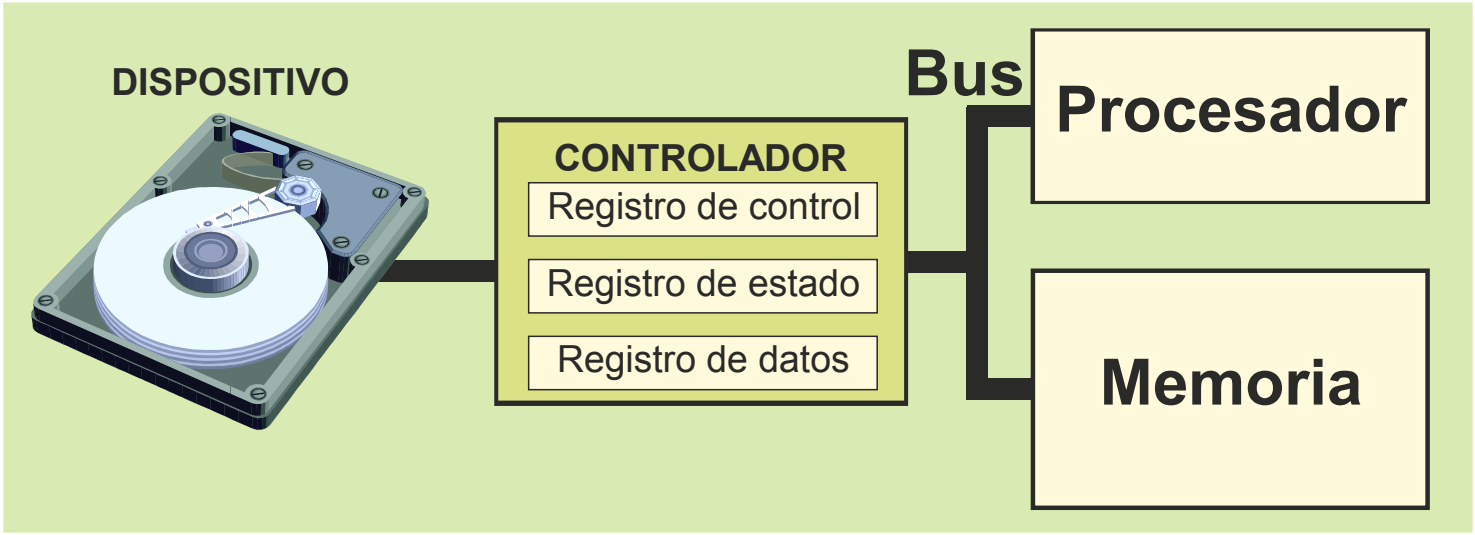
SISTEMAS OPERATIVOS
E/S y SISTEMA de FICHEROS

Pedro de Miguel Anasagasti
Mª de los Santos Pérez Hernández

CONCEPTOS BÁSICOS DE E/S



- Arquitectura con mapa de E/S propio IN y OUT (p.e. Intel)
- Arquitectura con mapa de E/S en memoria LOAD y STORE (p.e. Motorola)

El acceso a los dispositivos es:

- Complejo
  - Detalles físicos de los dispositivos
  - Dependiente de las direcciones físicas
- Sin protección
  - Si el usuario accede a nivel físico no tiene restricciones. El controlador del dispositivo no limita

Hardware de E/S

- Dispositivos de E/S
  - De bloques (discos, cintas, placas red)
  - De caracteres (teclado, ratón)
  - Acceso aleatorio
  - Acceso secuencial
- Controladores de dispositivos
  - E/S programada: No concurrencia E/S-procesador
  - Interrupciones: Concurrencia E/S-procesador
  - DMA: Máxima concurrencia E/S-procesador

Objetivos del SO en E/S

- Controlar el funcionamiento de los dispositivos de E/S
- Facilitar el manejo de los dispositivos de E/S a través de interfaces
- Proporcionar mecanismos de protección
- Explotar la concurrencia E/S-procesador

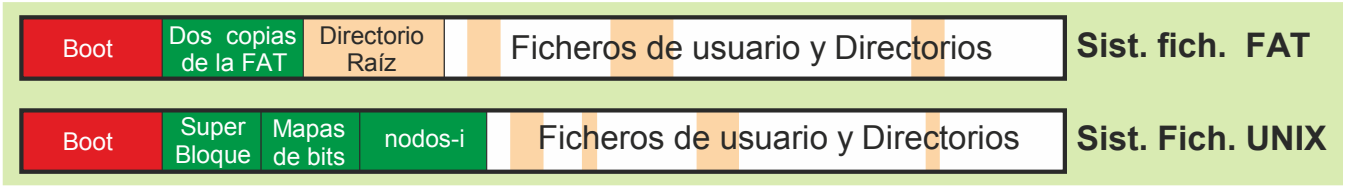
CONCEPTO DE SISTEMA DE FICHEROS

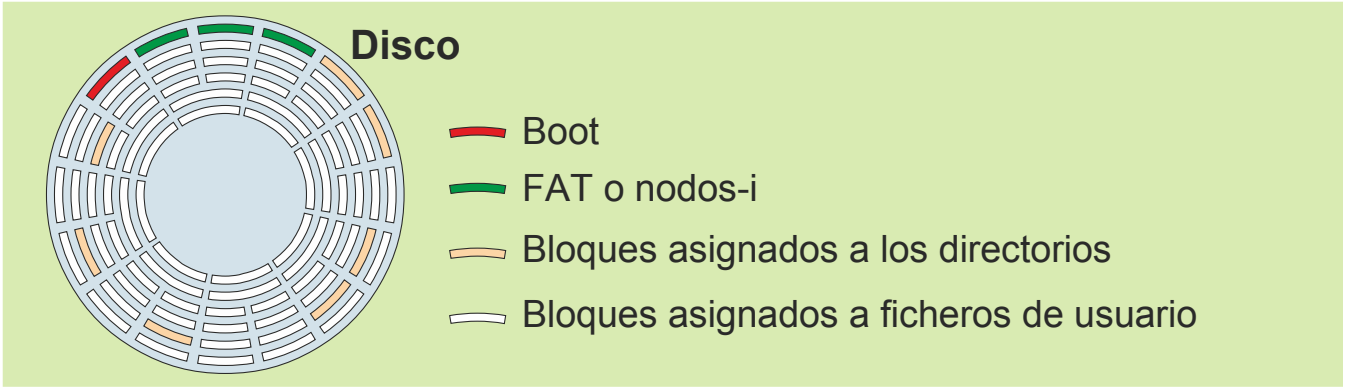
FICHERO Y SISTEMA DE FICHEROS

Fichero. Cónjunto de informaciones relacionadas que se almacenan en almacenamiento secundario y que se identifica mediante un nombre.

Sistema de ficheros. Conjunto autónomo de informaciones incluidas en una unidad de almacenamiento (partición o volumen) que permiten su explotación. Se compone de:

- **Información neta: Ficheros de usuario (programas y datos).**
- **Metainformación**
  - **Estructura física de los ficheros.**
  - **Información asociada a los ficheros.**
  - **Directorios.**





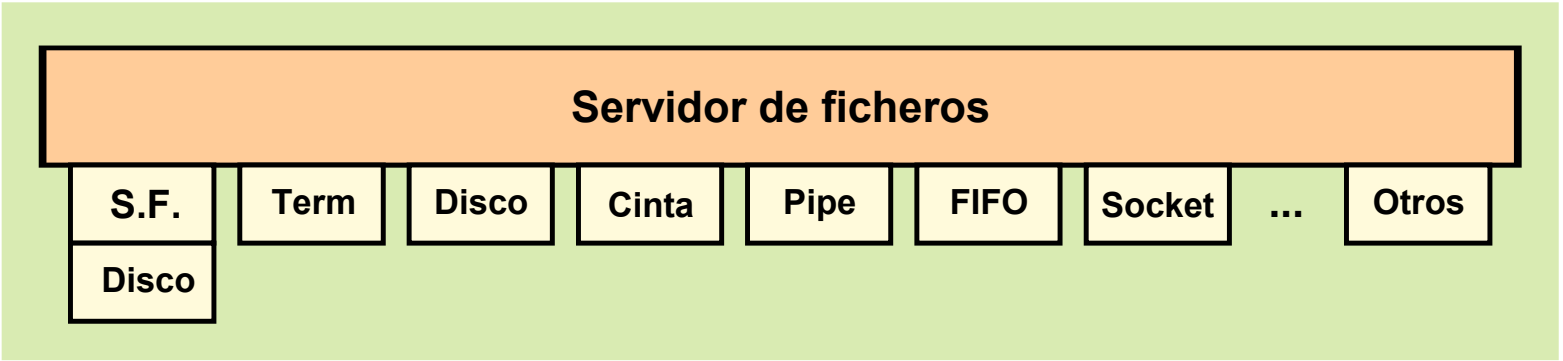
SERVIDOR DE FICHEROS

Servidor de Ficheros: capa de software entre dispositivos y usuarios que:

- Suministra una visión lógica uniforme de los dispositivos, presentándolos como ficheros
- Ofrece primitivas de acceso cómodas e independientes de los detalles físicos
- Incorpora mecanismos de protección

El servidor de ficheros permite manejar los siguientes tipos de objetos:

- Ficheros de usuario (datos y programas)
- Ficheros especiales : Orientado a carácter (ej. terminal)
  Orientado a bloque (ej. disco)
- Mecanismos de comunicación con y sin nombre
  - Pipe, FIFO, socket UNIX

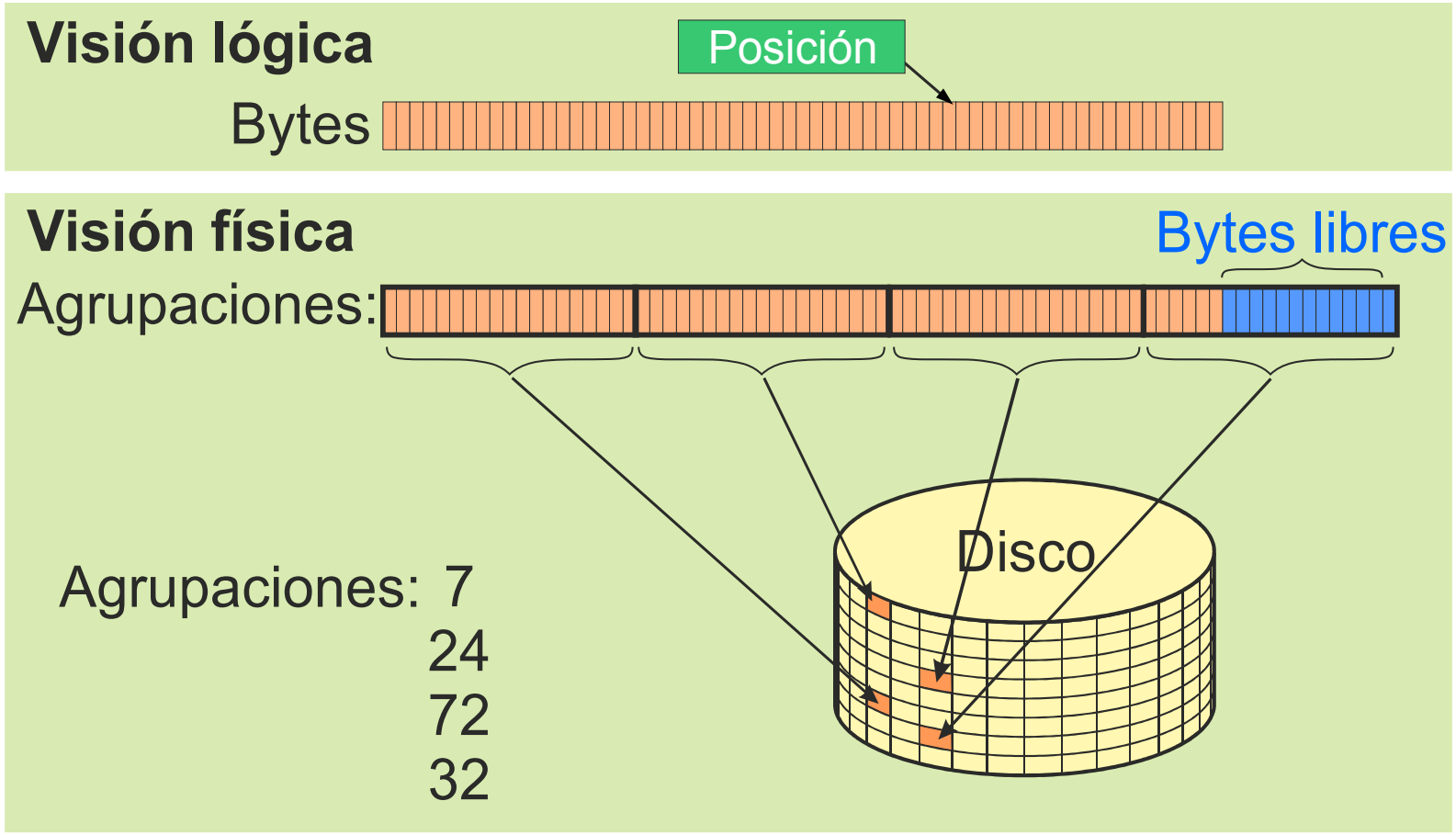



UNIVERSIDAD POLITECNICA MADRID
TECHNICA
IMPENDI
NATIONI

FICHEROS

ESTRUCTURA LÓGICA Y FÍSICA DE UN FICHERO

- **Estructura lógica del fichero. Secuencia ordenada de bytes con un puntero que indica la posición a partir de la cual se lee o escribe.**
  - **Sobre esta cadena de bytes se pueden montar otras estructuras: por ejemplo, conjunto de registros.**
- **Estructura física del fichero. Secuencia ordenada de agrupaciones**



VISIÓN FÍSICA DE UN FICHERO

Sector: Unidad mínima de transferencia que puede manejar el controlador de disco (2^m bytes, normalmente 2^9)

Bloque: Es un conjunto de sectores de disco y es la unidad de transferencia mínima que usa el sistema de ficheros (bloque = 2^n sectores)

- Única para cada sistema de ficheros y definible por el usuario
- El bloque se puede direccionar de manera independiente

Agrupación: Conjunto de bloques que se utilizan como una unidad lógica de gestión de almacenamiento (agrupación = 2^p bloques).

