



Sistemas Operativos

Curso
2017-2018

Módulo 2: Gestión de archivos y directorios
Estructuras en memoria



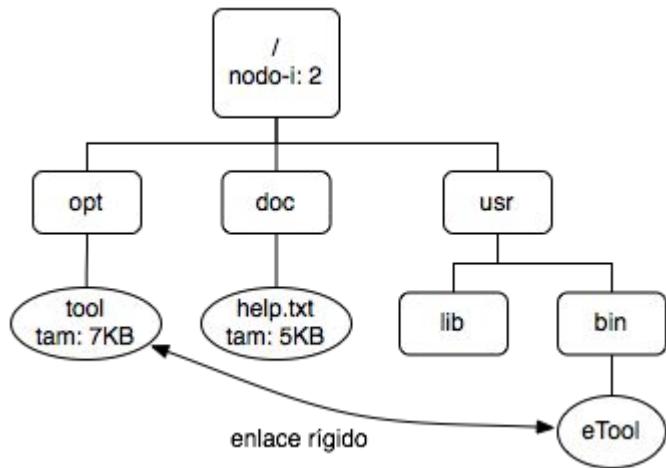
¿Qué sabemos ya?

- Visión lógica del sistema de ficheros
 - API para el programador
 - Visión lógica de un fichero como secuencia de bytes
- Alternativas de organización física en un dispositivo
 - Organización de bloques físicos en disco
 - Contenido de directorios
 - Enlaces simbólicos y rígidos
 - Idea de *montaje* de un sistema de ficheros
 - Información que se mantiene en el dispositivo para cada sistema de ficheros (FAT, nodos-i, superbloque, mapa de bits...)



Entonces, ¿no nos quedan huecos?

- Si es así, prueba con este ejercicio...



Dado este grafo de directorios, indica

- Un posible contenido del disco (nodos-i, bloques de datos...) si cada nodo-i tiene 2 punteros directos y un indirecto y los bloques son de 2KB
- Explica, paso a paso, qué ocurre al ejecutar el siguiente código:

```
char buf[128];
int fd = open("/opt/tool", O_RDONLY);
lseek(fd, 3000, SEEK_CUR);
read(fd, &buf, 100);
write(fd, &buf[8], 1);
```



Ah, pues sí quedaban huecos

- ¿Qué devuelve la llamada *open()*?
 - ¿Y cómo a partir de un entero llegamos al bloque de disco que queremos?
- ¿Qué hace la llamada *lseek()*? ¿Modifica algo en el disco?
- Si el fichero *tool* existe y se creó con permisos de escritura, ¿por qué la llamada a *write()* fallará?



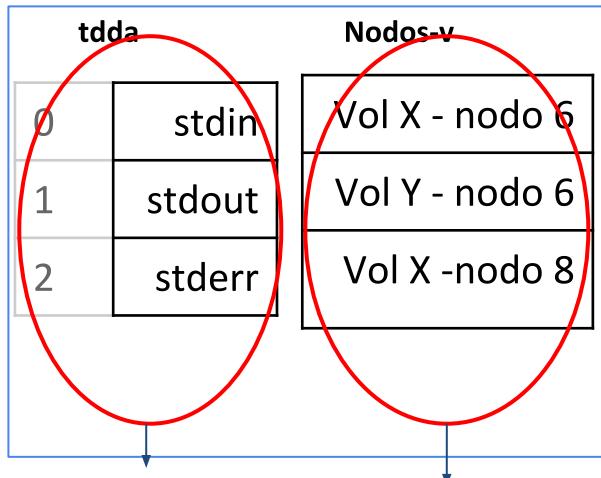
¡ Necesidad de más estructuras !

- ¿Dónde almacenaremos esas nuevas estructuras?
 - ¿En disco? Implicaría que cada apertura (llamada a *open()*) quedara reflejada en disco
 - ¿De verdad esperas que, tras hacer *open()* y apagar el ordenador, la apertura del fichero esté ahí cuando vuelvas a encender?
 - ¿Memoria principal? Suena bien...
 - ¿Alguna idea de qué puede representar ese entero que devuelve *open()*?



