa a un fichero
 - Número de I-node
 - Tipo fichero
 - Protección
 - UID, GID
 - Tamaño
 - Fecha último cambio
 - Número de links
 - Información sobre localización de los bloques del fichero

UNIX: I-node, localización

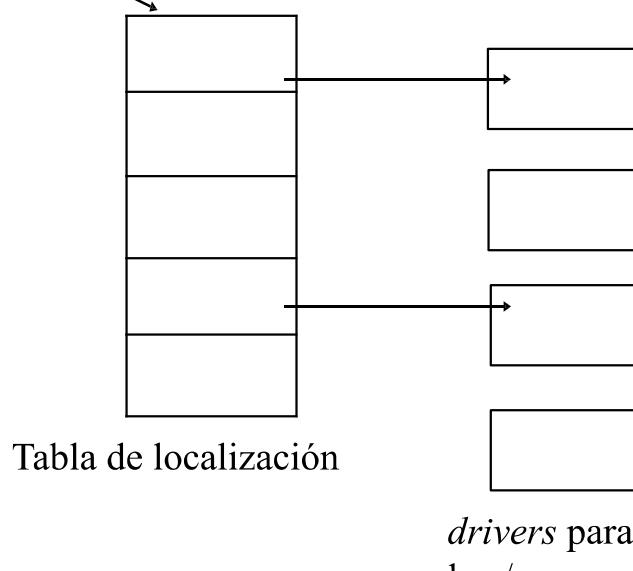
- Asignación indexada con varios niveles



UNIX: I-node, localización

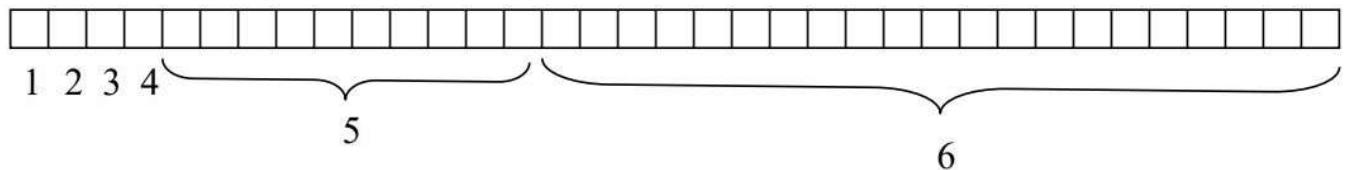
- Tipo de fichero
Dispositivo

número **Mayor** / número
menor



UNIX: Estructura de un disco

Ejemplo: MINIX



- 1. Bloque de arranque
- 2. Superbloque
- 3. Mapa de bits de nodos-I
- 4. Mapa de bits de bloques (o zonas)
- 5. Nodos-I
- 6. Bloques de datos

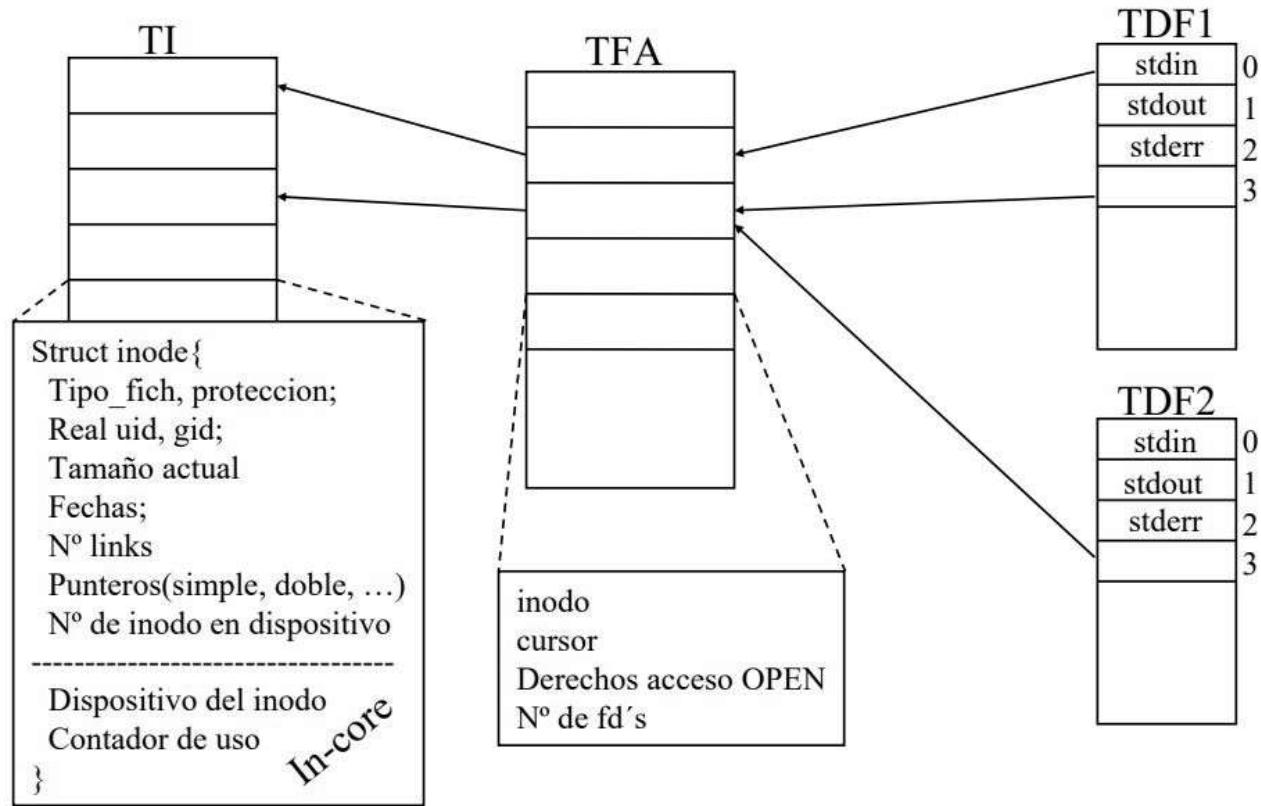
Ejercicio: asignación en UNIX

- Recuerda:
 - Estructura de directorio: árbol
 - Directorio: fichero con parejas <nombre, i-nodo>
 - Localización de la información de un fichero
 - I-nodo: índice con varios niveles
 - Estructura de un disco
 - Tabla de I-nodos y bloques de datos
- Ejercicio: ¿ Cuantos bloques de disco se han de leer para encontrar un byte de un fichero?
Por ej.
 - Byte: 450
 - Fichero: /users3/sistemas/error.h

Tablas en memoria del Sistema de Ficheros

- Guardan información de los ficheros en uso
 - Razones: eficiencia, implementación del cursor, tipos de acceso
- El usuario debe avisar al sistema que desea trabajar con un fichero: llamada open()
- El usuario debe avisar cuando deja de trabajar con un fichero: llamada close()
- Tablas en memoria: 2 tablas globales
 - I-nodos de todos los ficheros en uso (los abiertos) o los recientemente usados ...
 TI: tabla de I-nodos
 - Para el acceso secuencial se necesita un cursor. Si queremos que varios procesos trabajen al mismo tiempo con un fichero:
 - varios cursores, distintos permisos de acceso
 TFA: tabla de ficheros abiertos
- Tablas en memoria: 1 tabla por proceso que contiene punteros a las entradas de TFA que está usando
 TDF: Tabla de Descriptores de Fichero