- Reserva espacio

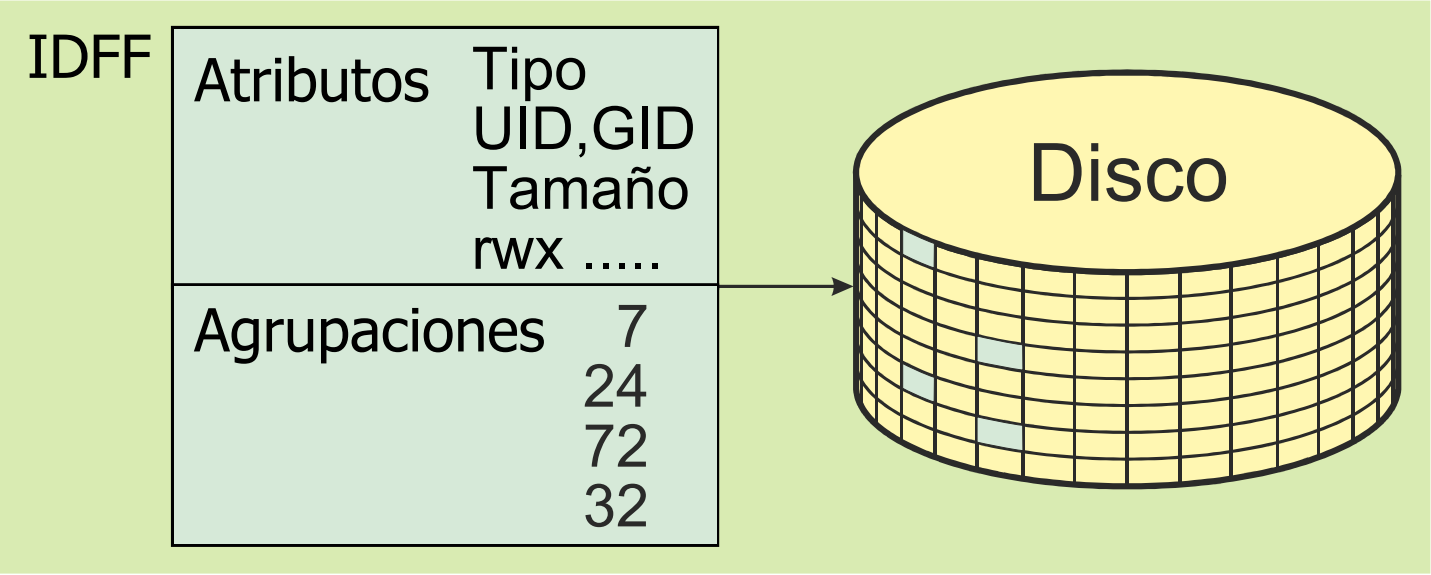
Al fichero se le asignan agrupaciones, pero se accede siempre en bloques

En algunos sistemas Bloque = Agrupación

IDENTIFICADOR DE DESCRIPCIÓN FÍSICA DE FICHERO (IDFF)

IDFF. Estructura de información que recoge la descripción física del fichero

- **Secuencia ordenada de agrupaciones + ATRIBUTOS**
- **Almacenada en el propio disco**
- **FAT en MS-DOS**
- **Registro MFT (Master File Table) de NTFS (New Technology File System) Microsoft**
- **Nodo-i en UNIX**



CARACTERÍSTICAS DE LOS FICHEROS

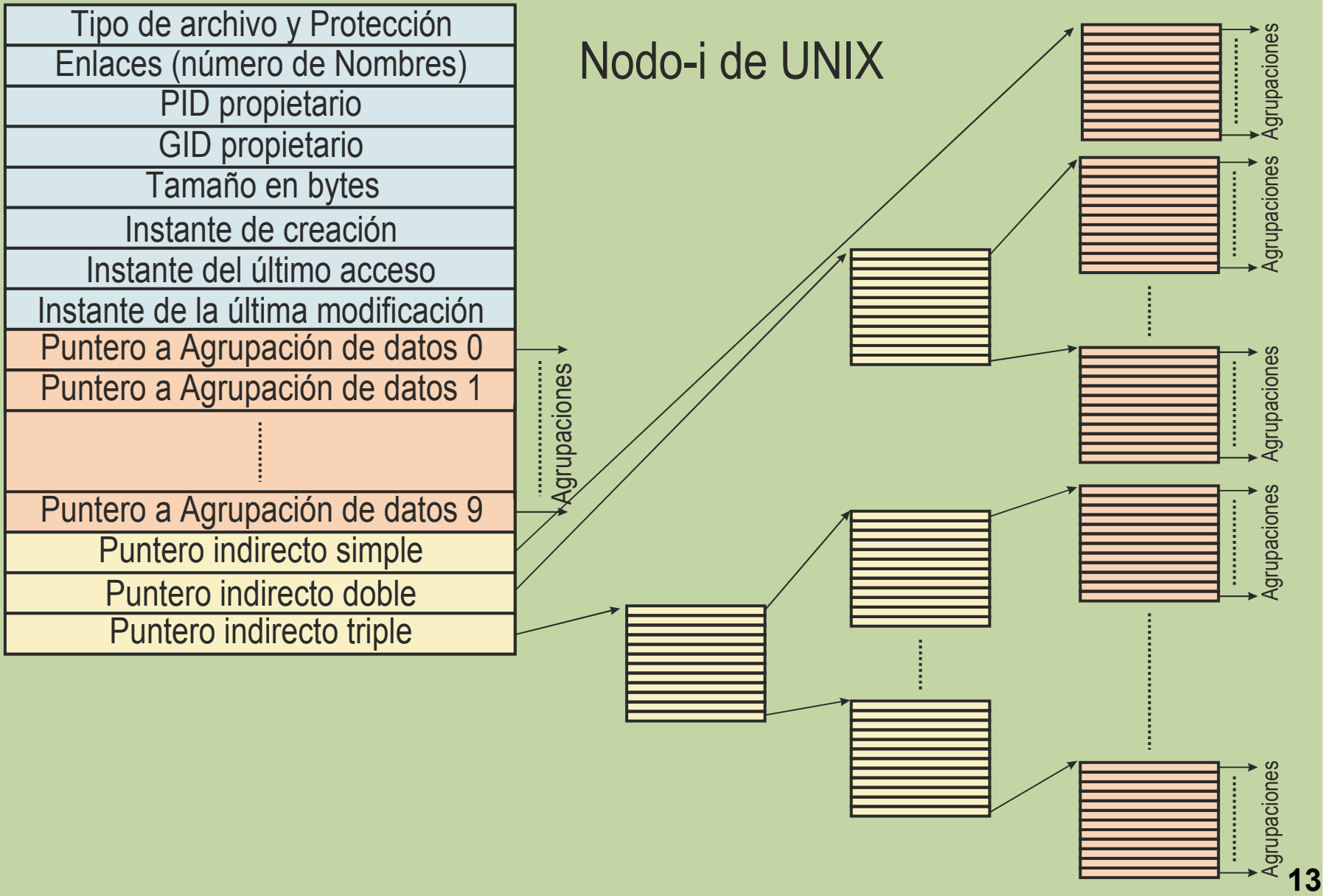
Atributos de un fichero

- Identificador único del fichero
- Tipo de fichero
- Dueño y grupo
- Información de protección
- Tamaño real en bytes
- Hora y fecha de creación
- Hora y fecha del último acceso
- Hora y fecha de la última modificación
- Número de enlaces (número de nombres)

Tamaño máximo de un fichero

- Depende de las limitaciones establecidas por:
  - La metainformación (Atributo tamaño real, direccionamiento de las agrupaciones –tamaño direcciones y nº de direcciones–)
  - Las agrupaciones de datos disponibles (agrupaciones existentes en disco o cuotas establecidas por el administrador)

- **Tamaño típico 128 B. Punteros de 4B.**



Asignación de espacio al fichero

- Se hace por agrupaciones.
- El fichero puede quedar disperso o fragmentado. Sus agrupaciones no tienen por qué ser contiguas.
  - Objetivo de la desfragmentación: mejorar el acceso secuencial.
  - SSD no se desfragmentan.

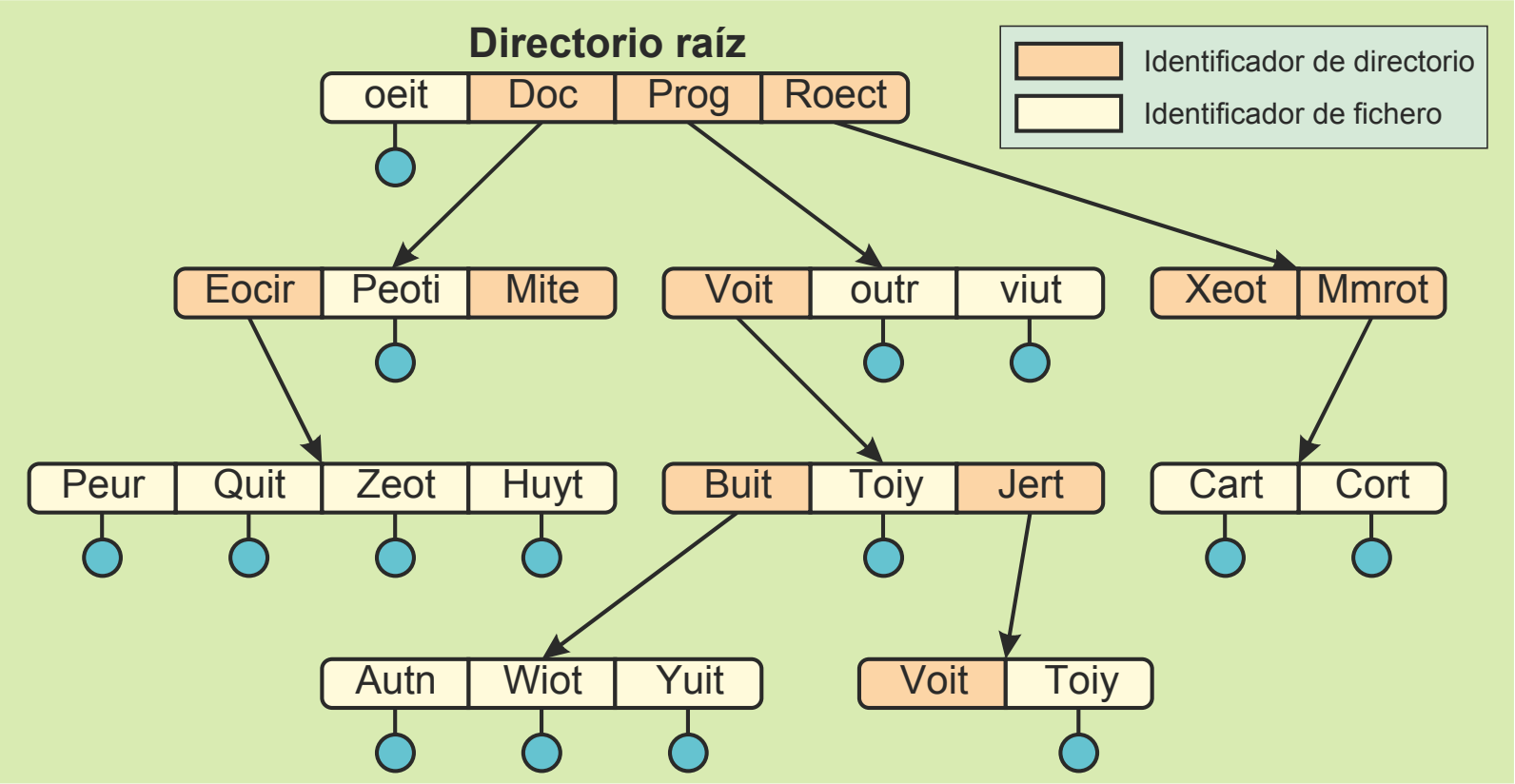
Gestión del espacio libre

- Agrupaciones libres/ocupadas.
- IDFF (p.e. nodos-i) libres/ocupados.

DIRECTORIOS

Estructura lógica: Esquema jerárquico de nombres:

- **Directorio raíz.**
- **Subdirectorios.**



- **Ejemplo UNIX: /usr/include/stdio.h**
- **Ejemplo Windows: C:\DocenciaSO\sos2\transparencias\pipes.ppt**

Directorio único versus directorio por dispositivo. Operación de montado

ESTRUCTURA LÓGICA DE LOS DIRECTORIOS

- Un directorio es una tabla con entradas que asocian nombres simbólicos con identificadores de fichero (nº IDFF / nº nodo_i)
- Esquema jerárquico. La organización jerárquica de un directorio:
  - Simplifica el nombrado de ficheros (nombres únicos)
  - Proporciona una gestión de la distribución => agrupar ficheros de forma lógica (mismo usuario, misma aplicación)
- Cuando se abre un fichero el SO busca el nombre en la estructura de directorios y comprueba los privilegios. Es una operación larga y costosa

DIRECTORIO DE TRABAJO O ACTUAL

- **El SO mantiene en el BCP el nombre del directorio actual o de trabajo**
- **El SO mantiene en memoria el nodo_i del directorio de trabajo**
- **Un proceso puede cambiar su directorio de trabajo**
  - **El SO comprueba los permisos antes del cambio**
- **Nobre absoluto**
  - **Es el nombre desde el directorio raíz**
  - **Empieza por / en UNIX o por la unidad en Windows**
- **Nombre relativo**
  - **Es el nombre a partir del directorio de trabajo**
  - **Ahorro de accesos a disco**
    - Se parte del nodo_i del directorio de trabajo
    - Se requieren menos accesos a disco para analizar el nombre relativo que el nombre absoluto

NOMBRADO DE FICHEROS Y DIRECTORIOS

Nombres de fichero y directorio POSIX:

- **Nombre completo (empieza por /)**

  `/usr/include/stdio.h`

- **Nombre relativo al directorio actual o de trabajo (no empieza por /)**

  `stdio.h` **asumiendo que** `/usr/include` **es el directorio actual.**

- **La entradas** `.` **(propio directorio) y** `..` **(directorio padre) pueden utilizarse para formar rutas de acceso**

  `../include/stdio.h`

  `./../include/stdio.h`

  `/usr/./include/../include/stdio.h`

  Estos tres ficheros hacen referencia a `stdio.h` **si** `/usr/include` **es el directorio actual**

- **Árbol único (los dispositivos se montan en un punto del árbol)**

Nombres de fichero y directorio Windows:

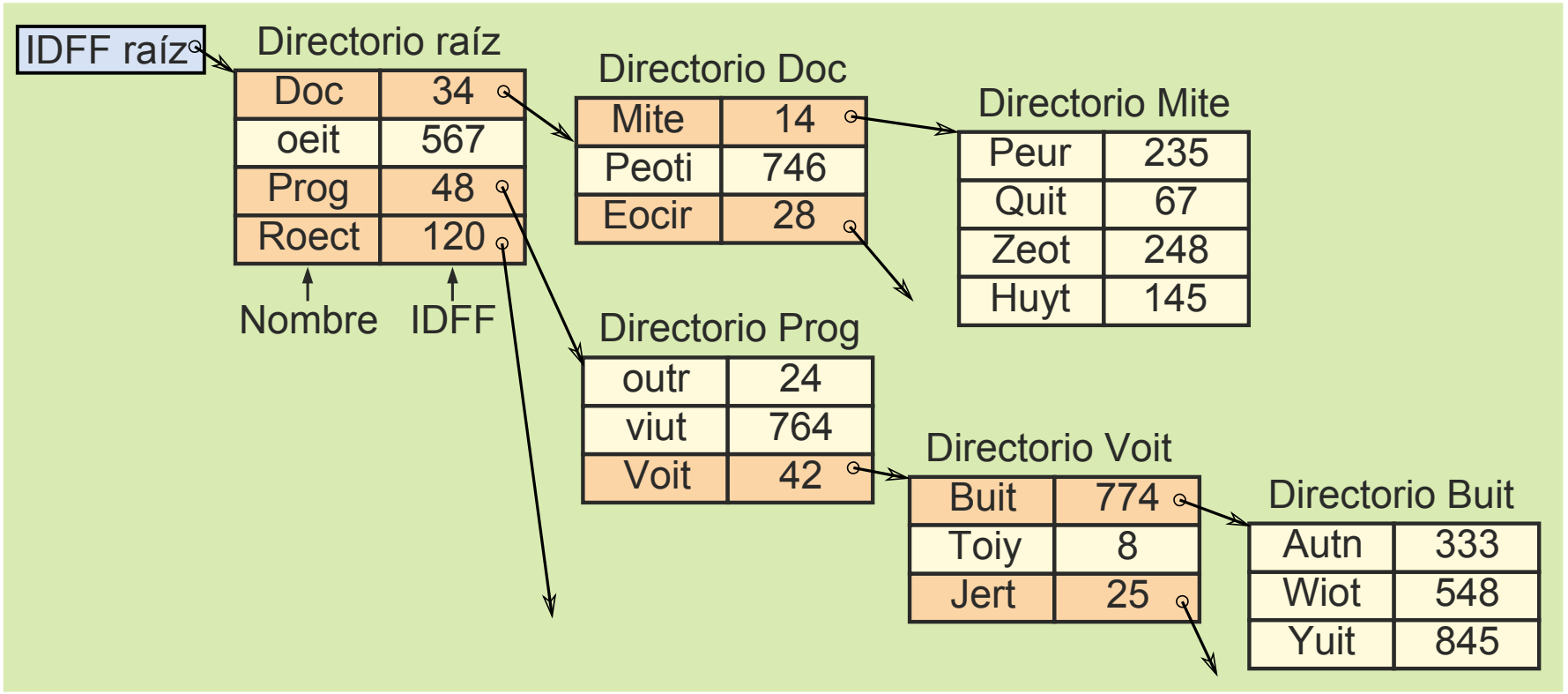
- **En lugar de utilizar /, se utiliza **
- **Entradas** `.` **(propio directorio) y** `..` **(directorio padre)**
- **Varios árboles (uno por sistema de ficheros). NTFS permite montado**

ESTRUCTURA FÍSICA DE LOS DIRECTORIOS

Estructura física: Tabla Nombre-IDFF. (En UNIX es tabla Nombre-nºNodo_i)

IDFF: identificador de descripción física de fichero. Permite obtener la tabla que describe el fichero (en UNIX es el nodo-i).

Para alcanzar el IDFF (nodo_i en UNIX) de un fichero hay que recorrer el árbol.