Tabla de descriptores de ficheros abiertos

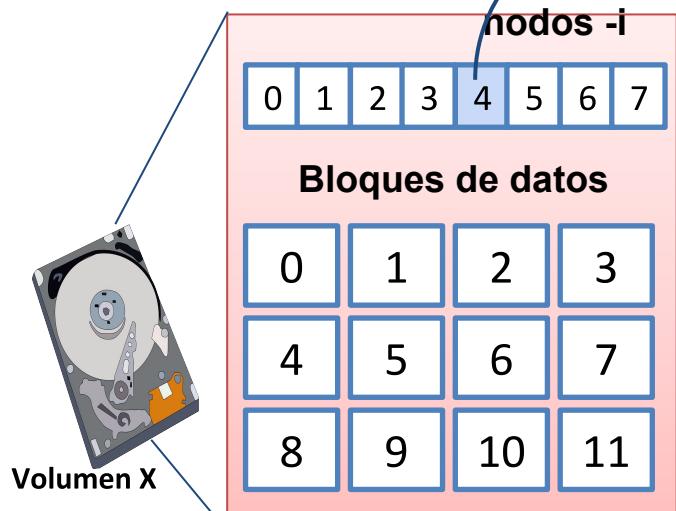
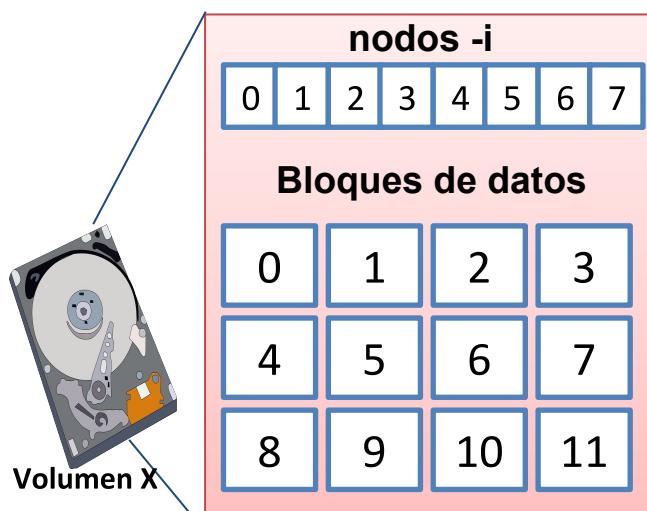
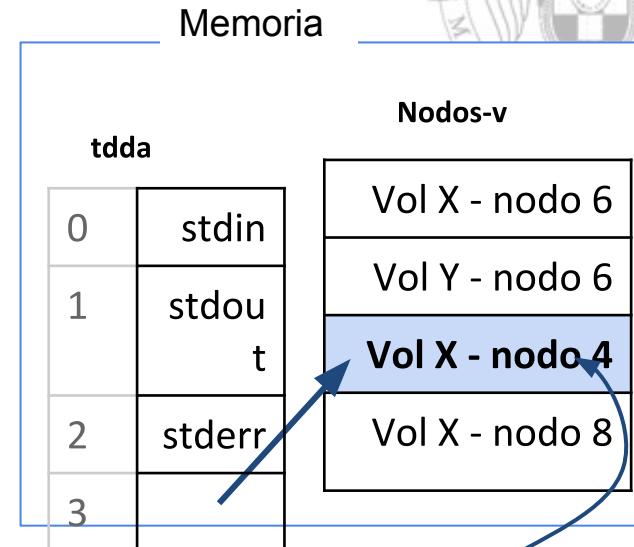
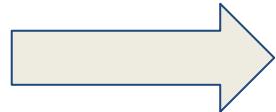
Memoria



Una por proceso

Una en el sistema

```
/* apertura de fichero  
correspondiente al nodo-i 4 */  
fd= open("/tmp/hola.txt,...));  
// fd devuelve el entero 3
```





Recuerda: interpretación de nombres

```
fd = open("/tmp/hola.txt", O_RDWR);
```

1. Leer nodo-i 2 (correspondiente a '/'). Es posible que ya esté en memoria (**tabla nodos-v**)
 2. Leer bloque de datos con contenido apuntado desde nodo-i 2: encontramos nodo-i de *tmp*
 3. Leer nodo-i de directorio *tmp* (puede que ya en memoria)
 4. Leer bloque de datos con contenido apuntado desde nodo-i de *tmp*
 5. Leer de disco el nodo-i de fichero *hola.txt* (el 5 en nuestro ejemplo) y buscarle entrada libre en *tabla nodos-v*
 6. Comprobar permisos: ¿el usuario actual tiene permisos de lectura y escritura sobre ese fichero?
 7. Buscar nueva entrada en **tdda** (entrada 3 en nuestro ejemplo) y apuntar (puntero a una dirección de memoria) a la entrada recién creada en la *tabla nodos-v*
 8. Devolver '3' como valor de salid en *open()* (o -1 si hubo un error)
-
- ¿Cómo sería este mismo proceso con un sistema **FAT**?