Tablas en memoria del Sistema de Ficheros (2)



Sistemas Operativos

Ficheros: Llamadas al Sistema

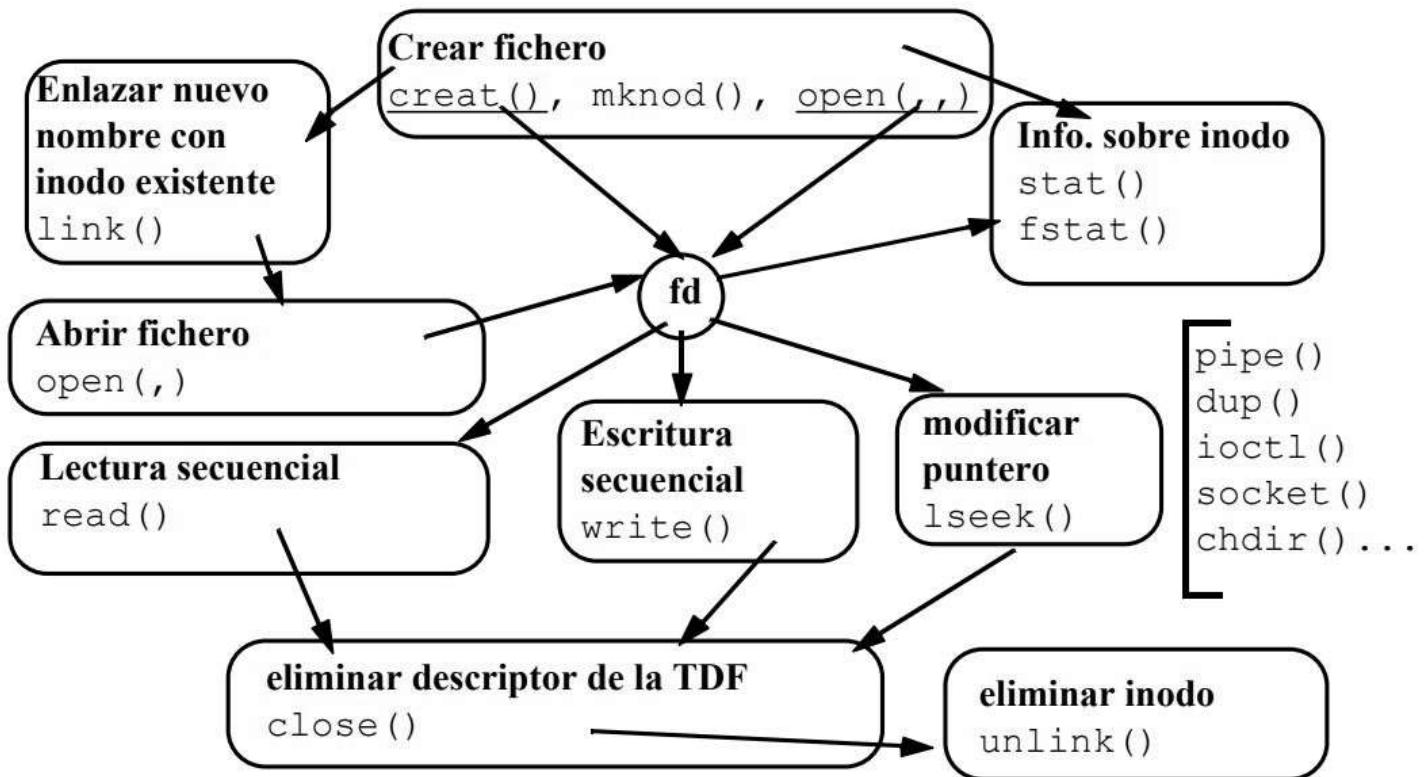


Ficheros: Llamadas al Sistema

- Esquema de llamadas sobre ficheros
- Manejo de ficheros existentes: creat(), open(), close(), read(), write(), lseek(). Ejemplos
- Llamada mknod()
- Información sobre ficheros: stat(), fstat()
- Llamadas link() y unlink()
- E/S standard vs. llamadas al sistema
- forkfiles.c

[Ste05]: cap. 3, 4, 5

Esquema de llamadas sobre ficheros



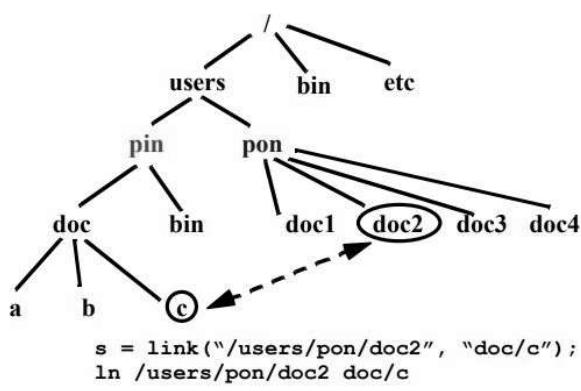
creat()

- **Sintaxis:** # include <fcntl.h>
int **creat** (char * *path*, mode_t *mode*)
- **Acción:** crea un fichero nuevo o reescribe uno existente
El fichero queda abierto sólo para escritura
path nombre del fichero a crear (absoluto o relativo)
Si *path* no existe: asigna/inicializa i-nodo
crea entrada en TFA (cursor al principio)
nueva entrada directorio <*path*,i-nodo>
Si *path* existe: trunca el fichero
mode permisos de acceso del fichero (según umask)
en octal!
- **Devuelve:** primer fd libre en TDF (apunta a TFA) ó –1 (errno)

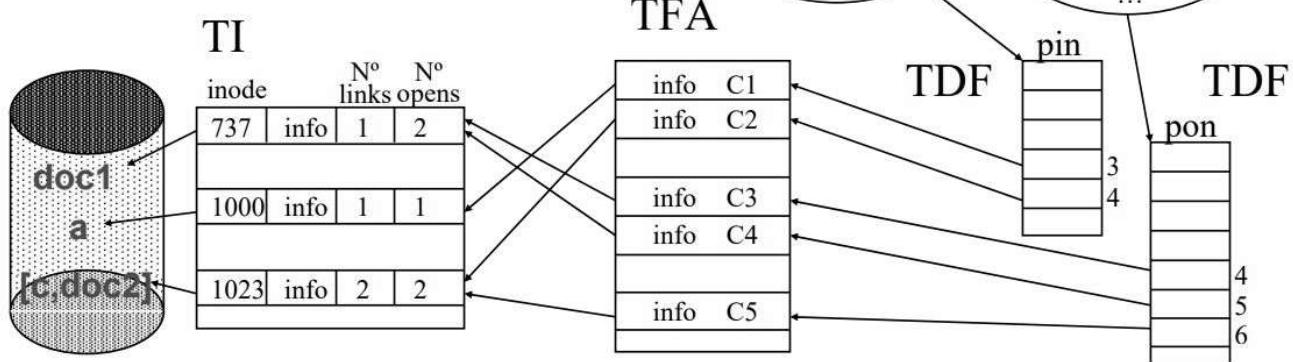
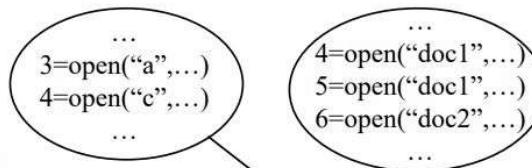
open()