SERVICIOS DEL SERVIDOR DE FICHEROS

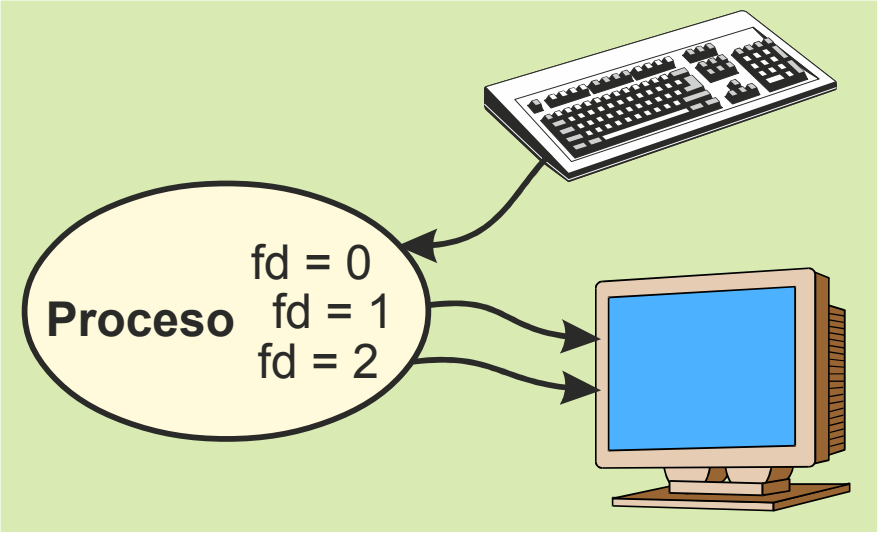
VIDA DEL FICHERO

- Se crea.
  - Se abre → se obtiene un fd (descriptor de fichero).
    - Se escribe y lee a través del fd.
  - Se cierra.
- Se borra.

- Para trabajar con un fichero hay que definir una sesión con las llamadas open y close.
- Las operaciones de lectura y escritura se hacen a partir del puntero de posición, que queda modificado.
- Las escrituras pueden hacer crecer al fichero.

DESCRIPTOR DE FICHERO fd

- Entero no negativo que identifica un fichero abierto
- Se almacenan en el BCP
- Se asignan en orden 0, 1, 2, ...
- Los procesos tienen al menos:
  - 0 entrada estándar
  - 1 salida estándar
  - 2 salida de error



- El proceso hijo hereda los descriptores del padre (fork())
- Los ficheros abiertos siguen abiertos después del exec()
- Importante: Todas las llamadas al sistema que devuelven un descriptor de fichero, devuelven el más bajo disponible (por proceso), salvo dup2

SEMÁNTICA DE COUTILIZACIÓN

Semántica de coutilización: especifica el efecto de varios procesos accediendo de forma simultánea al mismo fichero

Existen diferentes tipos de semánticas

Semántica de coutilización UNIX

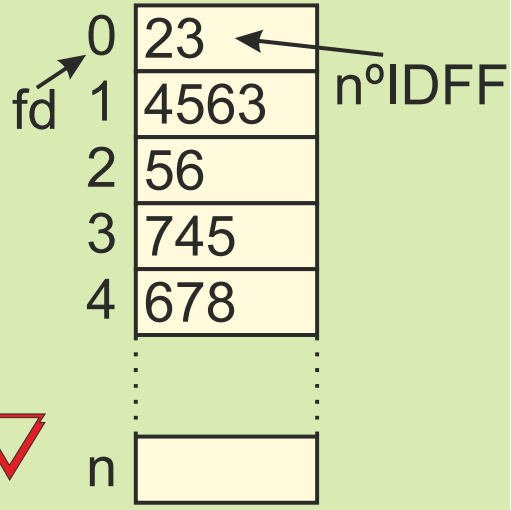
- **Los procesos pueden compartir ficheros de forma simultánea**
- **Las escrituras son inmediatamente visibles para todos los procesos con el fichero abierto**
- **Los datos se escriben en orden de llegada al SO (si se quiere un orden específico => usar cerrojos)**
- **El puntero se comparte cuando se hereda o duplica el descriptor**
- **El puntero se crea en el open**
- **La coutilización afecta a los metadatos**

- Analiza el nombre completo hasta obtener el nºIDFF (nºNodo_i) del fichero.
- Llena la primera entrada libre de la tabla de descriptores de ficheros del BCP. De momento, consideramos que se introduce el nº IDFF.
- El programa utiliza el fd mientras que el SO utiliza el nº IDFF.

TABLA DE PROCESOS

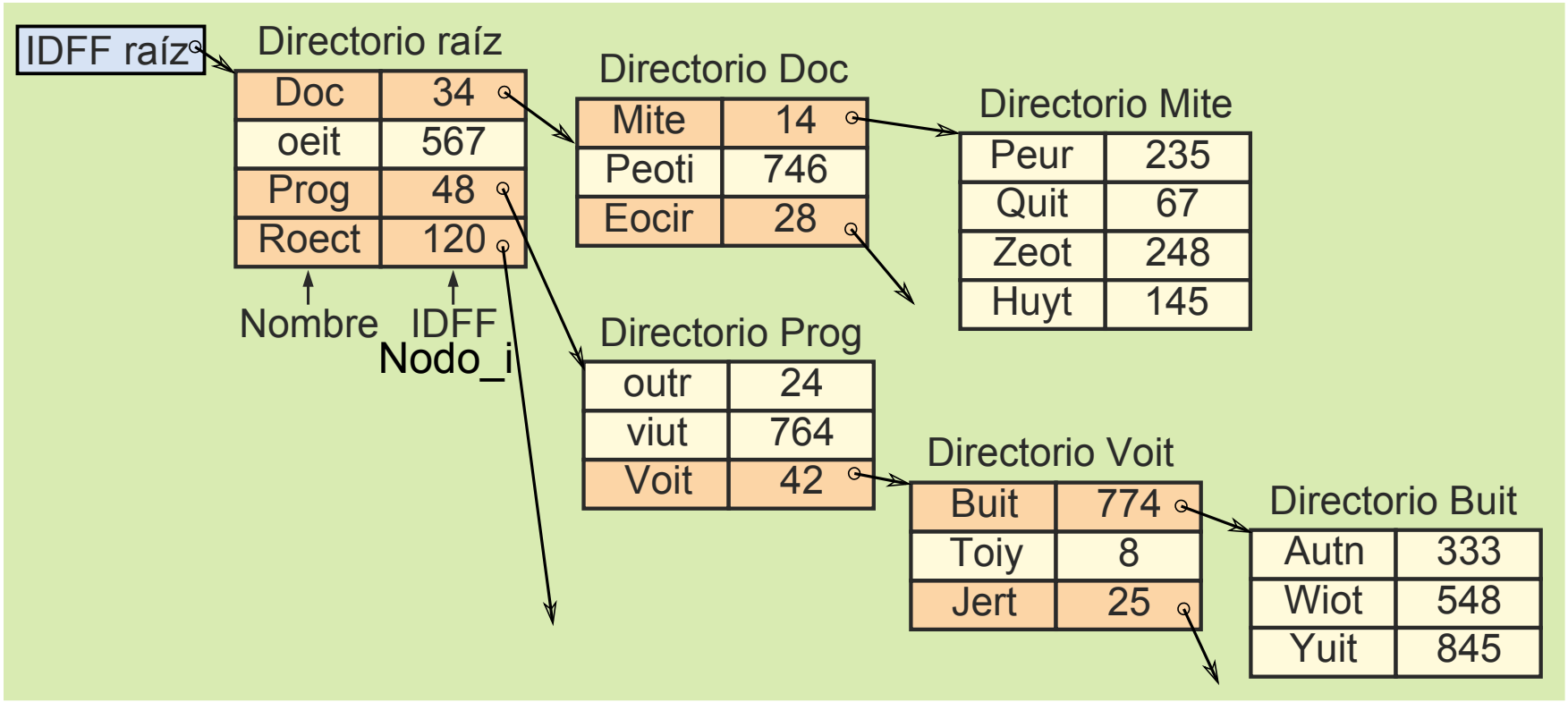
BCP 0	BCP 1		BCP n
pid	pid		pid
uid, gid real	uid, gid real		uid, gid real
uid, gid efect.	uid, gid efect.		uid, gid efect.
pid padre	pid padre		pid padre
Estado (registros)	Estado (registros)		Estado (registros)
Segmentos memoria	Segmentos memoria		Segmentos memoria
Tabla de fd	Tabla de fd		Tabla de fd

Tabla de fd (file descriptors)



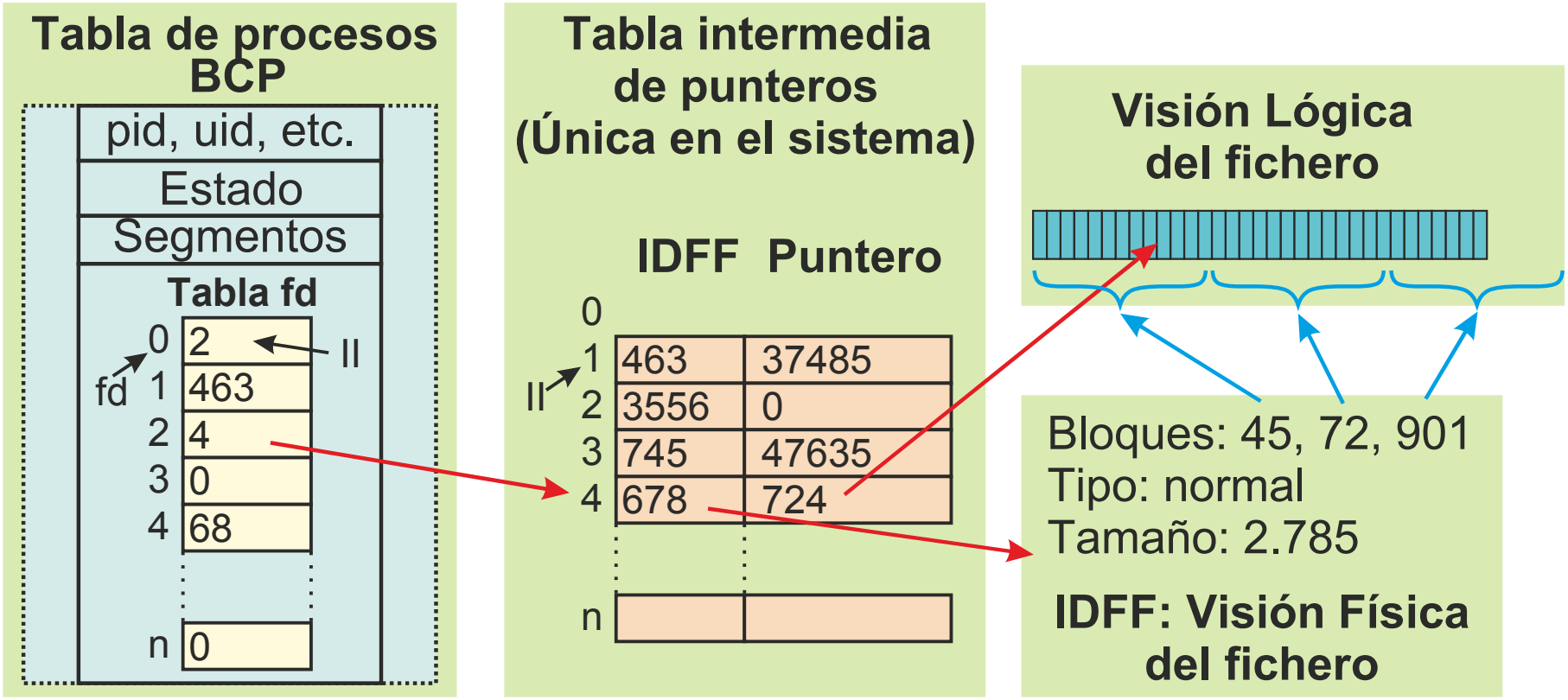
- ¿Dónde están los punteros de los ficheros?
- ¿Se quieren compartir los punteros?

- **Recorrer las tablas de directorio hasta encontrar el nombre buscado.**
- **Se analizan los permisos que están almacenados en los IDFF (Nodo_i).**
- **Se parte del directorio raíz o del directorio de trabajo.**



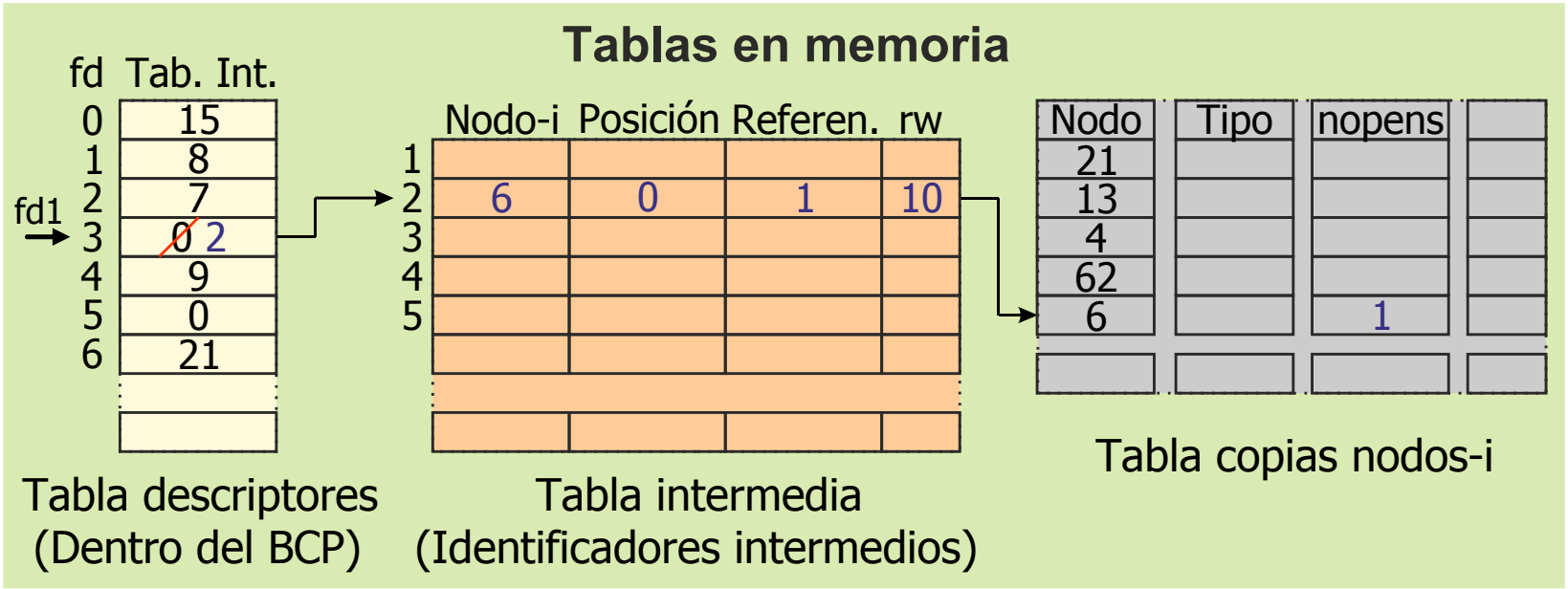
- La tabla fd existe por proceso y contiene Identificadores Intermedios (II).
- En memoria hay una tabla intermedia de punteros única que relaciona los II con los punteros de posición para escritura y lectura y con los IDFF.
- El valor II = 0 indica que ese elemento de la tabla de fd está libre.