¿Falta algo?

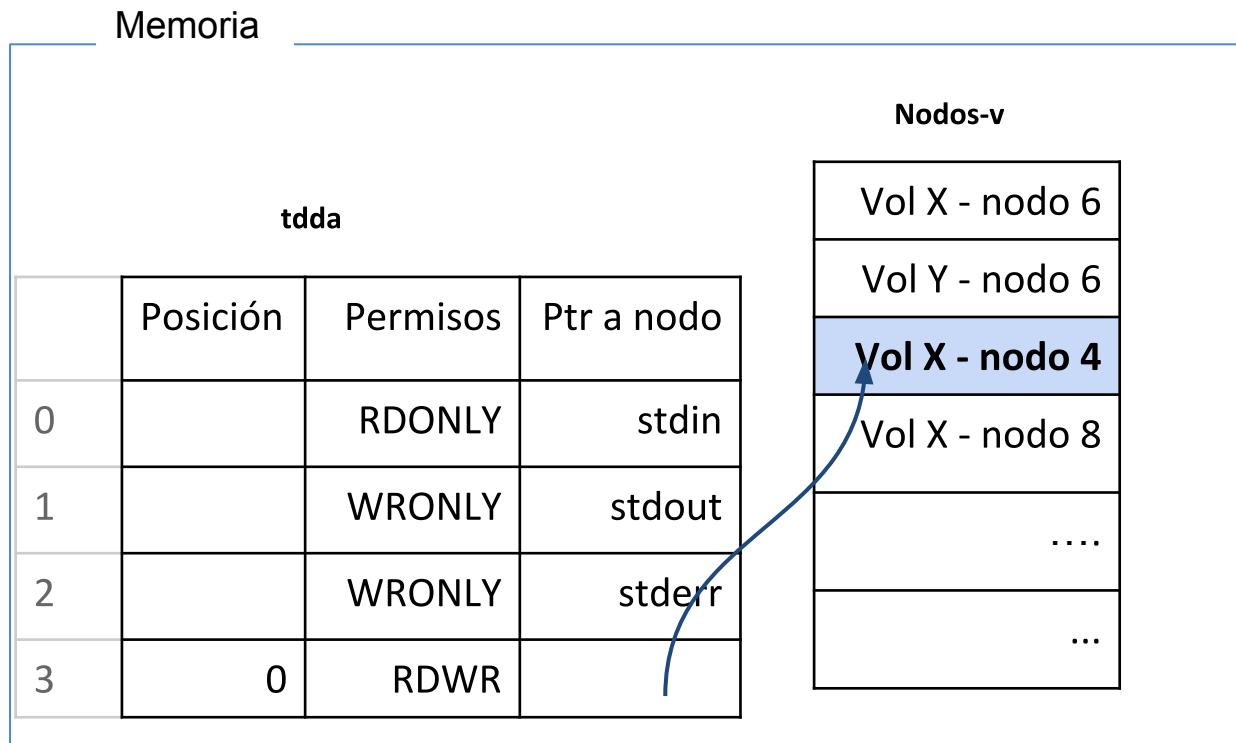
- ¿Dónde especificamos los permisos de apertura?
 - Un fichero puede tener permisos de lectura y escritura, pero el usuario hacer una apertura de solo lectura
- ¿Y qué hay de la *posición* de lectura/escritura que teníamos en la visión lógica del fichero?
- ¿Ideas?



Solución 1 (INCOMPLETA)

- Lo incluimos en la propia tabla *tdda*

```
/* apertura de fichero  
correspondiente al nodo-i 4 */  
fd= open(“/tmp/hola.txt,0_RDWR));  
  
// fd devuelve el entero 3
```