- **Sintaxis:** # include <fcntl.h>
int **open** (char * *path*, int *oflag*)
 - **Acción:** abre un fichero existente
path nombre del fichero a abrir (absoluto o relativo)
oflag modo de apertura del fichero:
O_RDONLY, O_WRONLY, O_RDWR (0,1,2)
sino modo de apertura indefinido
+ OR con O_APPEND (cursor final antes de escribir), O_CREAT...
 - se identifica el i-nodo
 - se crea entrada en TFA
 - (cursor al principio, *oflag*)
 - open con 3 argumentos
 - **Devuelve:** primer fd libre en TDF (apunta a TFA) ó -1 (errno)
creat(pathname,mode) =
open(pathname,O_WRONLY | O_CREAT | O_TRUNC,mode)
Si queremos crear fichero pero poderlo leer también
open(pathname,O_RDWR | O_CREAT | O_TRUNC,mode)

Tablas en memoria del Sistema de Ficheros (3)



pin/doc		pon	
Nombre	i-nodo	Nombre	i-nodo
a	1000	doc1	737
b	1015	doc2	1023
c	1023	doc3	1090
		doc4	800



close()

- **Sintaxis:** # include <unistd.h>
int **close** (int *fildes*)
- **Acción:** cierra el descriptor de fichero *fildes*
 - quedá libre la entrada *fildes* de TDF
 - decrementa *nfildes* en TFA
 - si (*nfildes* == 0)
 - quedá libre entrada en TFA
 - decrementa *nopens* en TI
 - si (*nopens* == 0)
 - si (*nlinks* == 0) borra fichero;
 - elimina inodo de TI;
- **Devuelve:** 0 si bien ó –1 (errno)
- Cuando termina un proceso el kernel cierra todos los ficheros de su TDF y actualiza tablas TFA y TI

/* ej42.c */

```
#include <errno.h>
#include <stdio.h>
#include <fcntl.h>
#define DIM 300

main( argc, argv )
int argc; char *argv[];
{    int fd[DIM], contador, i;

    if( (fd[0] = creat( argv[1], 0777 )) == -1 ){
        perror( "creat" ); exit(1);
    printf( "Abro el %d\n", fd[0] );
    for( i = 1; ( fd[i] = open( argv[1], O_RDONLY )) != -1 ; i++ )
        printf( "Abro el %d\n", fd[i] );
    contador=i;
    if( errno == EMFILE )
        printf( "\tTotal fich. abiertos: %d\n", (contador+3) );
    else{ perror( "open" ); exit(1); }
    for( i=0; i<contador; ++i)
        close( fd[i] );
    unlink( argv[1] );
}
```

Crea tantos descriptores de fichero como permite el sistema
(OPEN_MAX=256)

EMFILE: agotada TDF
(ENFILE: agotada TFA)

Iseek()

- **Sintaxis:** # include <unistd.h>
off_t Iseek (int *fildes*, off_t *offset*, int *whence*)
- **Acción:** mueve el cursor de lectura/escritura de un fichero
se modifica la posición del cursor *offset* bytes
hacia delante o hacia detrás (según signo de *offset*)
desde un punto de referencia (*whence*)
whence posición desde la que se aplica el offset
SEEK_SET / L_SET / 0 principio fichero (*offset*>=0)
SEEK_CUR / L_CUR / 1 posición actual
SEEK_END / L_END / 2 posición final
- **Devuelve:** nueva posición del cursor respecto del principio
-1 si error (errno)
- **Ejemplo:** Para saber la posición actual del cursor:
off_t cursor;
cursor = Iseek(fd, 0, SEEK_CUR);

read()

- **Sintaxis:** # include <unistd.h>
 ssize_t **read** (int *fildes*, void **buf*, size_t *nbyte*)
- **Acción:** lee *nbyte* bytes del fichero asociado al descriptor *fildes* y los almacena en *buf*
 la lectura comienza en la posición indicada por el cursor (en TFA)
 se modifica la posición del cursor con el número de bytes realmente leídos
- **Devuelve:** número de bytes realmente leídos
 0 si ya no hay más bytes que leer
 -1 si error (errno)

write()

- **Sintaxis:** # include <unistd.h>
 `ssize_t write (int fildes, const void *buf, size_t nbyte)`
- **Acción:** escribe *nbyte* bytes del buffer *buf* en el fichero asociado al descriptor *fildes*
 la escritura comienza en la posición indicada por el cursor (en TFA)
 se modifica la posición del cursor con el número de bytes realmente escritos
 se actualiza el tamaño del fichero en i-nodo
- **Devuelve:** número de bytes realmente escritos
 –1 si error (errno)

/* ej10.c */

Copia el contenido de fich1 en fich2

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

main( argc, argv )
int argc; char *argv[];
{   int fdFnt, fDst;
    void copia();

    if ( argc != 3){
        printf( "Uso: %s fich1 fich2 ", argv[0] ); exit( 1 );
    }

    if ( (fdFnt = open( argv[1], O_RDONLY )) == -1 )
        { fprintf( stderr, "\tError open\n" ); exit( 1 ); }

    if ( (fDst = creat( argv[2], 0666 )) == -1 )
        { fprintf( stderr, "\tError creat\n" ); exit( 1 ); }

    copia( fdFnt, fDst );
    exit( 0 );
}
```

```
void copia ( fnt, dst )
int fnt, dst;
{   int cuenta;
    char buf[BUFSIZ];

    while ( (cuenta = read( fnt, buf, sizeof( buf ) )) > 0 )
        write( dst, buf, cuenta );
}
```

Probar:

- ej10 fich_existe fich_nuevo
- ej10 ej10.c nuevo.c
- ej10 ej10 ej10_nuevo
- ej10 .direct
- ej10 fich_existe /dev/tty

en <stdio.h>

/* reverse.c */

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

main(argc,argv)
int argc;    char *argv[];
{   char c;
    int i, fdfnt;
    long where;

    if(argc != 2){ printf( "Uso: %s fichero_a_invertir" argv[0]); exit(1); }

    if((fdfnt = open( argv[1], O_RDONLY )) == -1) syserr("open");
    if((where = lseek( fdfnt, -1L ,SEEK_END )) == -1 )      syserr("lseek");

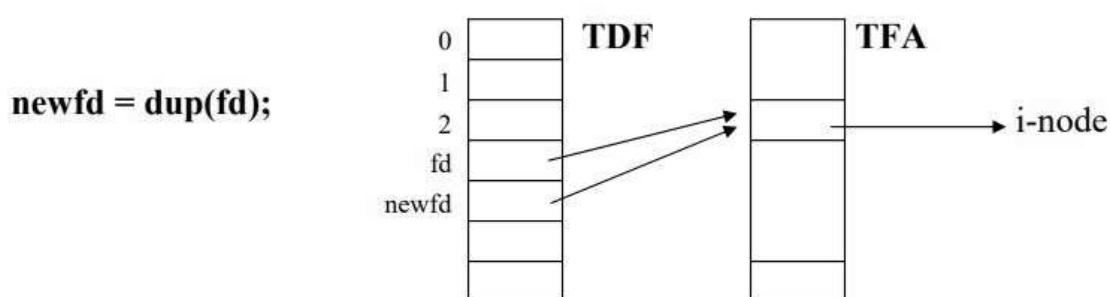
    while(where >= 0){
        read(fdfnt, &c, 1);
        write(1, &c, 1);
        where = lseek ( fdfnt, -2L ,SEEK_CUR );
    };
}
```