(p.e. el programa usa fd = 2 → II = 4 → IDF = 678; Posición = 724)



SERVICIO OPEN IV

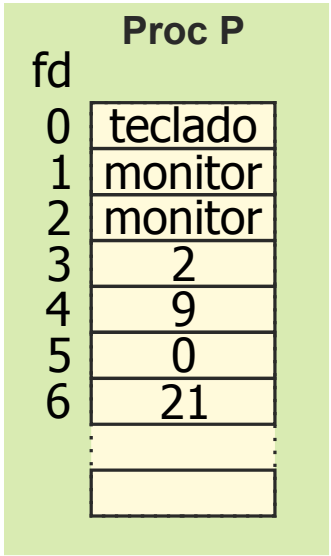
- Exige recorrer el árbol de directorios
- Se comprueban los derechos de acceso en cada directorio recorrido
- Se añade una entrada en la tabla de ficheros abiertos que existe en el BCP del proceso
- Se añade una entrada en la tabla intermedia de punteros, poniendo el puntero a 0 y las referencias a 1. (Al crear un hijo se incrementará el campo de referencias)
- Si el fichero no estaba abierto se copia el nodo_i en la tabla de nodos_i residente en memoria y se añade el campo nopens (nº de opens). Se hace nopens = 1
- Si el fichero ya estaba abierto se hace nopens = nopens + 1, para llevar la cuenta de cuantas veces está abierto el fichero



OPEN. ABRE UN FICHERO

Servicio: (man 2 open)

```
int open(char *name, int flags{, mode_t mode});
```



Argumentos:

- *name*: Nombre del fichero
- *flags*: Opciones de apertura:
  - O_RDONLY: Sólo lectura
  - O_WRONLY: Sólo escritura
  - O_RDWR: Lectura y escritura
  - O_APPEND: Se accede siempre al final del fichero
  - O_CREAT: Si existe no tiene efecto. Si no existe lo crea
  - O_TRUNC: Trunca a cero si se abre para escritura
- *mode*: Bits de permiso para el fichero. Valen sólo cuando se crea (O_CREAT)

Devuelve: Un descriptor de fichero o -1 si hay error o señal (errno = EINTR*)

Descripción:

- Se comprueban los derechos de acceso de todo el camino especificado (absoluto o relativo)
- Se asigna descriptor de fichero en la tabla del BCP, una entrada en la tabla intermedia y se copia el nodo_i a memoria o incrementa su nº opens

Ejemplos:

```
fd = open("/home/juan/dat.txt", O_RDONLY);
fd = open("/home/juan/dat.txt", O_WRONLY|O_CREAT|O_TRUNC, 0640);
```

CREAT. CREA Y ABRE PARA ESCRITURA UN FICHERO

Servicio:

```
int creat(char *name, mode_t mode);
```

Argumentos:

- *name*: Nombre de fichero
- *mode*: Bits de permiso para el fichero (en caso de nueva creación)

Devuelve:

- Devuelve un descriptor de fichero o -1 si fracasa

Descripción:

- Se comprueban los derechos de acceso de todo el camino
- El fichero se abre sólo para escritura
- Si no existe crea un fichero vacío
  - UID_dueño = UID_efectivo
  - GID_dueño = GID_efectivo
  - Con los permisos indicados enmascarados (mode & ~umask)
- Si existe lo trunca (tamaño = 0) sin cambiar los bits de permiso
- Se asigna descriptor de fichero en la tabla del BCP, una entrada en la tabla intermedia y se copia el nodo_i en memoria o incrementa su nº opens

Ejemplos que producen el mismo resultado:

```
fd = creat("datos.txt", 0751);
fd = open("datos.txt", O_WRONLY | O_CREAT | O_TRUNC, 0751);
```

¿Tienen sentido estos permisos?

READ. LECTURA DE UN FICHERO*

Servicio:

```
ssize_t read(int fd, void *buf, size_t n_bytes);
```

Argumentos:

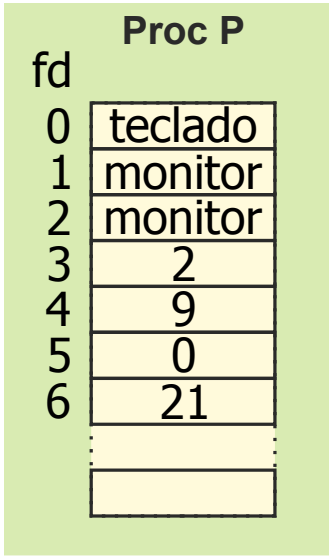
- ***fd***: **descriptor de fichero**
- ***buf***: **zona donde almacenar los datos**
- ***n_bytes***: **número de bytes a leer**

Devuelve:

- **Número de bytes realmente leídos o -1 si fracasa**

Descripción:

- **Transfiere *n_bytes* como máximo**
- **Puede leer menos datos de los solicitados si se llega al fin de fichero. También si lee de un terminal, de un pipe o de un socket**
- **El servicio puede fracasar por una señal, retornando -1. NO ES ERROR**
- **Después de la lectura se incrementa el puntero del fichero con el número de bytes realmente transferidos**
- **Si retorna 0, indica final de fichero**
- **En ficheros especiales se queda bloqueado hasta que hay datos**



*: Objeto asociado a un descriptor de fichero, en general

WRITE. ESCRITURA DE UN FICHERO *

Servicio:

```
ssize_t write(int fd, void *buf, size_t n_bytes);
```

Argumentos:

- ***fd***: descriptor de fichero
- ***buf***: zona de datos a escribir
- ***n_bytes***: número de bytes a escribir

Devuelve:

- Número de bytes realmente escritos -1 si fracasa

Descripción:

- Transfiere *n_bytes* o menos
- Si se rebasa el fin de fichero el fichero aumenta de tamaño
- Puede escribir menos datos de los solicitados si se llega al tamaño máximo del fichero o se rebasa algún limite de implementación del sistema operativo
- El servicio puede fracasar por una señal, retornando -1 . NO ES ERROR
- Después de la escritura se incrementa el puntero del fichero con el número de bytes realmente transferidos

*: Objeto asociado a un descriptor de fichero, en general

EJEMPLOS


```
int total;
n = read(d, &total, sizeof(int)); /*Se lee un entero */
total = 1246; /*Si sizeof(int) = 4 total ? 00 00 04 DE */
n = write(d, &total, sizeof(int));
   /*Se escriben los cuatro bytes 00, 00, 04 y DE, no los cuatro caracteres ASCII
   1, 2, 4 y 6 */
float m[160];
n = read(d, m, sizeof(float)*160);
/*Se leen hasta 160 float, lo que pueden ser 4*160 bytes */
typedef struct registro{
    int identificador;
    float edad, altura, peso;
} registro;
registro individuo[500];
n = read(d, individuo, 500*sizeof(registro));
estadistica_ciega(individuo, n/sizeof(registro));
```

CLOSE. CIERRE DE UN DESCRIPTOR DE FICHERO

Servicio:

```
int close(int fd);
```

Argumentos:

- ***fd***: descriptor de fichero

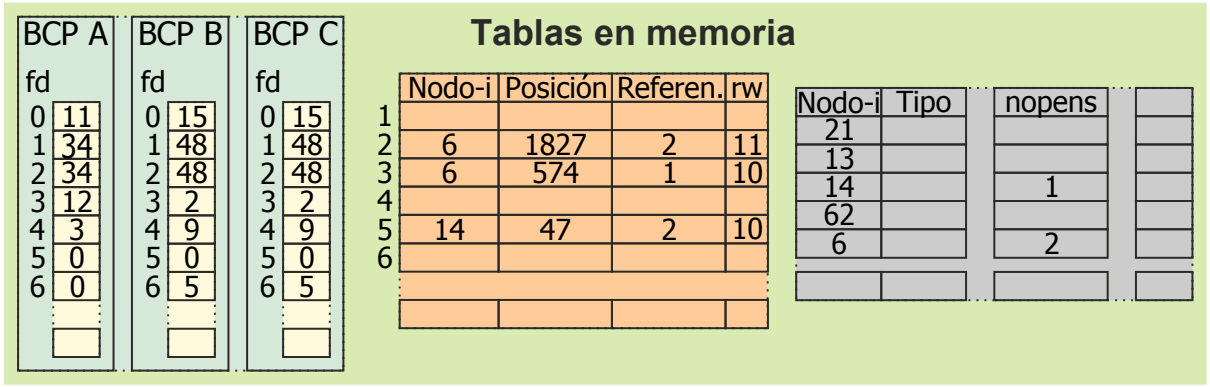
Devuelve:

- 0 o -1 si fracasa

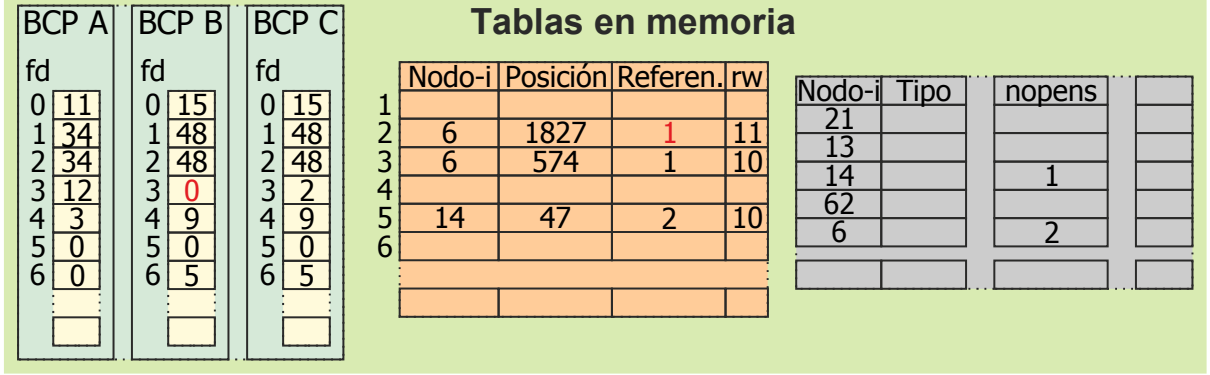
Descripción:

- El proceso pierde la asociación entre el descriptor y el fichero (u objeto correspondiente)
- Se decrementa el número de referencias en la tabla intermedia. Si referencias = 0 entonces:
  - Se libera la correspondiente entrada de la tabla intermedia y
  - Se decrementa nopens (número de duplicados) de la tabla copias nodos_i
  - Si nopens = 0 se libera la entrada en la tabla copias nodos_i

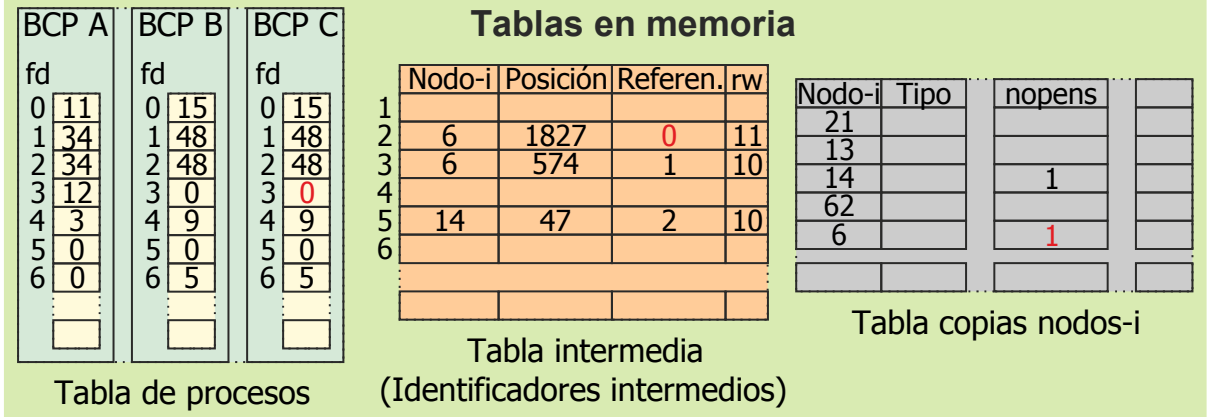
CLOSE. CIERRE DE UN DESCRIPTOR DE FICHERO



Proceso B ejecuta: `close (3);`



Proceso C ejecuta: `close (3);`



EJEMPLO FICHEROS. COPIA UN FICHERO EN OTRO I

Los nombres de los ficheros se pasan como argumentos

```
#define BUFSIZE 4096
int main(int argc, char **argv) {
int fd_ent, fd_sal, n_read;
char buffer[BUFSIZE];
//Abre fichero origen o entrada
fd_ent = open(argv[1], O_RDONLY);
if (fd_ent < 0) {
   perror("open"); //transforma errno en un mensaje de error
   exit(1);
}
//Crea fichero de destino o salida
fd_sal = creat(argv[2], 0666);
if (fd_sal < 0) {
   close(fd_ent);
   perror("creat");
   exit(1);
}
```

EJEMPLO FICHEROS. COPIA UN FICHERO EN OTRO II


```
/* bucle de lectura del fichero de entrada */
while ((n_read = read(fd_ent, buffer, BUFSIZE)) > 0) {
   /* escribir el buffer al fichero de salida */
   if (write(fd_sal, buffer, n_read) < n_read) {
      perror("write");
      close(fd_ent);
      close(fd_sal);
      exit(1);
   }
}
close(fd_ent);
close(fd_sal);
if (n_read < 0) {
   perror("read");
   exit(1);
   }
return 0;
}
```

LSEEK. MODIFICA EL PUNTERO DE POSICIÓN

Servicio:

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

Argumentos:

- ***fd***: Descriptor de fichero
- ***offset***: desplazamiento (positivo o negativo)
- ***whence***: base del desplazamiento

Devuelve:

- La nueva posición del puntero o -1 si fracasa (terminal, pipe, socket o FIFO `errno=EPIPE`).

Descripción:

- Coloca el puntero de acceso asociado a *fd*
- La nueva posición, que no puede ser negativa, se calcula según el valor de *whence*:
  - SEEK_SET: posición = offset
  - SEEK_CUR: posición = posición actual + offset
  - SEEK_END: posición = final del fichero + offset
- Nos podemos salir del tamaño del fichero (no se aumenta el tamaño)

LSEEK. EJEMPLOS

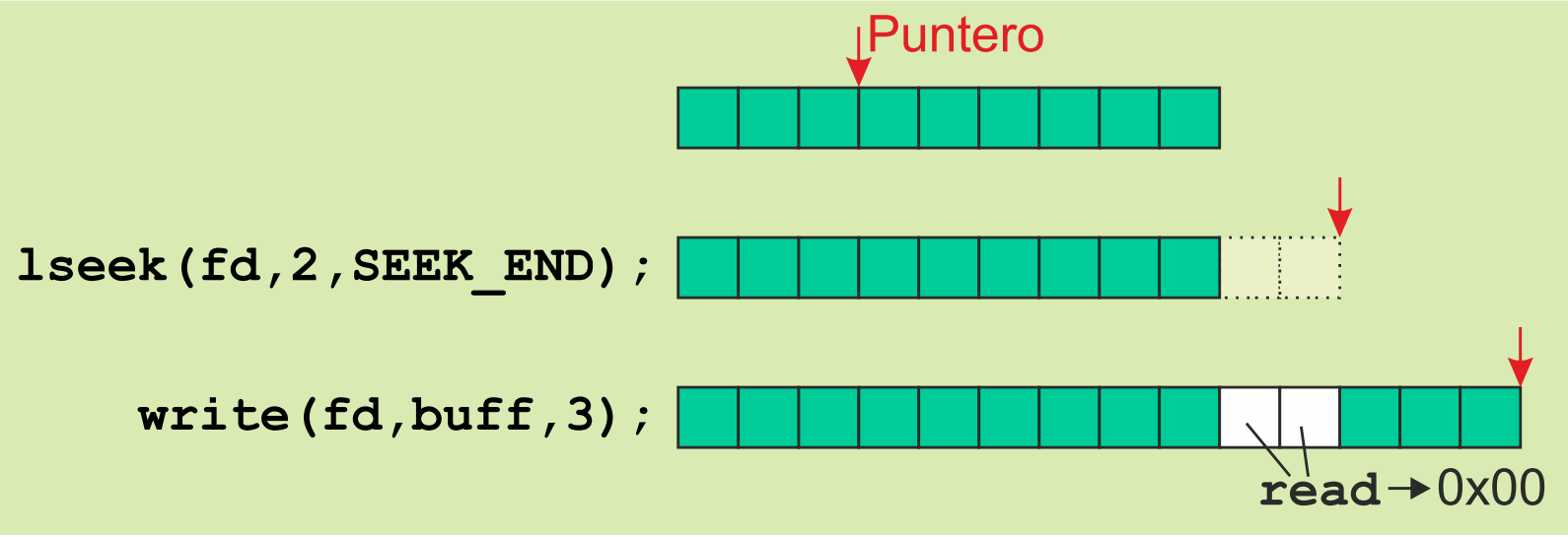
- **Tamaño de fichero:**

```
tam = lseek(fd,0,SEEK_END);
```
← **¿qué pasa si hay huecos?**

- **Posición actual:**

```
pos = lseek(fd,0,SEEK_CUR);
```

- **Creación de huecos:**



- **Al escribir es cuando se aumenta el tamaño real**
- **El tamaño ocupado pude ser < que el real (agrupaciones de hueco no asignadas)**
- **Si se lee del hueco no escrito se obtienen nulos (0x00)**

DUP. DUPLICA UN DESCRIPTOR DE FICHERO

Servicio:

```
int dup(int fd);
```

Argumentos:

- ***fd*: descriptor de fichero**

Devuelve:

- **Un descriptor de fichero que comparte todas las propiedades del fd o -1 si fracasa**

Descripción:

- **Crea un nuevo descriptor de fichero que tiene en común con el anterior:**
  - **Accede al mismo fichero**
  - **Comparte el mismo puntero de posición**
  - **El modo de acceso es idéntico**
- **El nuevo descriptor tendrá el menor valor numérico posible**
- **Se incrementa en 1 el número de referencias (tabla intermedia)**

DUP2. DUPLICA UN DESCRIPTOR DE FICHERO

Servicio:

```
int dup2(int oldfd, int newfd);
```

Argumentos:

- ***oldfd***: descriptor de fichero existente
- ***newfd***: nuevo descriptor de fichero

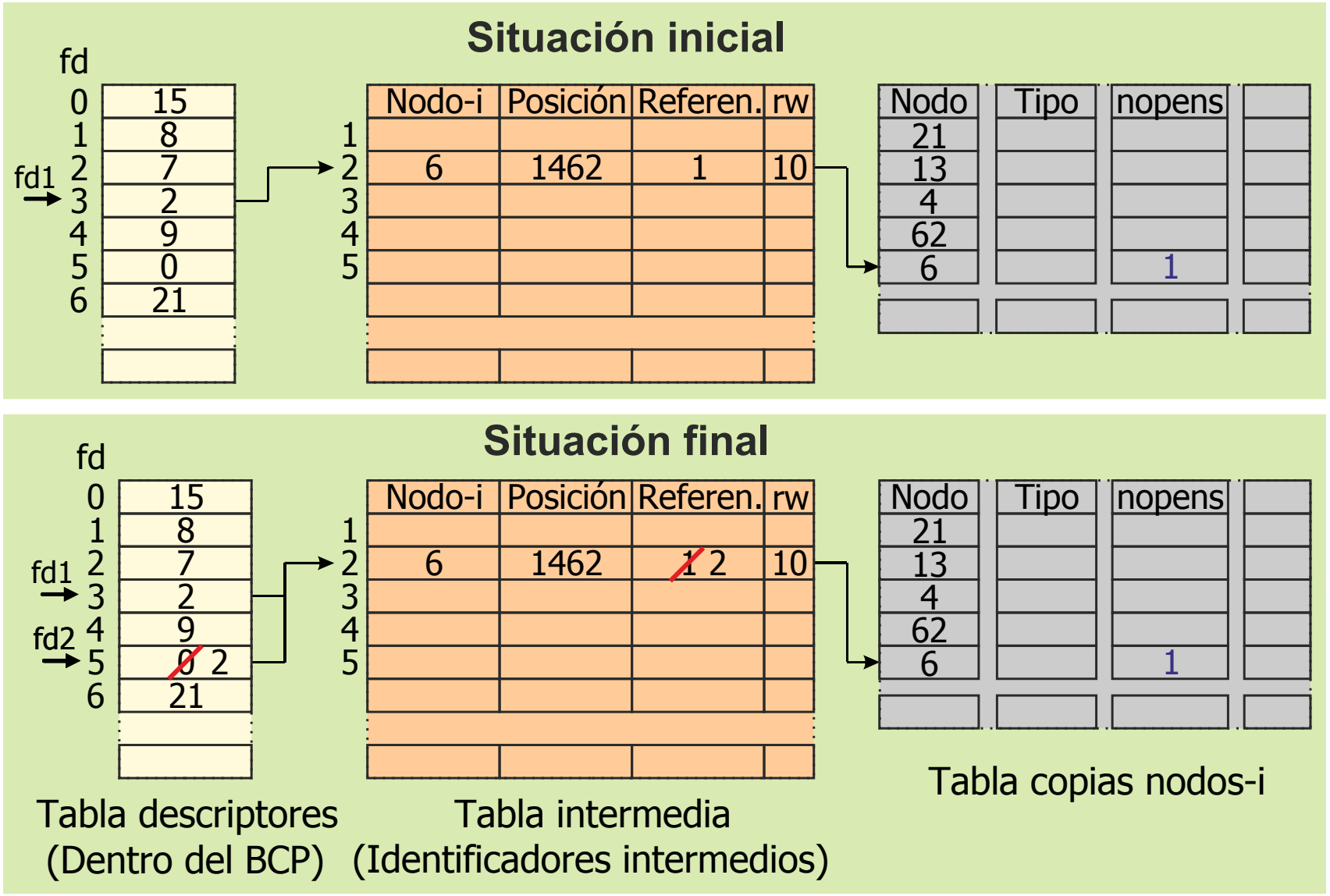
Devuelve:

- El nuevo descriptor de fichero o -1 si fracasa

Descripción:

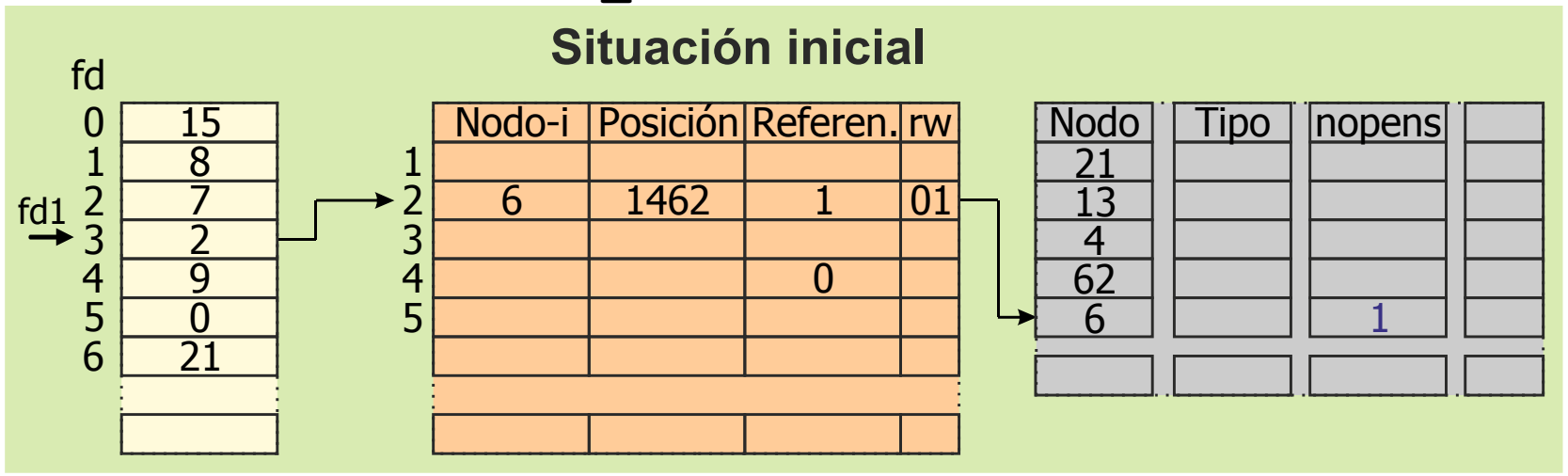
- Crea un nuevo descriptor de fichero, cuyo número es *newfd*, que tiene en común con *oldfd*:
  - Accede al mismo fichero
  - Comparte el mismo puntero de posición
  - El modo de acceso es idéntico
- Si *newfd* estaba abierto, lo cierra antes de realizar el duplicado
- Se incrementa en 1 el número de referencias (tabla intermedia)

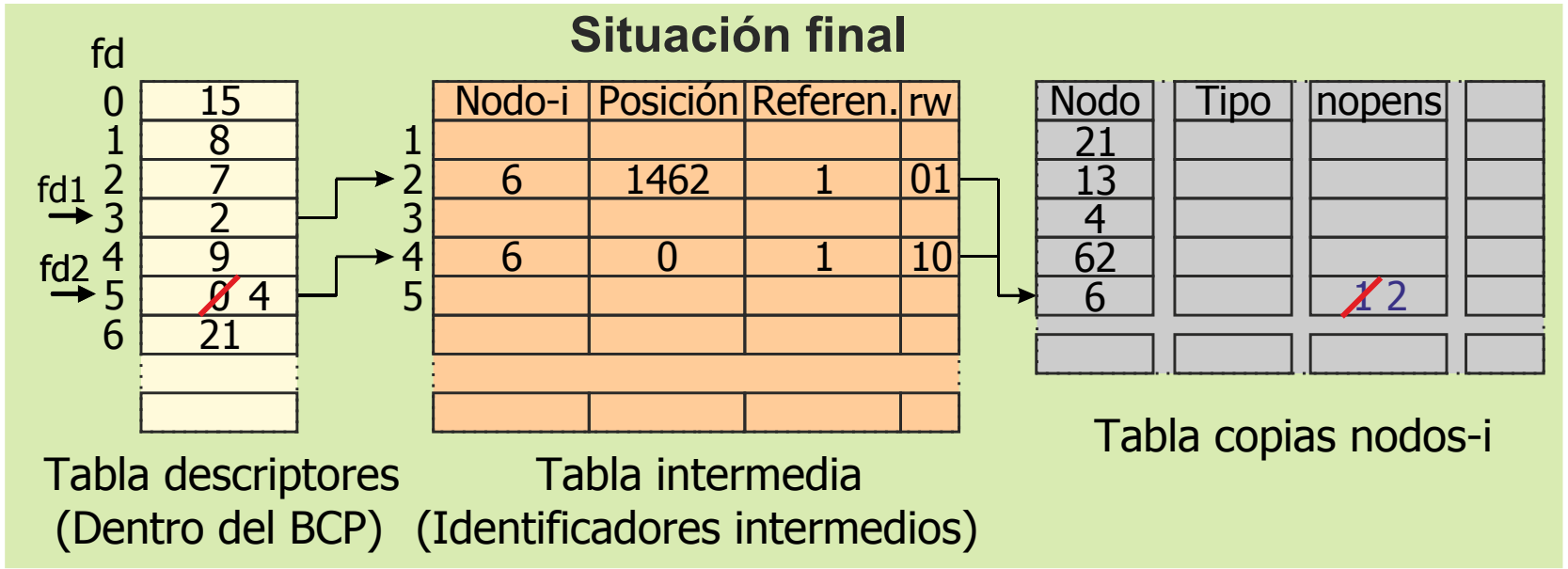
```
fd2= dup(fd1);
```



El mismo proceso vuelve a abrir el mismo fichero datos.txt de nodo-i = 6

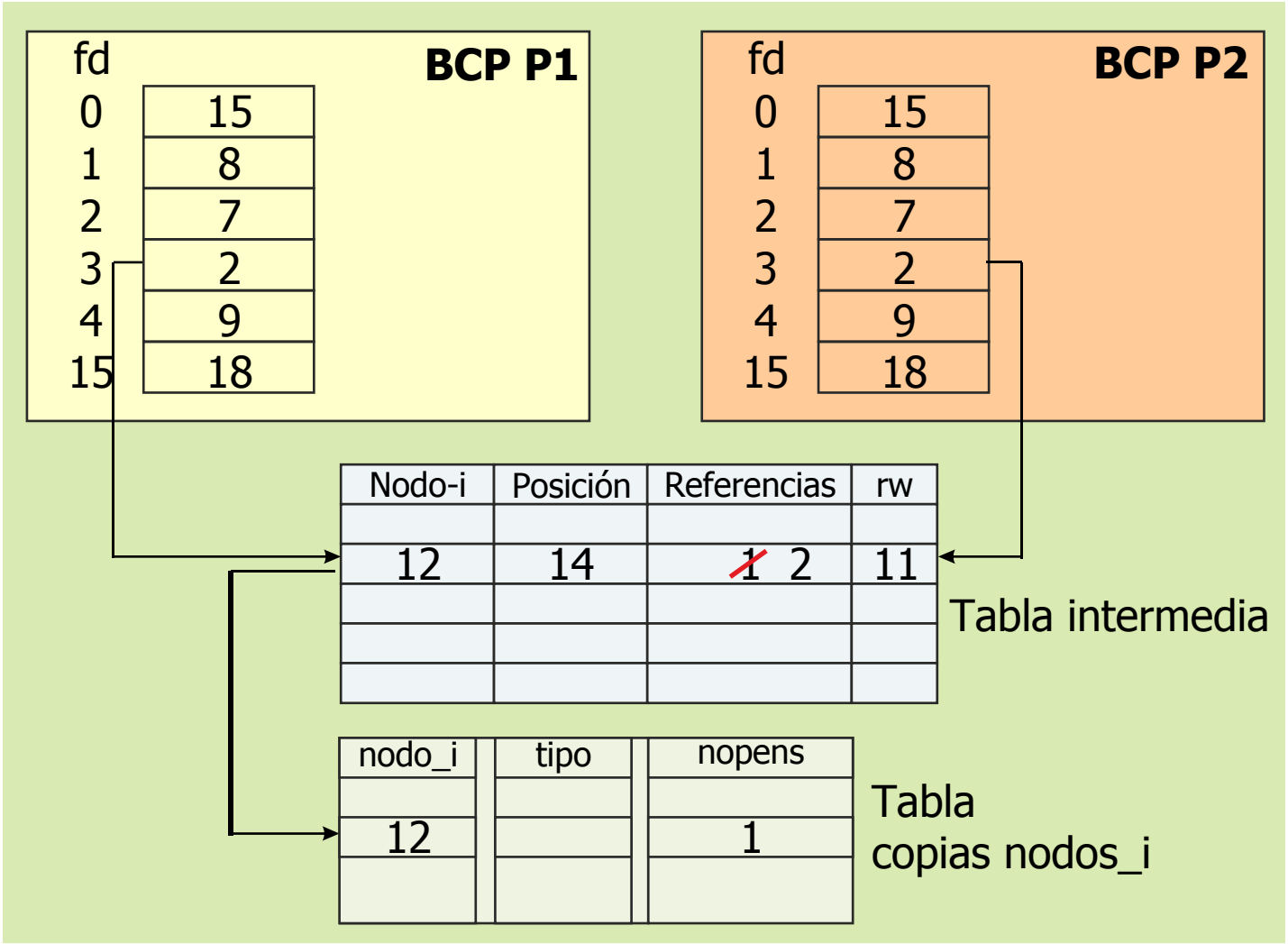
```
fd2 = open("datos.txt",O_RDONLY);
```





GESTIÓN DE PUNTEROS (FORK)

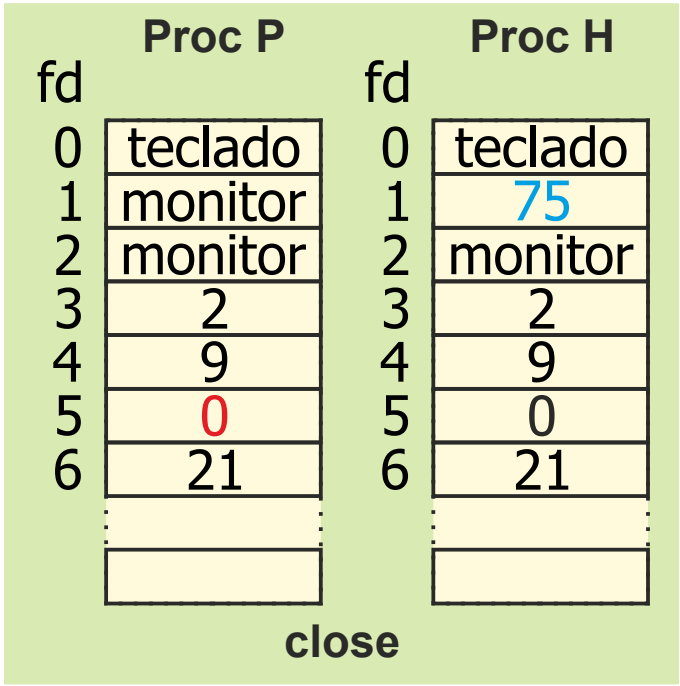

```
pid = fork(); /*crea proceso hijo */
```



EJEMPLO: REDIRECCIONES

Programa que ejecuta: ls > fichero

```
int main(void){
   pid_t pid;
   int status, fd;
   fd = open("fichero", O_WRONLY|O_CREAT|O_TRUNC, 0666);
   if (fd < 0) {perror("open"); exit(1);}
   pid = fork();
   switch(pid) {
      case -1:          /* error */
         perror("fork"); exit(1);
      case 0:           /* proceso hijo ejecuta "ls"
         close(1);
         dup(fd);       /* el descriptor se copia en
         close(fd);
         execlp("ls","ls",NULL);
         perror("execlp"); exit(1);
      default:          /* proceso padre */
         close(fd);
         while (pid != wait(&status));
   }
   return 0;
}
```



EJEMPLO: REDIRECCIONES

Programa que ejecuta: ls > fichero

```
int main(void){
   pid_t pid;
   int status, fd;
   fd = open("fichero", O_WRONLY|O_CREAT|O_TRUNC, 0666);
   if (fd < 0) {
      perror("open"); exit(1);
   }
   pid = fork();
   switch(pid) {
      case -1:          /* error */
         perror("fork"); exit(1);
      case 0:           /* proceso hijo ejecuta "ls" */
         dup2(fd, 1); /* el descriptor se copia en el 1 */
         close(fd);
         execlp("ls","ls",NULL);
         perror("execlp");
         exit(1);
      default:          /* proceso padre */
         close(fd);
         while (pid != wait(&status));
   }
   return 0;
}
```