Invierte el contenido de un fichero

formato long

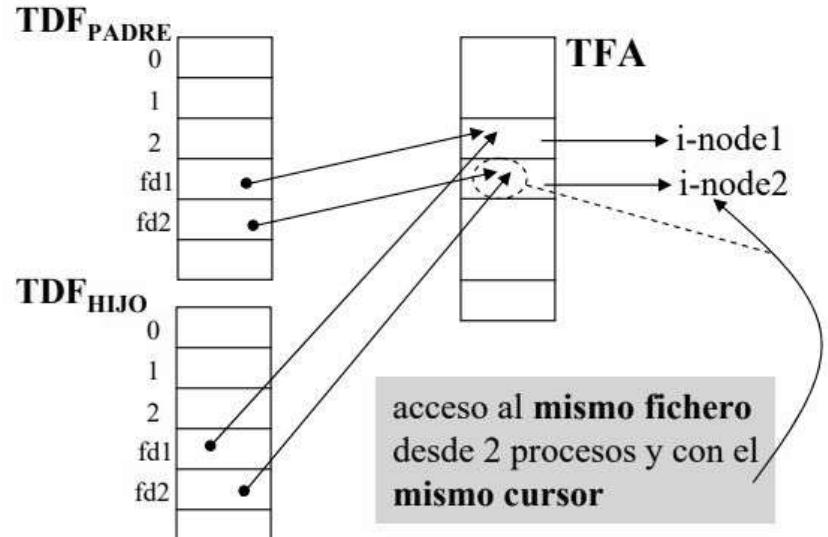
dup()

- **Sintaxis:** `#include <unistd.h>`
`int dup(int fd);`
- **Acción:** duplica *fd* con un nuevo descriptor (*newfd*)
fd es un descriptor ya asignado con open, creat, pipe, dup... => apunta a TFA
tras dup, *fd* y *newfd* apuntan a la misma entrada en TFA
- **Devuelve:** el siguiente descriptor libre en TDF ó –1 si error



Tablas vs fork() y exec()

- Tras hacer un **fork()** el proceso HIJO recibe una copia de la Tabla de Descriptores de Fichero del proceso PADRE:
=> - la TDF se *duplica* entera
- TODAS las entradas de la TDF_{HIJO} apuntan a la misma entrada en TFA que la correspondiente entrada de la TDF_{PADRE}



- Tras hacer un **exec()** el proceso *ejecutado* mantiene la TDF del proceso *ejecutor*

/* forkfiles.c */

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

int sfd, tfd;
char c;

main(argc, argv)
int argc; char **argv;
{
    if( argc != 3 ) exit( 1 );
    if( (sfd = open( argv[1], O_RDONLY ) ) == -1 ) syserr( "open" );
    if( (tfid = creat( argv[2], 0777 ) ) == -1 ) syserr( "creat" );

    if( fork() == -1) syserr( "fork" );
    copy();
    exit( 0 );
}

copy()
{
    for(;;){
        if( read( sfd, &c, 1) != 1) return;
        write( tfid, &c, 1 );
    }
}
```

Padre e Hijo acceden a los mismos
ficheros a través del cursor

stat(), fstat()

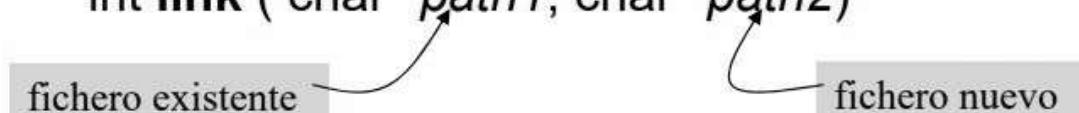
- **Sintaxis:** # include <sys/stat.h>
int **stat** (char **path*, struct stat **buf*)
int **fstat** (int *fd*, struct stat **buf*)
- **Acción:** recogen información del fichero asociado al nombre *path* (**stat**) o al descriptor *fd* (**fstat**)
se lee el i-nodo para llenar la estructura *buf*

fichero abierto