SERVICIOS UNIX SOBRE DIRECTORIOS

OPENDIR. ABRE UN DIRECTORIO

El fichero directorio es un fichero de registros de tipo `struct dirent`. Esta estructura se define en `dirent.h` y es dependiente de la implementación, pero incluye:

- **`ino_t d_ino`: número de nodo_i**
- **`ino_t d_ino`: nombre del fichero (de tamaño varible)**

Servicio:

```
DIR *opendir(char *dirname);
```

Argumentos:

- ***dirname*: Nombre del directorio**

Devuelve:

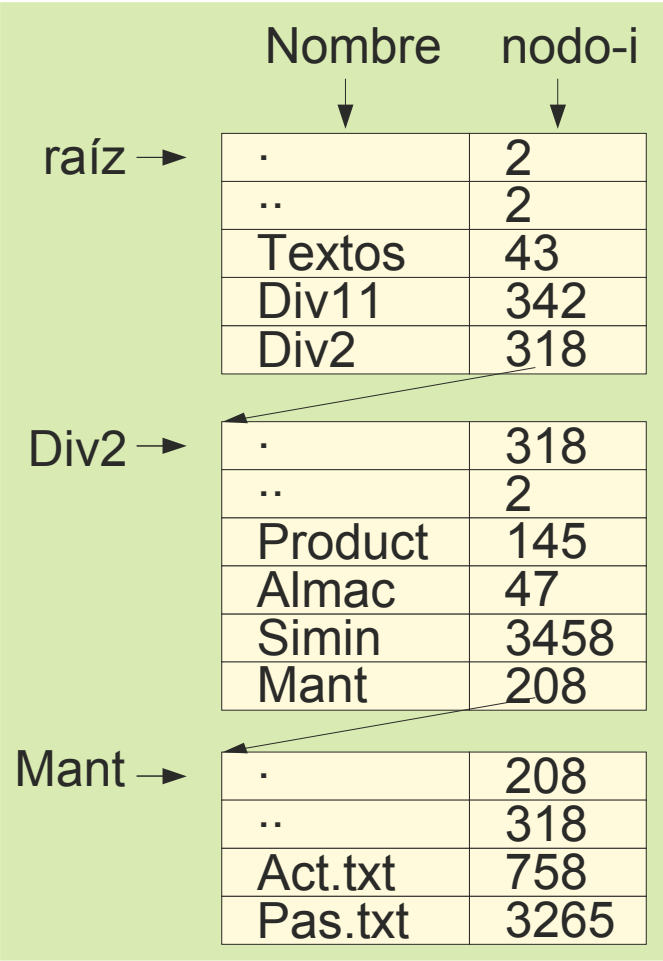
- **Un puntero para utilizarse en `readdir()`, `rewindir()` o `closedir()` y `NULL` si fracasa**

Descripción:

- **Abre un directorio y se coloca en el primer registro o entrada del mismo**

Permisos:

- **Permiso de lectura de `dirname`**



READDIR. LECTURA DE UN REGISTRO DE DIRECTORIO

Servicio:

```
struct dirent *readdir(DIR *dirp);
```

Argumentos:

- ***dirp***: puntero devuelto por `opendir()`

Devuelve:

- Un puntero a un objeto del tipo `struct dirent` que representa un registro de directorio o `NULL` si hubo error o se ha llegado al final del directorio

Descripción:

- Devuelve el siguiente registro del directorio `dirp`
- Avanza el puntero al siguiente registro

CLOSEDIR. CIERRA UN DIRECTORIO

Servicio:

```
int closedir(DIR *dirp);
```

Argumentos:

- ***dirp***: puntero devuelto por `opendir()`

Devuelve:

- Cero o -1 si fracasa

Descripción:

- Cierra la asociación entre `dirp` y la secuencia de entradas de directorio

REWINDIR. INICIALIZA EL PUNTERO DEL DIRECTORIO

Servicio:

```
void rewindir(DIR *dirp);
```

Argumentos:

- ***dirp*** **puntero devuelto por** `opendir()`

Descripción:

- **Sitúa el puntero de posición del directorio en el primer registro o entrada**

MKDIR. CREA UN DIRECTORIO

Servicio:

```
int mkdir(char *name, mode_t mode);
```

Argumentos:

- *name*: nombre del directorio
- *mode*: bits de protección

Devuelve:

- Cero o -1 si fracasa

Descripción:

- Crea un directorio de nombre `name` y permisos `mode & ~umask`
- UID_dueño directorio = UID_efectivo proceso
- GID_dueño directorio = GID_efectivo proceso

Permisos

- Escritura en directorio donde se añade `name`

RMDIR. BORRA UN DIRECTORIO

Servicio:

```
int rmdir(char *name);
```

Argumentos:

- ***name***: **nombre del directorio**

Devuelve:

- **Cero o -1 si fracasa**

Descripción:

- **Borra el directorio si está vacío**
- **Si el directorio no está vacío no se borra**

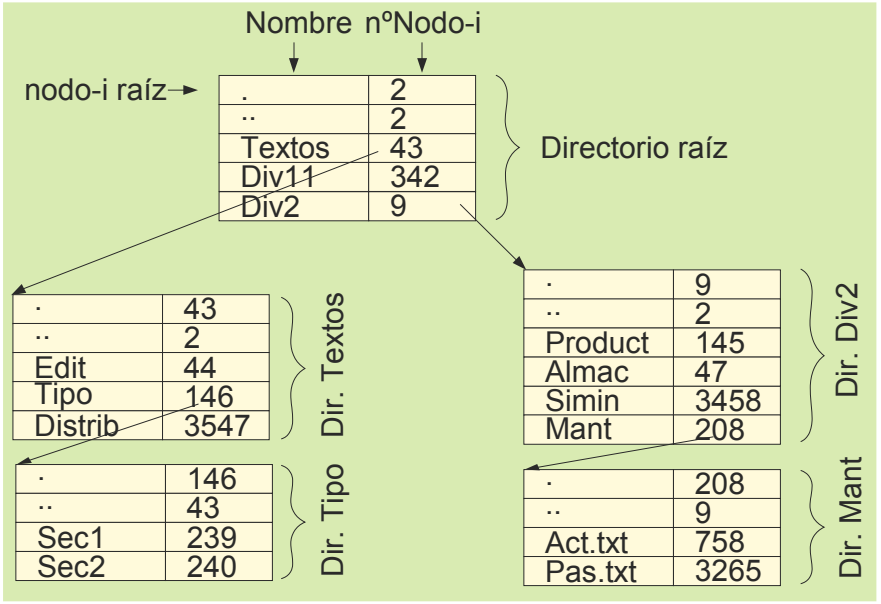
Permisos

- **Escritura en directorio de donde se elimina `name`**

ENLACES

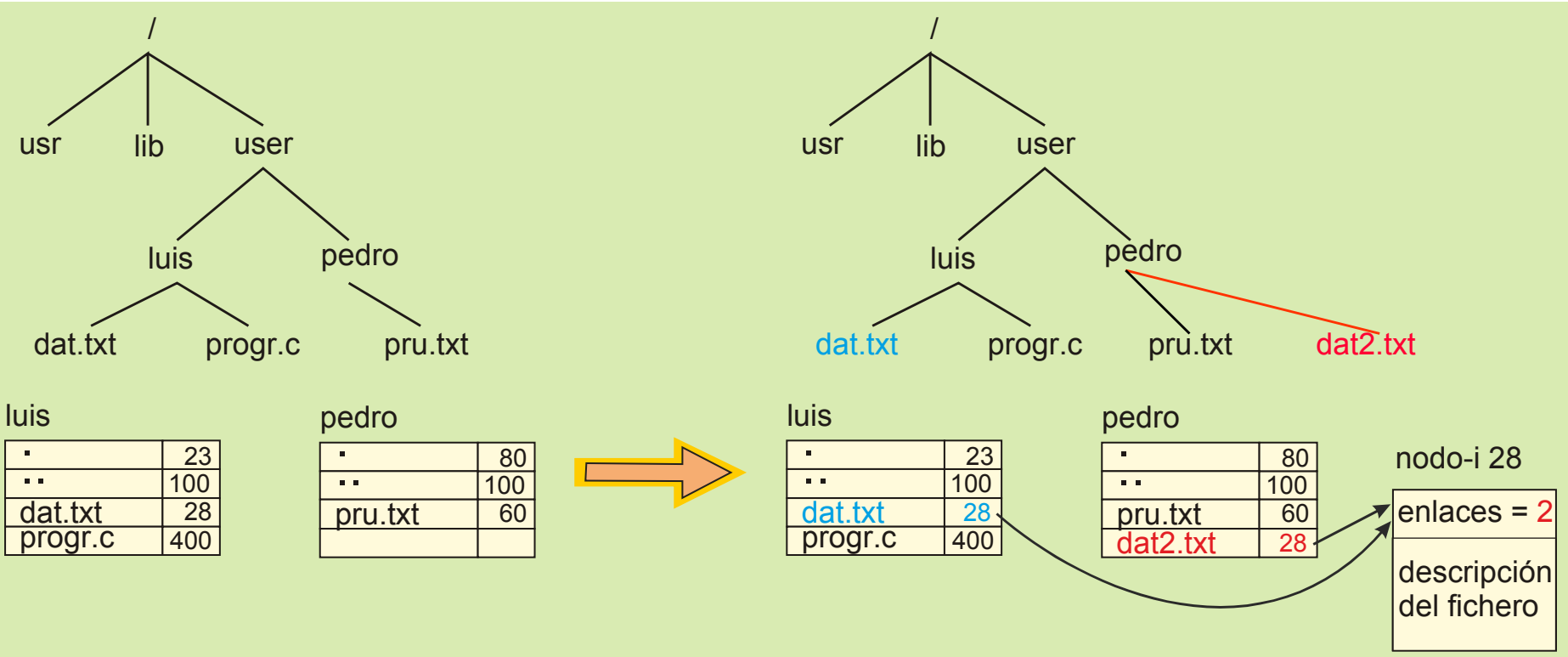
Permite que dos o más nombres hagan referencia al mismo fichero

- **Enlace físico**
  - **El fichero sólo se elimina cuando se borran todos sus nombres**
  - **Contador en el nodo-i del fichero. Necesario para saber cuando se pude borrar un fichero. Con contador = 0 se borra el fichero**
  - **Sólo se permiten enlazar ficheros y subdirectorios del mismo volumen**
  - **No perder de vista que los nombres. y .. también cuentan. ¿Cuántos enlaces tiene un directorio?**
- **Enlace simbólico**
  - **El fichero se elimina cuando el contador de enlaces físicos = 0. Si permanece el enlace simbólico se provocará un error al tratar de abrirlo**
  - **Se puede hacer con ficheros y directorios**
  - **Se puede hacer entre ficheros de diferentes volúmenes**



ENLACE FÍSICO

- **Dos entradas de directorio al mismo nodo-i**
- **Solamente para ficheros de un mismo SF**
- **Hay que incrementar el contador de enlaces en el nodo-i**



Mandato: `ln /user/luis/dat.txt /user/pedro/dat2.txt`

Servicio: `link ("/user/luis/dat.txt","/user/pedro/dat2.txt");`

LINK. CREA UNA ENTRADA (ENLACE) DE DIRECTORIO

Servicio:

```
int link(char *existing, char *new);
```

Argumentos:

- ***existing***: nombre del fichero existente
- ***new***: nombre de la nueva entrada que será un enlace al fichero existente. No puede existir previamente

Devuelve: Cero o -1 si fracasa

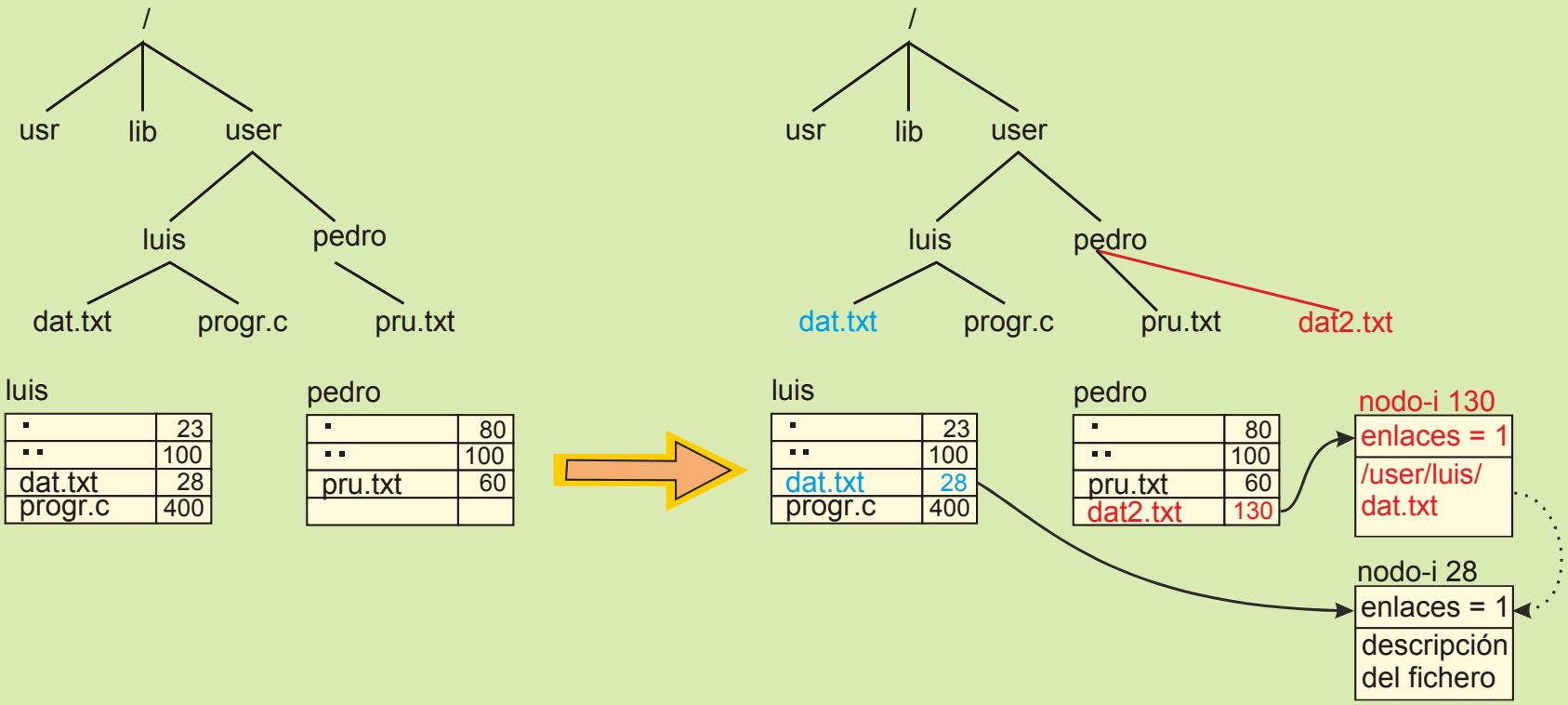
Descripción:

- **Crea un nuevo enlace físico para un fichero existente**
- **Requiere permisos de búsqueda para `existing` y de escritura en el directorio donde esté `new`**
- **El sistema no registra cuál es el enlace original**
- **`existing` no debe ser el nombre de un directorio salvo que se tenga privilegio suficiente y la implementación soporte el enlace de directorios. El enlace de directorios puede dar lugar a ciclos en el árbol de nombres**
- **Las entradas . y .. son enlaces físicos de directorio, pero controlados por el sistema operativo**

El `open` y `creat` también pueden crear una entrada de directorio

ENLACE SIMBÓLICO

- Para ficheros y directorios de un mismo o diferentes SF
- Se utiliza un nodo-i para almacenar la referencia
- No se incrementa el contador de enlaces del fichero
- Para abrir un fichero se comprueban los derechos del camino definido en el enlace simbólico



Mandato: `ln -s /user/luis/dat.txt /user/pedro/dat2.txt`

Servicio: `symlink ("/user/luis/dat.txt","/user/pedro/dat2.txt");`

SYMLINK. CREA UNA ENTRADA DE DIRECTORIO

Servicio:

```
int symlink(char *existing, char *new);
```

Argumentos:

- ***existing***: nombre del fichero existente (no se comprueba que exista)
- ***new***: nombre de la nueva entrada que será un enlace al fichero existente. Es necesario tener permisos de escritura en el directorio

Devuelve:

- Cero o -1 si fracasa

Descripción:

- Crea un nuevo enlace simbólico físico para un fichero o directorio existente
- Requiere permiso de escritura en el directorio donde esté `new`
- El fichero o directorio puede desaparecer dejando el enlace "colgado"
- Se pueden crear ciclos en el árbol de directorio, lo que da lugar a un error de acceso
- Para abrir un fichero se comprueban también los derechos de todo el camino definido en el enlace simbólico
- Una vez creado el enlace, la gran mayoría de los servicios que utilicen `new` como argumento se realizarán realmente sobre `existing`. Por ejemplo el `chmod` se hará sobre `existing`

UNLINK. ELIMINA UNA ENTRADA DEL DIRECTORIO

Servicio:

```
int unlink(char *name);
```

Argumentos:

- ***name***: nombre de fichero

Devuelve:

- Cero o -1 si fracasa

Descripción:

- Elimina la entrada del directorio y decrementa el número de enlaces del fichero correspondiente
- Cuando el número de enlaces es igual a cero:
  - Si ningún proceso lo mantiene abierto, se libera el espacio y nodo_i ocupado por el fichero
  - Si algún proceso lo mantiene abierto, se conserva el fichero hasta que lo cierren todos
- Si se hace sobre un enlace simbólico se decrementa el nº de enlaces del nodo_i simbólico. Cuando llega a 0 se libera dicho nodo_i. No se hace nada sobre el fichero enlazado
- Permiso de escritura en directorio de donde se elimina `name`

CHDIR. CAMBIA EL DIRECTORIO ACTUAL

Servicio:

```
int chdir(char *name);
```

Argumentos:

- ***name***: **nombre de un directorio**

Devuelve:

- **Cero o -1 si fracasa**

Descripción:

- **Modifica el directorio actual, aquel a partir del cual se forman los nombre relativos**

Permisos:

- **Requiere permisos de búsqueda para `name`**

RENAME. CAMBIA EL NOMBRE DE UN FICHERO

Servicio:

```
int rename(char *old, char *new);
```

Argumentos:

- ***old***: nombre de un fichero existente
- ***new***: nuevo nombre del fichero

Devuelve:

- Cero o -1 si fracasa

Descripción:

- Cambia el nombre del fichero `old`. El nuevo nombre es `new`

Permisos:

- Permiso de escritura en los directorios que contienen a `old` y a `new`.

GETCWD. OBTIENE EL NOMBRE DEL DIRECTORIO ACTUAL

Servicio:

```
char *getcwd(char *buf, size_t size);
```

Argumentos:

- ***buf***: dirección del espacio donde se va a almacenar el nombre del directorio actual
- ***size***: longitud en bytes de dicho espacio

Devuelve:

- Puntero a `buf` o `NULL` si fracasa (p.e. si el nombre ocupa más de `size`)

Descripción:

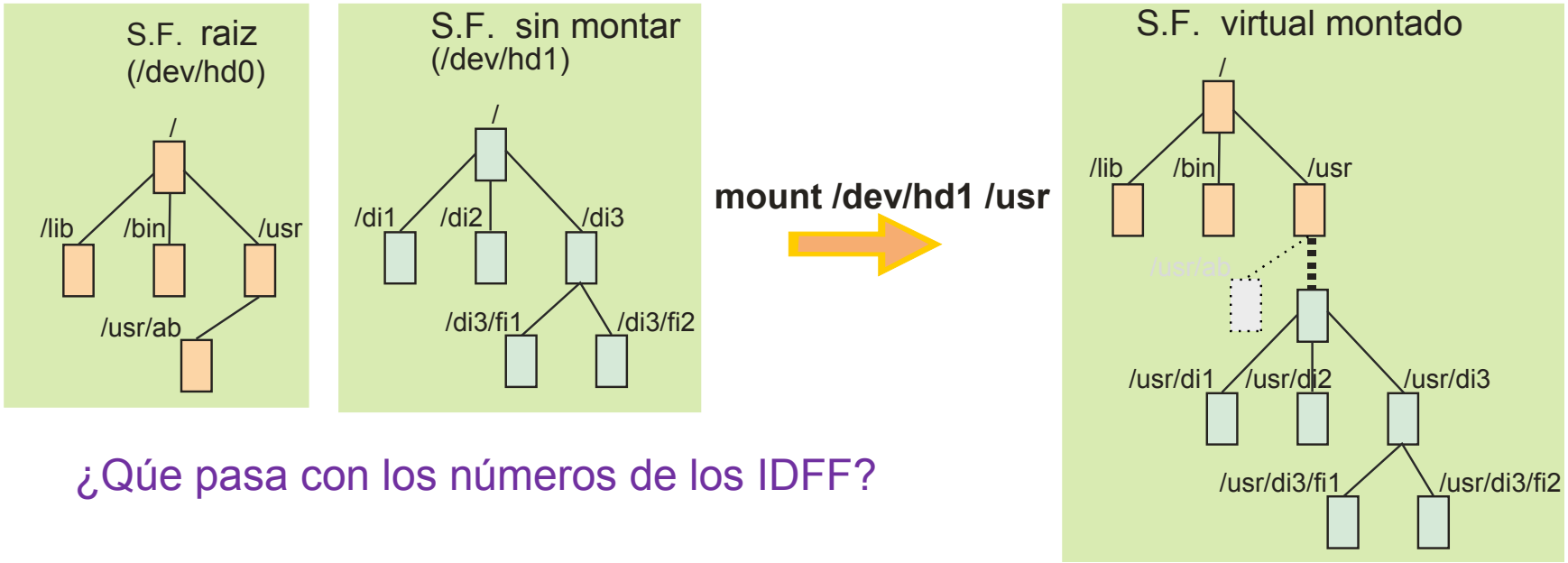
- Obtiene el nombre del directorio actual

PROGRAMA QUE LISTA UN DIRECTORIO


```
#define MAX_BUF   256
int main(int argc, char **argv){
    DIR *dirp;
    struct dirent *dp;
    char buf[MAX_BUF];
    /* imprime el directorio actual */
    getcwd(buf, MAX_BUF);
    printf("Directorio actual: %s\n", buf);
    /* abre el directorio pasado como argumento */
    dirp = opendir(argv[1]);
    if (dirp == NULL)  {
        fprintf(stderr,"No puedo abrir %s\n", argv[1]);
    } else {
        /* lee entrada a entrada */
        while ((dp = readdir(dirp)) != NULL)
            printf("%s\n", dp->d_name);
        closedir(dirp);
    }
   return 0;
}
```

MONTADO DE VOLÚMENES O PARTICIONES

- **Añade a un árbol de directorios los directorios de un dispositivo. Creando un sistema de ficheros virtual conjunto**
- **El montado es una operación realizada en memoria. No se modifica nada en los discos**
- **Oculta el nombre del dispositivo físico o partición que se monta**



¿Qúe pasa con los números de los IDFF?

- **Si en el directorio de montaje existen ficheros o directorios éstos quedan inaccesibles. (el directorio `/usr/ab` ya no está disponible). Una vez desmontado vuelven a ser accesibles.**
- **En UNIX `/etc/fstab` contiene los sistemas de ficheros disponibles con las opciones de montado para el mandato mount**

MOUNT. MONTA UN SISTEMA DE FICHEROS

Servicio:

```
int mount(const char *source, const char *target,
          const char *filesystemtype,
          unsigned long mountflags, const void *data);
```

Argumentos:

- ***source*** **: sistema de ficheros que se monta, generalmente un dispositivo como** `/dev/cdrom`**,** `/dev/hdb0`**,** `/dev/sda1` **o** `/home/imagen.iso`
- ***target*** **: directorio sobre el que se monta**
- ***filesystemtype*** **: tipo de sistema de ficheros, como "minix", "ext2", "ext3", "ext4", "msdos", "vfat","proc" o "nfs"**
- ***mountflags*** **: Opciones como:**
  - **MS_NOEXEC enmascara los bits de ejecución del sistema montado**
  - **MS_NOSUID enmascara los bits SUID y SGID del sistema montado**
  - **MS_RDONLY montado para lectura solamente**
- ***data*** **: opciones que dependen del tipo de sistema de ficheros.**

Devuelve:

Cero o -1 si fracasa

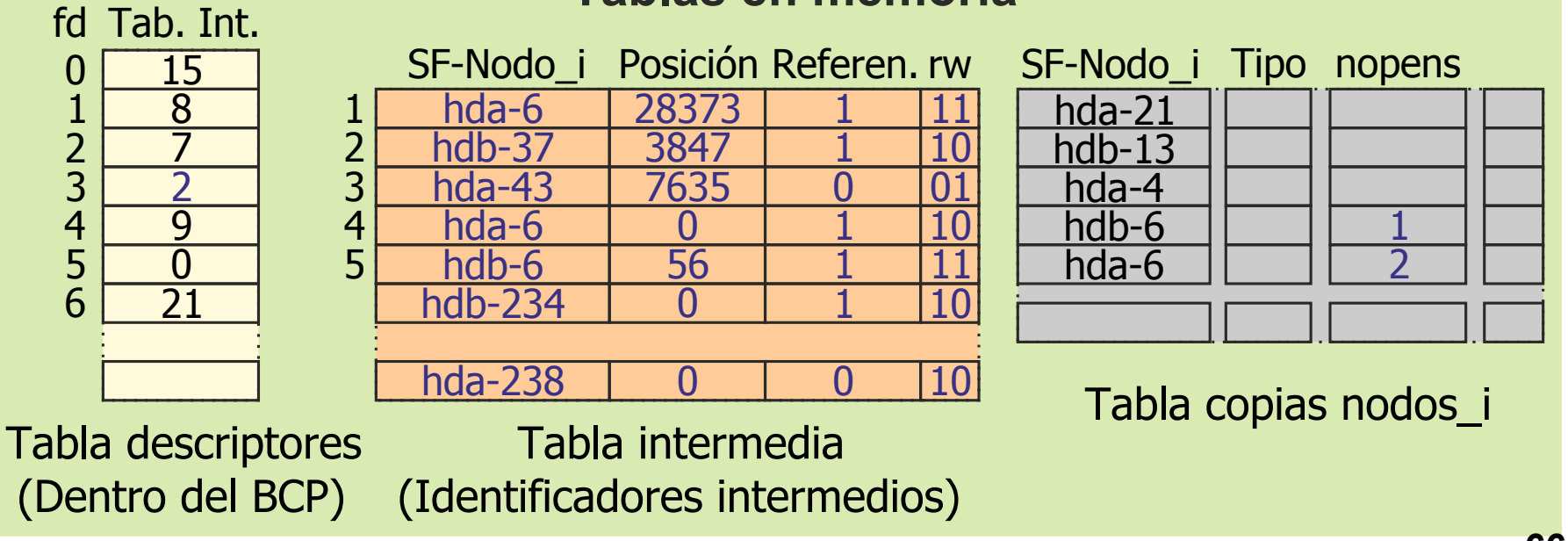
Descripción:

- **Monta un sistema de ficheros**
- **Requiere privilegios de superusuario**

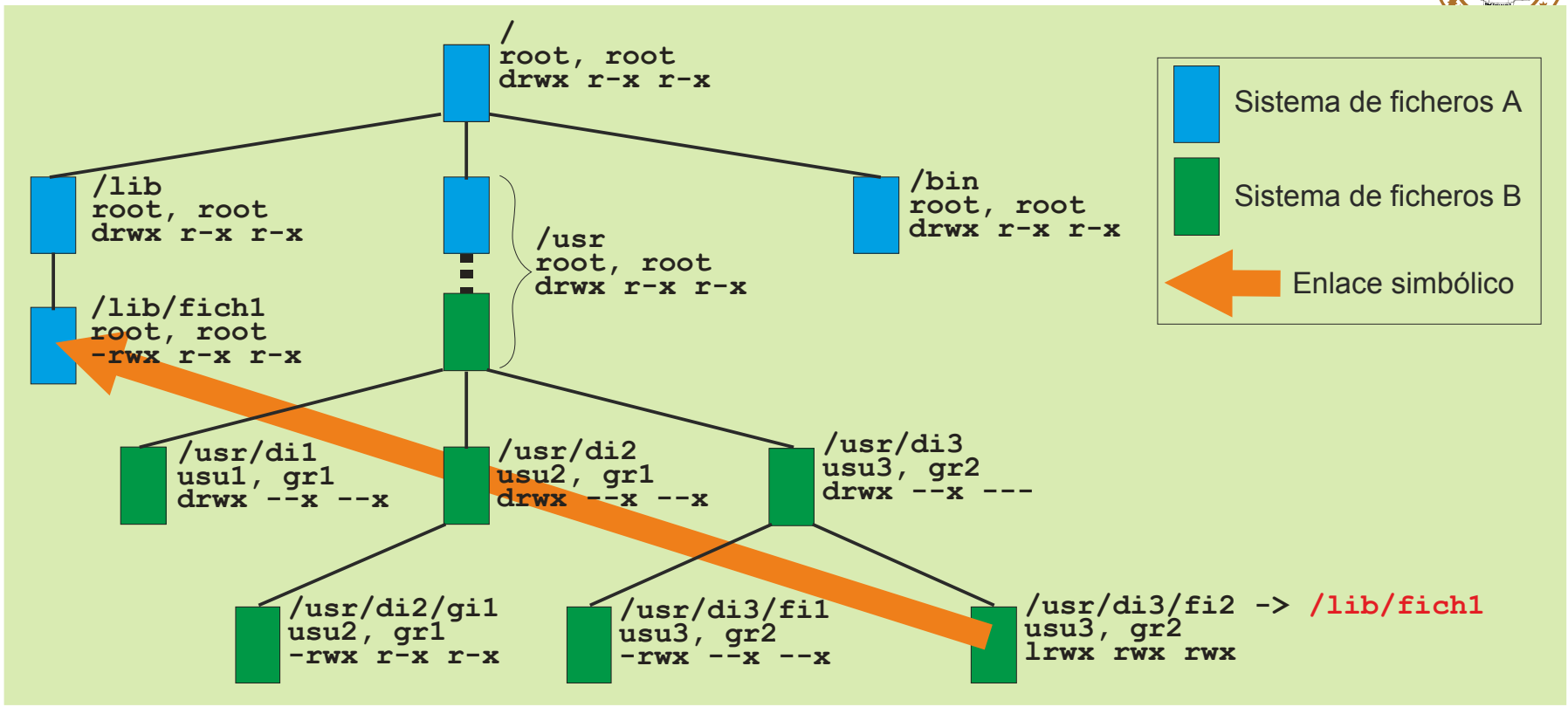
No confundir el servicio mount con el mandato mount

- Cada sistema de ficheros numera los nodos_i empezando en el 2
- Al montar sistemas de ficheros, los nº de nodo_i no son únicos
- Para identificar un nodo_i es necesario especificar el sistema de ficheros y en nº de nodo_I dentro de él. Esto es necesario, por ejemplo, en la tabla intermedia y en la tabla de copias de nodos_i
- Si en el directorio de montaje existen ficheros o directorios estos quedan inaccesibles

Tablas en memoria



UNIX: PERMISOS EN VOLÚMENES MONTADOS




```
$ ls -l "/usr/di3/fi2"
lrwxr-xr-x root root  10 Mar 11 2004 /usr/di3/fi2 -> /lib/fich1
```

Para abrir `/usr/di3/fi2` se comprueban los permisos de:

"/" (raíz SF A), "usr/" (raíz SF B), "di3/", "/" (raíz SF A), "lib/", "fich1"

(no se comprueban los permisos del nodo_i de "fi2" ni del nodo_i original de "usr/")

UMOUNT. DESMONTA UN SISTEMA DE FICHEROS

Servicio:

```
int umount(const char *target);
```

Argumentos:

- ***target*** **: directorio que se desmonta**

Devuelve:

- **Cero o -1 si fracasa**

Descripción:

- **Desmonta un sistema de ficheros**
- **Si hay ficheros abiertos no desmonta y devuelve error**
- **Requiere privilegios de superusuario**

SERVICIOS UNIX SOBRE DIRECTORIOS
PROTECCIÓN

PROTECCIÓN UNIX SOBRE UN FICHERO O DIRECTORIO

Los procesos se ejecutan con

- UID real
- UID efectivo (se utiliza para los permisos)
- GID real
- GID efectivo (se utiliza para los permisos)

Bits SETUID y SETGID

- Si un proceso ejecuta un fichero con el SETUID activo, el UID efectivo del proceso pasa a ser el UID del propietario del fichero
- Si un proceso ejecuta un fichero con el SETGID activo,el GID efectivo del proceso pasa a ser el GID del propietario del fichero

Reglas de protección:

- Si UID efectivo = 0 se concede el acceso (es superusuario)
- Si UID efectivo = UID del propietario, se utiliza el primer grupo de bits
- Si GID efectivo = GID del propietario, se utiliza el segundo grupo de bits
- En caso contrario, se utiliza el último grupo de bits

EJEMPLOS DE VOLCADO DEL MANDATO ls


```
drwxr-x---    2 pepito prof    48 Dec 26  2001 News
drwxr-xr-x    2 pepito prof    80 Sep 29  2004 bin
lrwxrwxrwx    1 root root       3 Jan 23 18:34 lvremove -> lvm
lrwxrwxrwx    1 root root       3 Jan 23 18:34 lvrename -> lvm
drwxrwxrwt   16 root root    1928 Apr  9 20:26 tmp
-rwxr-xr-x    1 root root    2436 Dec 26  2001 termwrap
-rwsr-xr-x    1 root root   22628 Jan  5 10:15 mount.cifs
```

Carácter inicial:

-	fichero normal
d	directorio
l	enlace simbólico
b	dispositivo de bloques
c	dispositivo de caracteres
p	FIFO
s	socket UNIX

Bit x:

- Si aparece una “s” significa que está activo el SETUID o SETGID, según la posición.
- Si aparece una “t” en un directorio se permite crear y borrar entradas con el UID efectivo del proceso (se usa para directorios temporales). No se podrán borrar las entradas de otros usuarios.