```
struct stat {  
    dev_t st_dev; /* ID of dev containing a directory entry for this file */  
    ino_t st_ino; /* Inode number */  
    mode_t st_mode; /* File type & Permission bits */  
    nlink_t st_nlink; /* Number of links */  
    uid_t st_uid; /* User ID of file owner */  
    gid_t st_gid; /* Group ID of file group */  
    dev_t st_rdev; /* major/minor IDs; defined only for char or blk spec files */  
    off_t st_size; /* File size (bytes) */  
    time_t st_atime; /* Time of last access */  
    time_t st_mtime; /* Last modification time */  
    time_t st_ctime; /* Last file status change time */  
    blksize_t st_blksize; /* best I/O block size (8192) */  
    ...  
};
```

- **Devuelve:** 0 si bien ó -1 (errno)

link()

- **Sintaxis:** # include <unistd.h>
int **link** (char **path1*, char **path2*)

- **Acción:** crea para *path2* una nueva entrada en el directorio con el mismo i-nodo de *path1*:

 $\langle path1, \text{i-nodo } i \rangle$ $\langle path2, \text{i-nodo } i \rangle$

 $(\text{número de links del i-nodo } i)++;$
- **Devuelve:** 0 si bien ó –1 (errno)

unlink()

- **Sintaxis:** # include <unistd.h>
int **unlink** (char **path*)
- **Acción:** elimina la entrada en el directorio del fichero *path*:
~~<*path*, i-nodo>~~
nlinks = nlinks-1 (en i-nodo);
si (nlinks == 0) {
 si (nopens (en TI) == 0)
 borra fichero;
 sino el borrado se hará en el último **close**;
}
- **Devuelve:** 0 si bien ó -1 (errno)

mknod()

- **Sintaxis:** # include <sys/stat.h>
 int **mknod** (char * *path*, mode_t *mode*, dev_t *dev*)

- **Acción:** crea tipos de ficheros especiales
(directorios, dispositivos, FIFOs...)
uso restringido al super-usuario excepto en FIFOs

path nombre del fichero a crear

mode tipo y permisos de acceso del fichero (+ umask)
constantes simbólicas definidas en *stat.h*:

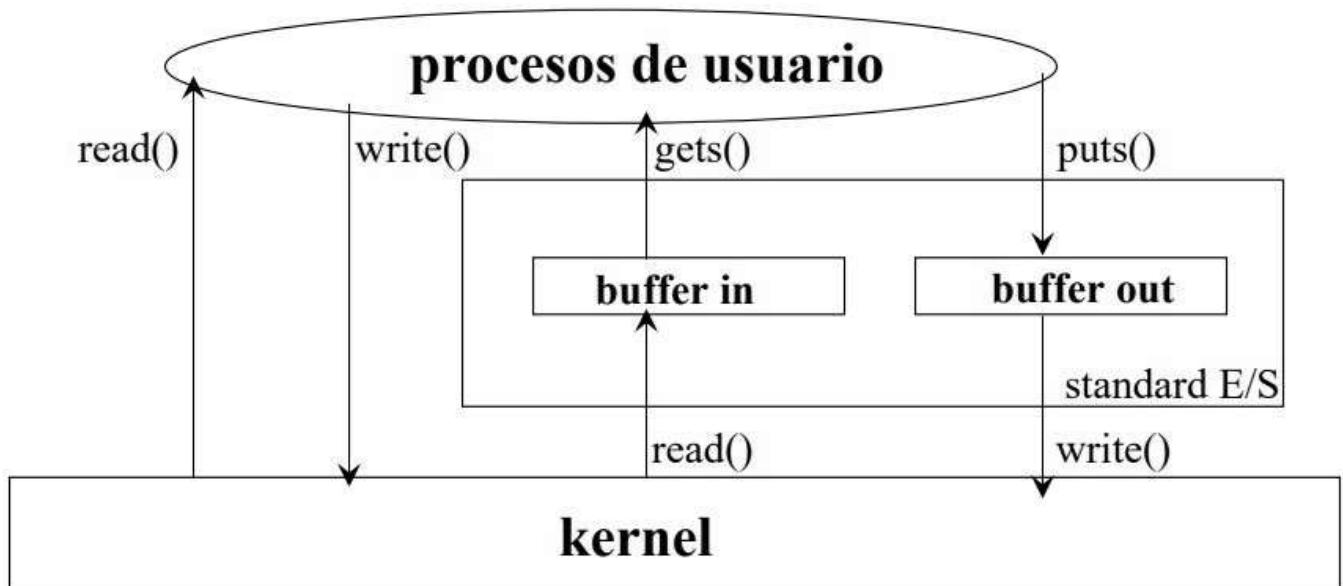
S_IFDIR, S_IFBLK, S_IFCHR, S_IFIFO ...
S_IRWXU, S_IRWXG, S_IRWXO ...

dev solo para ficheros dispositivos (caracteres, bloques)

The diagram illustrates the components of a device identifier. At the bottom center is an oval labeled "drivers". Two arrows point from this central node to two adjacent nodes at the top. The left arrow points to a node containing "número mayor
clase de dispositivo". The right arrow points to a node containing "número menor (ls -l /dev)
número dentro de una clase". Between these two top nodes is the word "y" (and).

- **Devuelve:** 0 si bien ó –1 si error (errno).

E/S estándar vs. llamadas al sistema



- Las funciones de biblioteca agrupan la E/S (menos SC's)
 - Fully buffered: E/S real se realiza cuando el buffer se llena o se ejecuta fflush
 - Line buffered: E/S real cuando llega carácter fin de línea ('\n' o ASCII 10)
Pensado para dispositivos de líneas (terminales, ...)
 - Unbuffered: E/S real sin uso de buffers

E/S estándar vs. llamadas al sistema

- Por defecto
 - stdin y stdout son:
 - Line buffered si están dirigidas a una terminal o
 - Fully buffered en caso contrario
 - stderr es unbuffered
- Se puede modificar el comportamiento de las funciones de biblioteca con setbuf() o setvbuf()
 - Cambiar tamaño de los buffers (por defecto BUFSIZ=1024)
 - fully buffered (_IOFBF)
 - line buffered (_IOLBF)
 - unbuffered (_IONBF)

E/S estándar vs. llamadas al sistema

- Llamadas al sistema

```
int fd;  
/*file descriptor*/  
  
fd = open(nombre, ...);  
read(fd, ...);  
write(fd, ...);  
lseek(fd, ...);  
...  
close(fd);
```

- Funciones de biblioteca C

```
FILE *fp;  
/*file pointer*/  
  
fp = fopen(nombre, ...);  
fgets(fp, ...);  
fprintf(fp, ...);  
fseek(fp, ...);  
...  
fclose(fp);
```

E/S estándar vs. llamadas al sistema

- **Llamadas al sistema**

Entrada/salida estandar

0=STDIN_FILENO
1=STDOUT_FILENO
2=STDERR_FILENO
son file descriptor

```
read(0, ...);  
write(1, ...);  
write(2, ...);  
...  
close(1);
```

- **Funciones de biblioteca C**

Entrada/salida estandar

stdin
stdout
stderr
son file pointer

```
fgets(stdin, ...);  
fprintf(stdout, ...);  
fprintf(stderr, ...);  
...  
fclose(stdout);
```

E/S estándar vs. llamadas al sistema

- Qué ocurre con los bufferes cuando:
 - **fork**: se copia el Buffer_{PADRE} al Buffer_{HIJO}
(si había cosas pendientes de escribir, se escriben 2 veces)
 - **exec**: desaparece el Buffer
(si había cosas pendientes de escribir, se pierden)
 - **exit**: se vacía el Buffer (equivalente a fflush)
 - **terminación involuntaria**:
desaparece el Buffer
(si había cosas pendientes de escribir, se pierden)

Ejemplo: ficheros, E/S estándar, redirecciones

```
#include<stdio.h>
main() {
    int id,estado,fd;

    printf("linea de texto n 1\n");
    if((id=fork())==0) {
        close(1);
        creat("salida.dat",0777);
        write(1,"linea de texto n 2\n",19);
        exit(1);
    }
    else {
        while(wait(&estado)!=id);
        write(1,"linea de texto n 3\n",19);
        exit(0);
    }
}
```

Sistemas
Operativos

Redireccionamiento

Redireccionamiento

- Procesos FILTRO
- Redirección entrada/salida/errores
- Ejemplos
- ¿Cómo hace el *Shell* para redireccionar?

Procesos FILTRO



FILTRO: Proceso que LEE datos de la entrada estándar (*stdin*), los TRANSFORMA y ESCRIBE los datos transformados en la salida estándar (*stdout*, *stderr*)



Esa convención inicial se puede cambiar REDIRECCIONANDO

Redirección

- Desde el intérprete de comandos:
 - Redirección de la salida:
`who > quien_hay.tmp`
 - Redirección de la entrada:
`sort < quien_hay.tmp`
 - Redirección de la salida de errores:
`cc -o ej10 ej10.c 2> error_10`
 - Redirección de la entrada y la salida:
`sort < quien_hay.tmp > quien_hay`



/* ej110.c */

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

main() {
    int fd;
    if( (fd = creat( "quien_hay.tmp", 0640) )== -1)
        if( errno == EACCES ){
            printf( "fichero existe sin permiso escritura\n" );
            exit( 0 );
        }
        else syserr( "creat" );
    close( fd );
    close( 1 );
    fd = open( "quien_hay.tmp", O_WRONLY );
    if( fd == 1 )
        fprintf( stderr , "Salida Redireccionada\n");
    else{
        fprintf( stderr, "Virus\n" );
        exit( 1 );
    }
    execl( "/bin/who", "who", 0 );
    syserr( "execl" );
}
```

who > quien_hay.tmp

se puede mejorar?

Ejercicios

- ej111.c:
Programa que realice las siguientes
redirecciones: sort < quien_hay.tmp >
quien_hay
(quien_hay.tmp es el fichero generado en ej110.c)
- ej112.c:
Programa que ejecute el
comando: date >>
quien_hay
(recordar: >> añade la salida del comando date
al final del fichero quien_hay. Usar open(...
O_APPEND) o lseek())

¿Cómo hace el shell para redireccionar?

```
switch ( fork() )
{
    . . .
    . . .
    case 0:          /* HIJO */
        if ( r_in )      redirector_entrada();
        if ( r_out )     redirector_salida();
        if ( r_out_app)  redirector_salida_append();

        exec( . . . . . . . . . . . . . . . );
default:
    . . .
    . . .
}
```

hereda TDF y por lo tanto
las redirecciones...

La separación de `fork()` y `exec()` simplifica el tratamiento de señales y los redireccionamientos. Además, permite que el código quede perfectamente localizado

Sistemas
Operativos

Comunicación entre Procesos

Comunicación entre Procesos

- Tuberías. Tipos
- PIPES. Llamadas asociadas
- Ejemplos de uso
- Creación de pipes desde el *shell*
- Más ejemplos

Tuberías (1 de 2)

- UNAMED PIPES O PIPES

Dos tipos:

- NAMED PIPES, FIFOS O NAMED FIFOS

CARACTERISTICAS COMUNES

- Unidireccionales (un proceso lee y el otro escribe)
- Implementación: i-nodo del que se utilizan los punteros directos
=> capacidad limitada y dependiente de la implementación: mínimo 4096 bytes



Tuberías (2 de 2)

DIFERENCIAS

UNAMED	NAMED
comunican procesos emparentados	comunican cualquier proceso
No están representados en el Sistema de Ficheros mediante un nombre	Tienen nombre en el Sistema de Ficheros, como un fichero cualquiera
Se crean con pipe()	Se crean con mknod()
Muy utilizados, especialmente en el <i>shell</i> : who sort lp -oq	Muy poco utilizados

PIPES: Llamadas asociadas (1)

PIPES: Llamadas asociadas (1)

- **Sintaxis:** # include <unistd.h>
int **pipe**(int *fd[2]*);
- **Acción:** crea una pipe
asigna i-nodo
crea 2 entradas en TFA

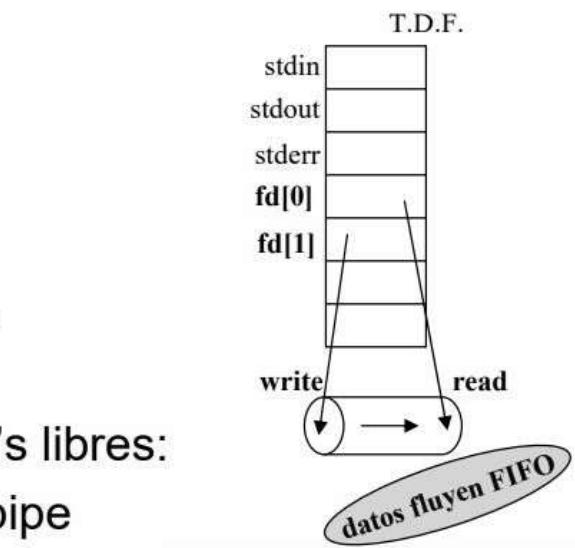
en *fd* se almacenan los 2 primeros fd's libres:

- *fd[0]* abierto para lectura en pipe
- *fd[1]* abierto para escritura en pipe
lo escrito en *fd[1]* se lee por *fd[0]*

apuntan a las correspondientes entradas en TFA

- para manejar la pipe como si fuese un fichero normal (read, write)

- **Devuelve:** 0 si bien ó –1 si error



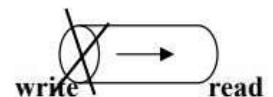
Prob. clásico del
productor-consumidor

Buffer en memoria (más rápido)
16 KiB en hendrix

PIPES: Transmisión de datos

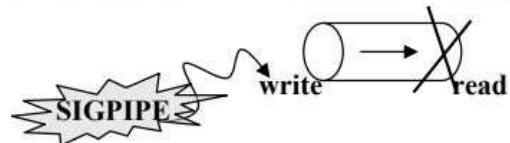
read()

- Se leen datos del extremo de lectura
- Si pipe vacía => bloquea
- Si pipe vacía y extremo de escritura cerrado => no bloquea y devuelve 0 (final de datos)



write()

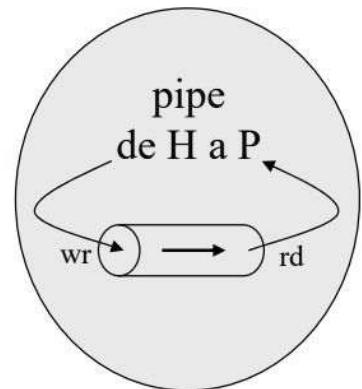
- Se escriben datos en el extremo de escritura
- Si pipe llena => bloquea
- Si no está abierto el extremo de lectura => señal SIGPIPE



Ejemplo de uso de pipes

```
#include "error.h"
main(argc,argv)
int argc;char *argv[];
{ int fpipe[2], file, n; char buf[512];
if(argc!=3) syserr("Numero de parametros");
pipe(fpipe);
switch(fork()) {
case -1: syserr("fork");
case 0: file=open(argv[1],0);
while((n=read(file,buf,sizeof(buf)))!=0)
    write(fpipe[1],buf,n);
printf("Fichero leido\n");
break;
default: close(fpipe[1]);
file=creat(argv[2],0600);
while((n=read(fpipe[0],buf,sizeof(buf)))!=0)
    write(file,buf,n);
printf("Fichero copiado\n");
}
exit(0); }
```

para copiar un fichero perdiendo el tiempo

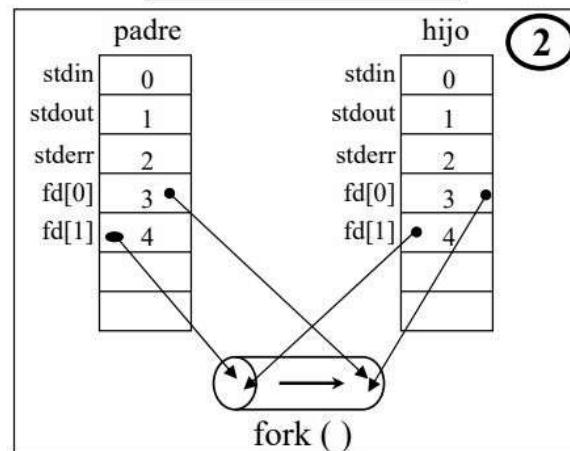
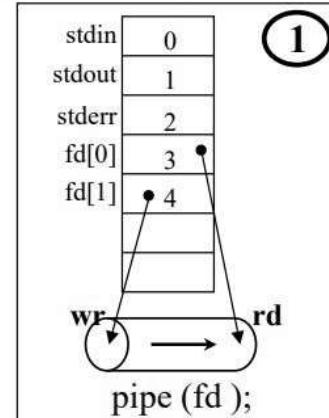