ACCESS. ACCESIBILIDAD SOBRE UN FICHERO

Servicio:

```
int access(char *name, int amode);
```

Argumentos:

- **name: nombre del fichero o directorio**
- **amode: modo de acceso que se quiere comprobar. `amode` es el `OR` inclusivo de `R_OK`, `W_OK`, `X_OK` o `F_OK` (comprobar existencia)**

Devuelve:

- **0 si el proceso tiene acceso al fichero (para lectura, escritura o ejecución) o -1 en caso contrario**

Descripción:

- **Utiliza el UID real y el GID real (en lugar de los efectivos) para comprobar los derechos de acceso sobre un fichero**

Ejemplos:

- `access("fichero",F_OK);` **devuelve 0 si el fichero existe o -1 si no existe**
- `access("fichero",R_OK|W_OK);` **devuelve 0 si el proceso que ejecuta la llamada tiene permisos de acceso de lectura y escritura sobre el fichero (utilizando el UID real y el GID real)**

ACCESS. Ejemplo

Ejemplo de uso `access` para que un programa que tenga activo el bit de SETUID o el de SETGID.

Antes de abrir un fichero se comprueba si el usuario real tiene derechos de acceso. Solamente se abrirá en dicho caso.

```
if (0 == access("archivo", R_OK|W_OK)) {
   /* Se realiza la apertura del fichero */
   fd7 = open ("archivo", O_RDWR);
   ......
} else {
   /* El acceso es negativo, por lo que no hace el open */
   ......
}
```

UMASK. ASIGNA LA MASCARA DE CREACIÓN

Servicio:

```
mode_t umask(mode_t cmask);
```

Argumentos:

- ***cmask***: bits de permiso a desasignar en la creación de ficheros

Devuelve:

- Devuelve la máscara previa

Descripción:

- Asigna la máscara de creación de ficheros del proceso que la invoca
- Los bits activos en la máscara son desactivados en la palabra de protección del fichero (permisos = `mode & ~umask`)
  - Si máscara = 022 , y se crea un fichero con permisos 0777 , los permisos con los que se crea realmente el fichero son 0755
- La máscara está almacenada en el BCP del proceso

CHMOD. CAMBIA LOS BITS DE PERMISO

Servicio:

```
int chmod(char *name, mode_t mode);
```

Argumentos:

- ***name*: nombre del fichero objetivo**
- **mode: Nuevos bits de protección**

Devuelve:

- **Cero o -1 si fracasa**

Descripción:

- **Modifica los bits de permiso y los bits SETUID y SETGID del fichero**
  - **S_ISUID = 04000**
  - **S_ISGID = 02000**
- **Sólo el propietario del fichero o el administrador pueden cambiar estos bits**
- **NO se utiliza la máscara**

int chmod(char *name, mode_t mode);

Servicio:

```
int chown(char *name, uid_t owner, gid_t group);
```

Argumentos:

- ***name***: nombre del fichero
- ***owner***: nuevo propietario del fichero
- ***group***: nuevo identificador de grupo del fichero

Devuelve:

- Cero o -1 si fracasa

Descripción:

- Modifica el identificador de usuario y de grupo del fichero
- Los bits SETUID y SETGID son borrados
- Sólo el propietario del fichero o el administrador pueden cambiar estos atributos

SERVICIOS UNIX SOBRE ATRIBUTOS

STAT, FSTAT. INFORMACIÓN SOBRE UN FICHERO

Servicios:

```
int stat(char *name, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(char *name, struct stat *buf);
```

Argumentos:

- ***name***: nombre del fichero
- ***fd***: descriptor de fichero
- ***buf***: puntero a un objeto de tipo `struct stat` donde se almacenará la información del fichero
- `lstat` se diferencia del `stat` en que devueve el estado del propio enlace simbólico y no del fichero al que apunta dicho enlace, como hace el `stat`

Devuelve:

- Cero o -1 si fracasa

Descripción:

- Obtiene información sobre un fichero y la almacena en una estructura de tipo `struct stat`

STAT, FSTAT. INFORMACIÓN SOBRE UN FICHERO


```
struct stat {
    mode_t     st_mode;     /* tipo de fichero + permisos */
    ino_t      st_ino;      /* número del fichero  */
    dev_t      st_dev;      /* dispositivo */
    nlink_t    st_nlink;    /* número de enlaces */
    uid_t      st_uid;      /* UID del propietario */
    gid_t      st_gid;      /* GID del propietario */
    off_t      st_size;     /* número de bytes */
    blksize_t  st_blksize;  /* tamaño bloque para I/O */
    blkcnt_t   st_blocks;   /* n° de agrupaciones asignadas */
    time_t     st_atime;    /* último acceso */
    time_t     st_mtime;    /* última modificacion de datos */
    time_t     st_ctime;    /* última modificacion del nodo_i*/
  };
```

Comprobación del tipo de fichero aplicado a st_mode:

`S_ISDIR(s.st_mode)`	Cierto si directorio
`S_ISREG(s.st_mode)`	Cierto si fichero normal
`S_ISLNK(s.st_mode)`	Cierto si enlace simbólico
`S_ISCHR(s.st_mode)`	Cierto si especial de caracteres
`S_ISBLK(s.st_mode)`	Cierto si especial de bloques
`S_ISFIFO(s.st_mode)`	Cierto si pipe o FIFO
`S_ISSOCK(s.st_mode)`	Cierto si socket

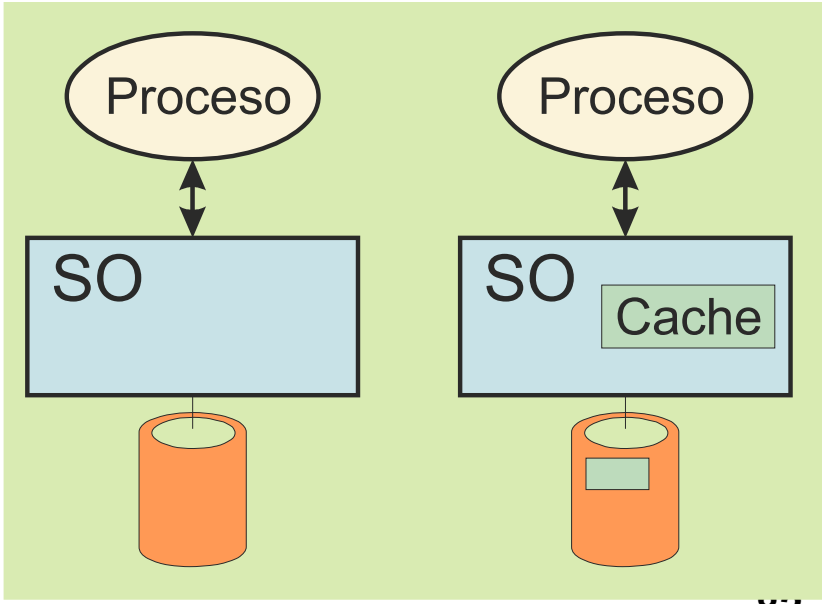
SERVIDOR DE FICHEROS
ASPECTOS DE DISEÑO

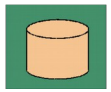
GESTIÓN DE ESPACIO LIBRE

- **Mapas de bits, o vectores de bits: un bit por recurso existente (descriptor de fichero, agrupación). Si el recurso está libre, el valor del bit asociado al mismo es 1, si está ocupado es 0.**
  - **Ejemplo, sea un disco en el que las agrupaciones 2, 3, 4, 8, 9 y 10 están ocupadas y el resto libres, y en el que los descriptores de fichero 2, 3 y 4 están ocupados. Sus mapas de bits de serían:**
    - MB de agrupaciones: 1100011100011....
    - MB de descriptores: 1100011...
  - **Fácil de implementar y sencillo de usar. Eficiente si el dispositivo no está muy lleno o muy fragmentado.**
- **Listas de recursos libres: mantener enlazados en una lista todos los recursos disponibles (agrupaciones o descriptores de ficheros) manteniendo un apuntador al primer elemento de la lista.**
  - **Este método no es eficiente, excepto para dispositivos muy llenos y fragmentados**
  - **La lista puede construirse dentro de las propias agrupaciones libres.**

CACHE DE BLOQUES

- Estructura de datos en memoria con los bloques más frecuentemente utilizados
  - Lecturas adelantadas
  - Limpieza de la cache (sync) significa escritura retardada
- Mejora las prestaciones si hay reutilización de la información (proximidad referencial)
- El tiempo medio de lectura puede ser distinto del de escritura
- Principal problema: fiabilidad del sistema de ficheros por escritura retardada (delayed-write) (30seg en Unix) o diferida (write-back). Solución: técnica de write-through, pero no es eficiente.
- Asociado a la gestión de memoria virtual (bloque cache = página)






UNIVERSIDAD POLITECNICA
MADRID
TECHNICA
IMPENDI
NATIONI

TUBERÍAS

TUBERÍA O PIPE

Servicio:

```
int pipe(int fildes[2]);
```

Argumentos:

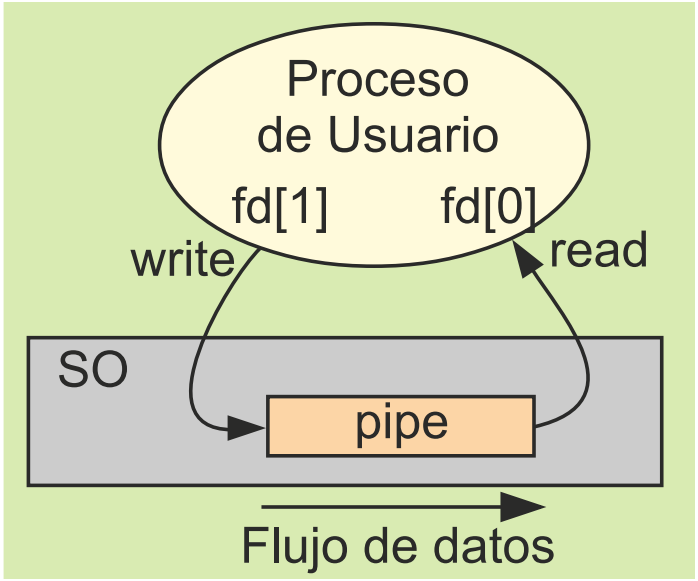
- ***fildes*: Vector de dos descriptores de fichero**
  - **fildes[0]: descriptor de lectura**
  - **fildes[1]: descriptor de escritura**

Devuelve:

- **0 o -1 si fracasa**

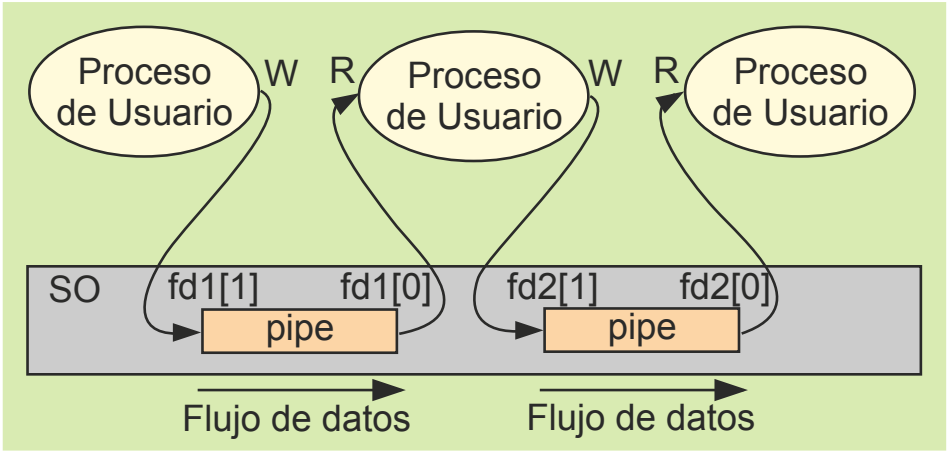
Ejemplo utilización:

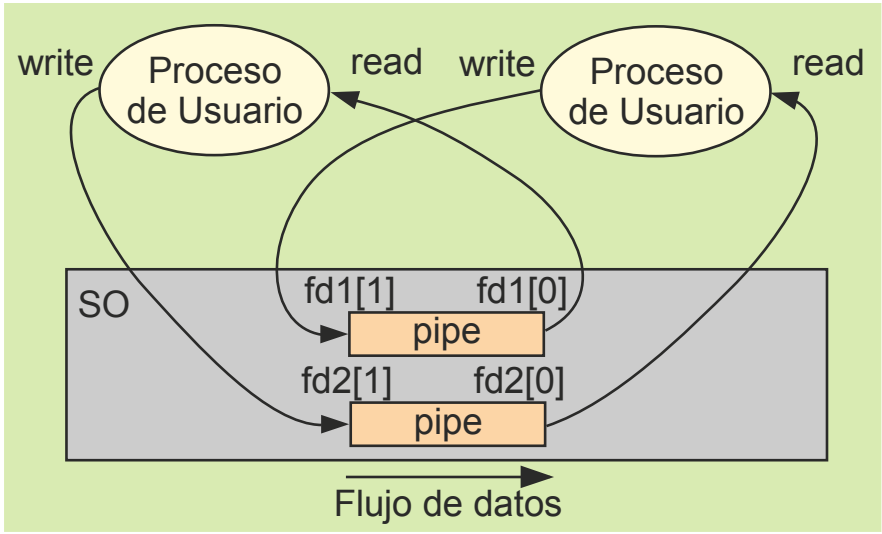
```
int miPipe[2], ret;
ret = pipe(miPipe);
```



TUBERÍA O PIPE

- Mecanismo de comunicación y sincronización sin nombre
- Identificación: dos descriptores de fichero
- Sólo pueden utilizarse entre procesos que crean y heredan los descriptores y dentro de la misma máquina
- En cada proceso deben cerrarse los extremos no utilizados
- Con buffering (Tamaño típico: 4 KB)





SEMÁNTICA DEL READ EN UN PIPE


```
ssize_t read(int fd[0], void *buf, size_t n_bytes);
```

- Pipe vacío → se bloquea el lector
- Pipe con *p* bytes →
  - Si p ≥ n devuelve n
  - Si p < n devuelve p
- Si pipe vacío y no hay escritores (no hay usuarios utilizando fd[1]) devuelve 0
- Lectura atómica (cuidado con tamaños grandes, debido al buffering)

!No dejar abiertos los descriptores NO usados!

SEMÁNTICA DEL WRITE EN UN PIPE


```
ssize_t write(int fd[1], void *buf, size_t n_bytes);
```

- Pipe lleno → se bloquea el escritor
- Si no hay lectores (no hay usuarios utilizando fd[0]) se recibe la señal SIGPIPE
- Escritura atómica (cuidado con tamaños grandes, debido al buffering)

!No dejar abiertos los descriptores NO usados!

ESQUELETO DE EJEMPLO CON PIPES


```
int main(void){
  int fd[2];        /* descriptores del pipe */
  int dato_p[4];    /* datos a producir */
  int dato_c;       /* dato a consumir */
  pipe(fd);
  if (fork() == 0) { /* productor (proceso hijo)*/
    close (fd[0]);
    for(;;){
      /* producir dato_p */
      ......
      write(fd[1], dato_p, 4*sizeof(int));
    }
  } else { /* consumidor (proceso padre)*/
    close (fd[1]);
    for(;;) {
      read(fd[0], &dato_c, sizeof(int));
      /* consumir dato */
      ......
    }
  }
return 0;
}
```