Pipes y redirecciones (1 de 4)

```
#include "error.h"
main()
{
    int fd[2];
    pipe( fd );
    switch( fork() ) {
        case -1: syserr( "fork" );
        case 0:   close( fd[0] );
                   close( 1 );
                   dup( fd[1] );
                   close( fd[1] );
                   execvp( "who", "who", 0 );
        default: close( fd[1] );
                  close( 0 );
                  dup( fd[0] );
                  close( fd[0] );
                  execvp( "sort", "sort", 0 );
    }
}
```

who | sort

1



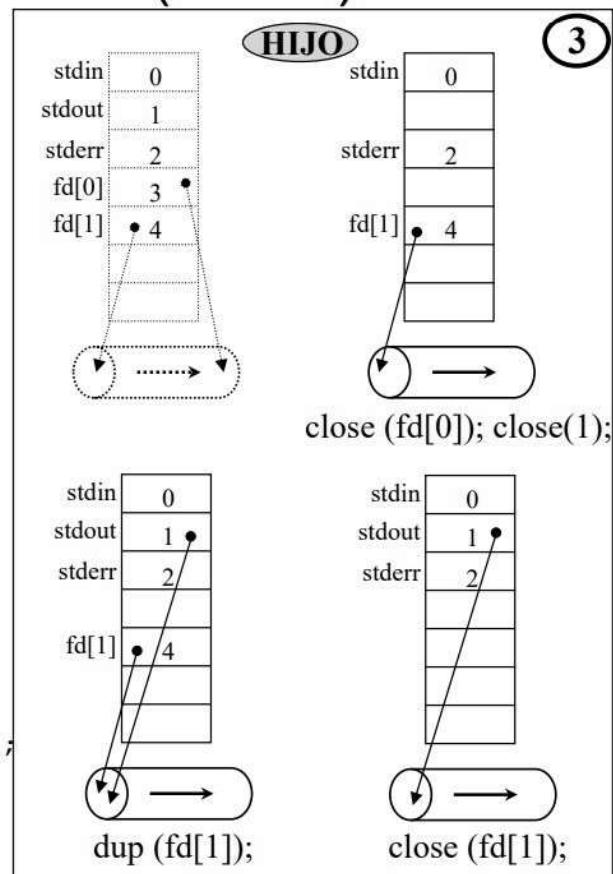
Pipes y redirecciones (2 de 4)

```
#include "error.h"
main()
{ int fd[2];

    pipe( fd );
    switch( fork() ) {
        case -1: syserr( "fork" );
        case 0:  close( fd[0] );
                  close( 1 );
                  dup( fd[1] );
                  close( fd[1] );
                  execlp( "who", "who", 0 );
        default: close( fd[1] );
                  close( 0 );
                  dup( fd[0] );
                  close( fd[0] );
                  execlp( "sort", "sort", 0 );
    }
}
```

3

who | sort

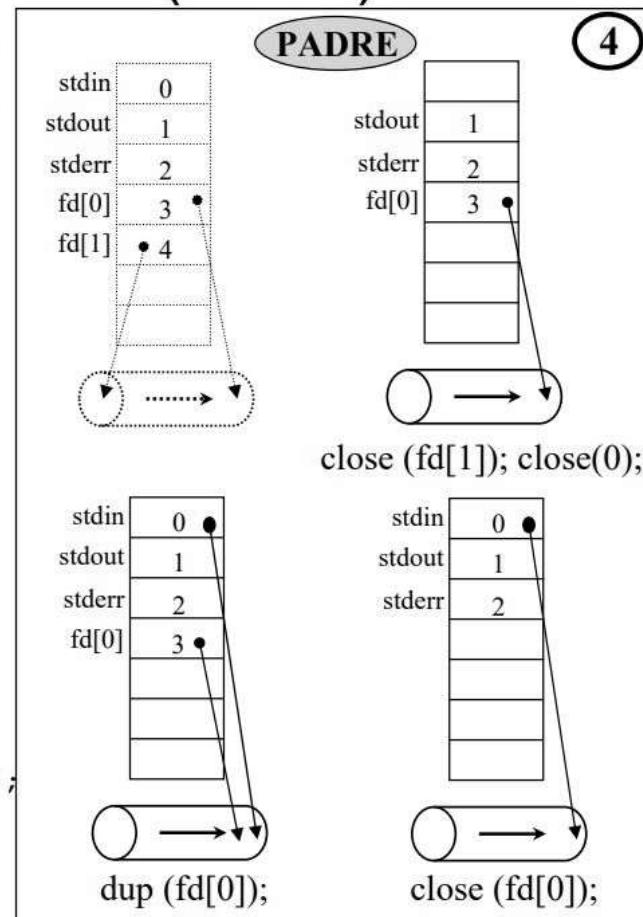


Pipes y redirecciones (3 de 4)

```
#include "error.h"
main()
{ int fd[2];

    pipe( fd );
    switch( fork() ) {
        case -1: syserr( "fork" );
        case 0:   close( fd[0] );
                   close( 1 );
                   dup( fd[1] );
                   close( fd[1] );
                   execlp( "who", "who", 0 );
        default: close( fd[1] );
                  close( 0 );
                  dup( fd[0] );
                  close( fd[0] );
                  execlp( "sort", "sort", 0 );
    }
}
```

who | sort

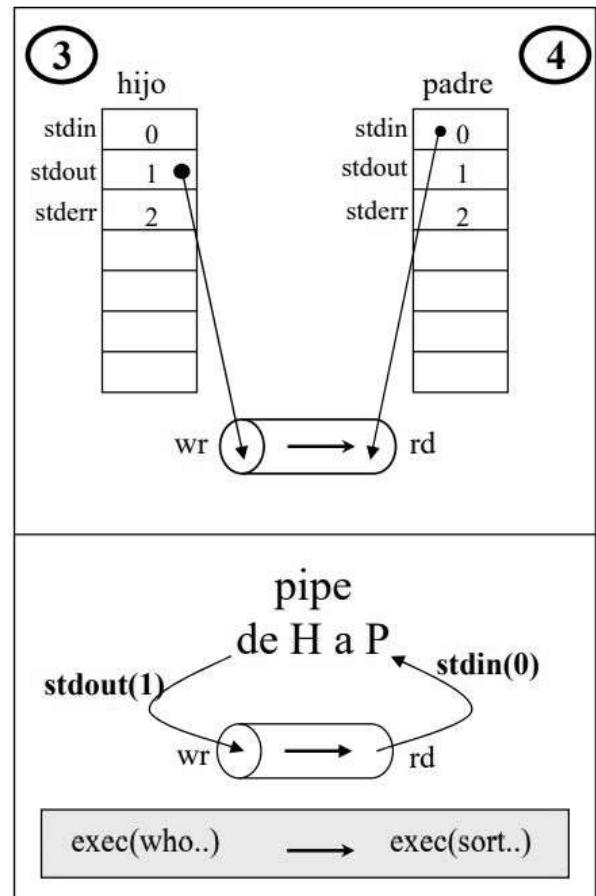


Pipes y redirecciones (4 de 4)

```
#include "error.h"
main()
{ int fd[2];

    pipe( fd );
    switch( fork() ) {
    case -1: syserr( "fork" );
    case 0:   close( fd[0] );
               close( 1 );
               dup( fd[1] );
               close( fd[1] );
               execp( "who", "who", 0 );
    default: close( fd[1] );
              close( 0 );
              dup( fd[0] );
              close( fd[0] );
              execp( "sort", "sort", 0 );
    }
}
```

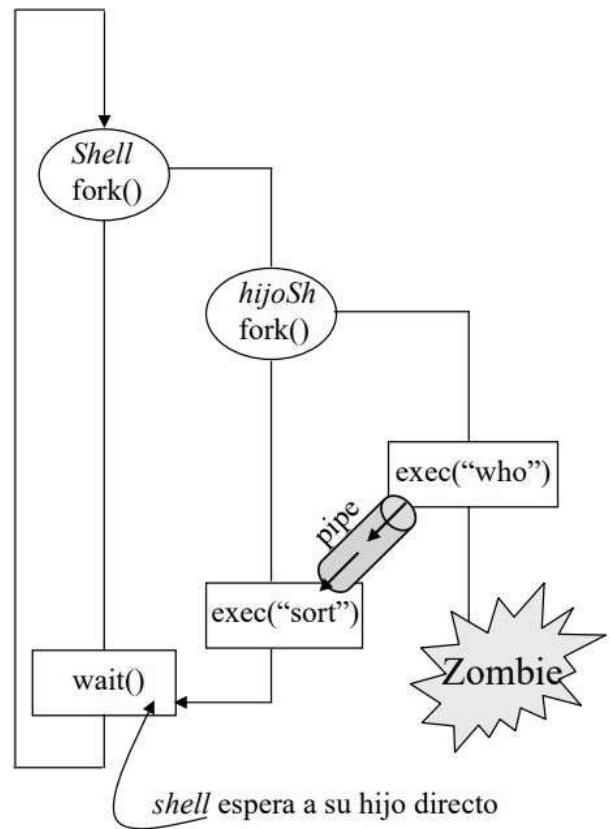
who | sort



Creación de pipes desde el shell

```
for( ; ; ){  
    lectura de comandos...;  
    parsing...;  
    switch( fork() ) {  
        case -1: ...  
        case 0: pipe( fd );  
            switch( fork() ) {  
                case -1: ...  
                case 0: redirecciones;  
                    exec ( "who" );  
                default: redirecciones;  
                    exec ( "sort" );  
            }  
        default: wait( estado );  
    }  
}
```

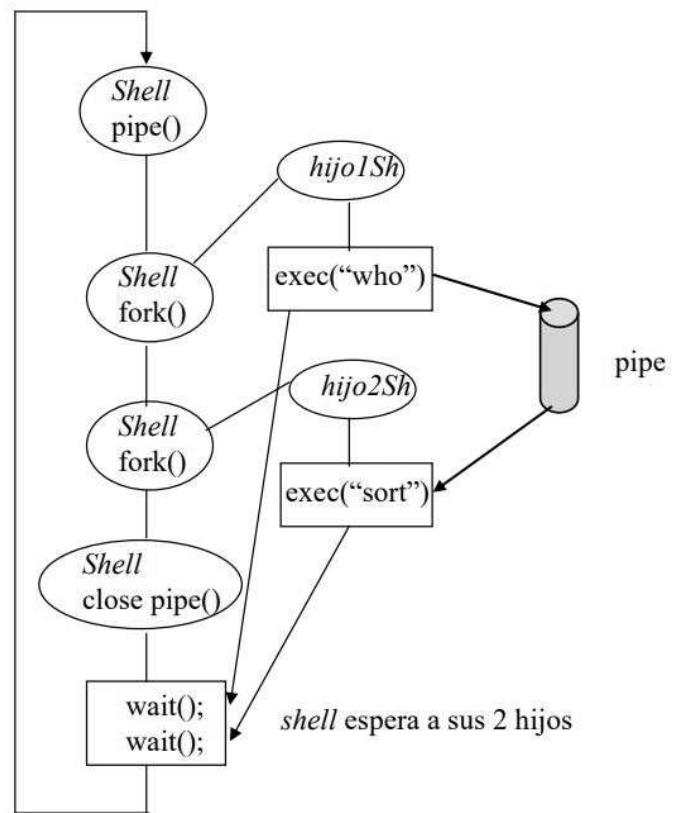
} who / sort



Creación de pipes desde el shell

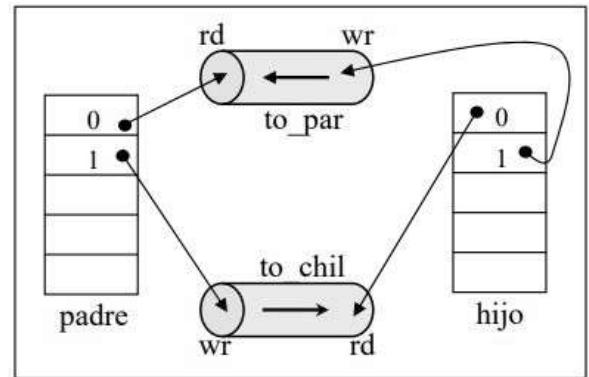
```
for( ; ; ){  
    lectura de comandos...;  
    parsing...;  
    pipe( fd );  
    switch( fork() ) {  
        case -1: ...  
        case 0: redirecciones;  
            exec("who");  
        default: switch( fork() ) {  
            case -1: ...  
            case 0: redirecciones;  
                exec ( "sort" );}}}  
close fd[0] fd[1];  
wait( estado ); /* 1er hijo */  
wait( estado ); /* 2º hijo */
```

who / sort



Comunicación bidireccional con 2 pipes

```
#include <string.h>
char string[] = "hello world";
main()
{   int count, i, to_par[2], to_child[2];
    char buf[256];
    pipe(to_par); pipe(to_child);
    if( fork() == 0 ){
        close(0); dup(to_chil[0]);
        close(1); dup(to_par[1]);
        close(to_par[1]); close(to_chil[0]);
        close(to_par[0]); close(to_chil[1]);
        for( ; ; ){
            if ((count = read(0, buf, sizeof(buf))) == 0)
                exit(1);
            write(1, buf, count);
        }
    }
    close(1); dup(to_chil[1]);
    close(0); dup(to_par[0]);
    close(to_chil[1]); close(to_par[0]);
    close(to_chil[0]); close(to_par[1]);
    for( i = 0 ; i < 15 ; i++ ){
        write(1, buf, count);
        read(0, buf, sizeof(buf));
    }
}
```



Otro ejemplo

```
/* primer.c */  
  
main(){  
    int id, fd[2];  
    id = fork();  
    pipe(fd);  
    switch ( id ) {  
        case -1: exit(1);  
        case 0:  close(0);  
                  dup(fd[0]);  
                  close(fd[0]);  
                  close(fd[1]);  
                  execlp("segun","segun",0);  
                  exit(1);  
        default: close(1);  
                  dup(fd[1]);  
                  close(fd[0]);  
                  close(fd[1]);  
                  execlp("segun","segun",0);  
                  exit(1);  
    }  
    exit(0);  
}
```

```
/* segun.c */  
  
main()  
{  char c;  
  
    while ( read(0, &c, 1 )!= 0 )  
        write(2, &c, 1);  
    write(1, &c, 1);  
    exit(0);  
}
```

No funciona