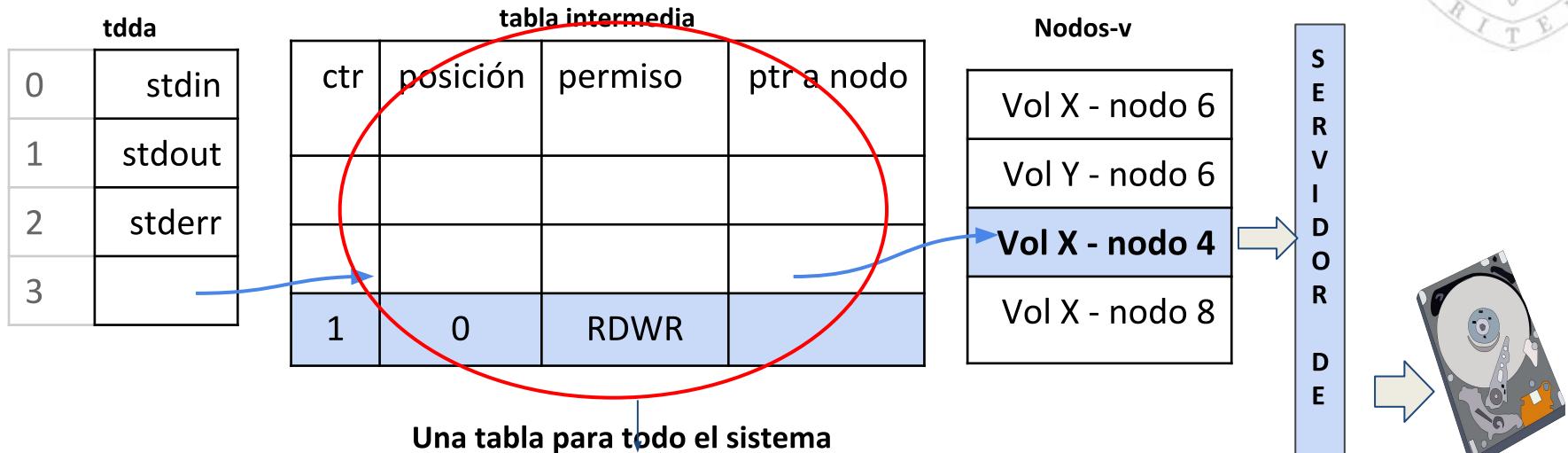


- ¿Y si quisiéramos compartir una instancia de apertura entre varios procesos?

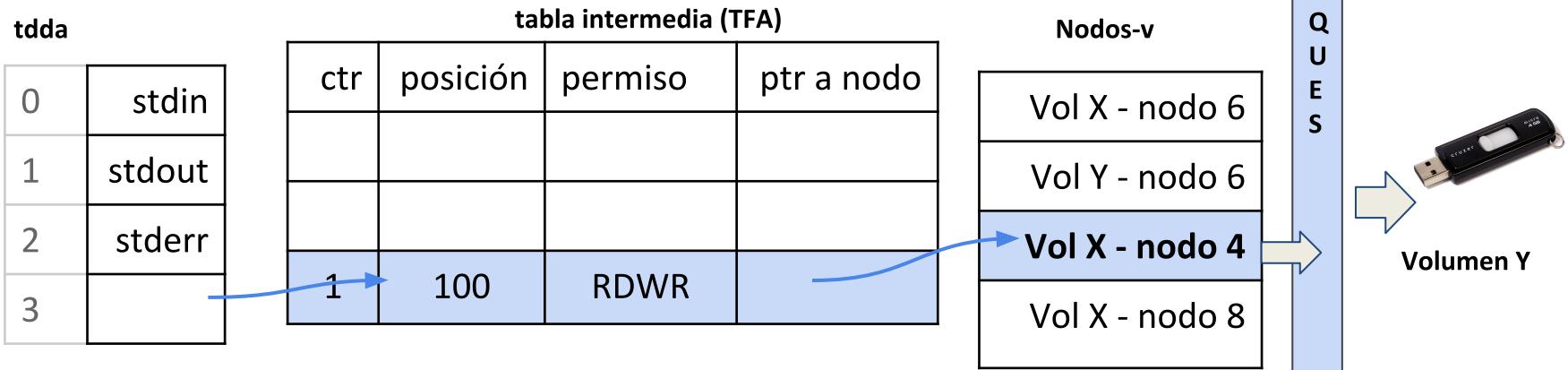
Solución 2 (DEFINITIVA)



`fd= open("/tmp/hola.txt,O_RDWR);`



`read(fd,buf,100);`





Una *tdda* por proceso

TDDA P1	
0	23
1	4563
2	56
3	4
4	678

TDDA P2	
0	230
1	563
2	98
3	3

TDDA P3	
0	2300
1	53
2	3
3	465
4	326

Tabla nodos-v

Nodo-v	Contador
...	
info de nodo-i 98	2
...	

TFA (intermedia)

Pos L/E	num nodo-i	Perm.	Cont. Refs.
...			
456	98	rw	2
3248	98	r	1
...			

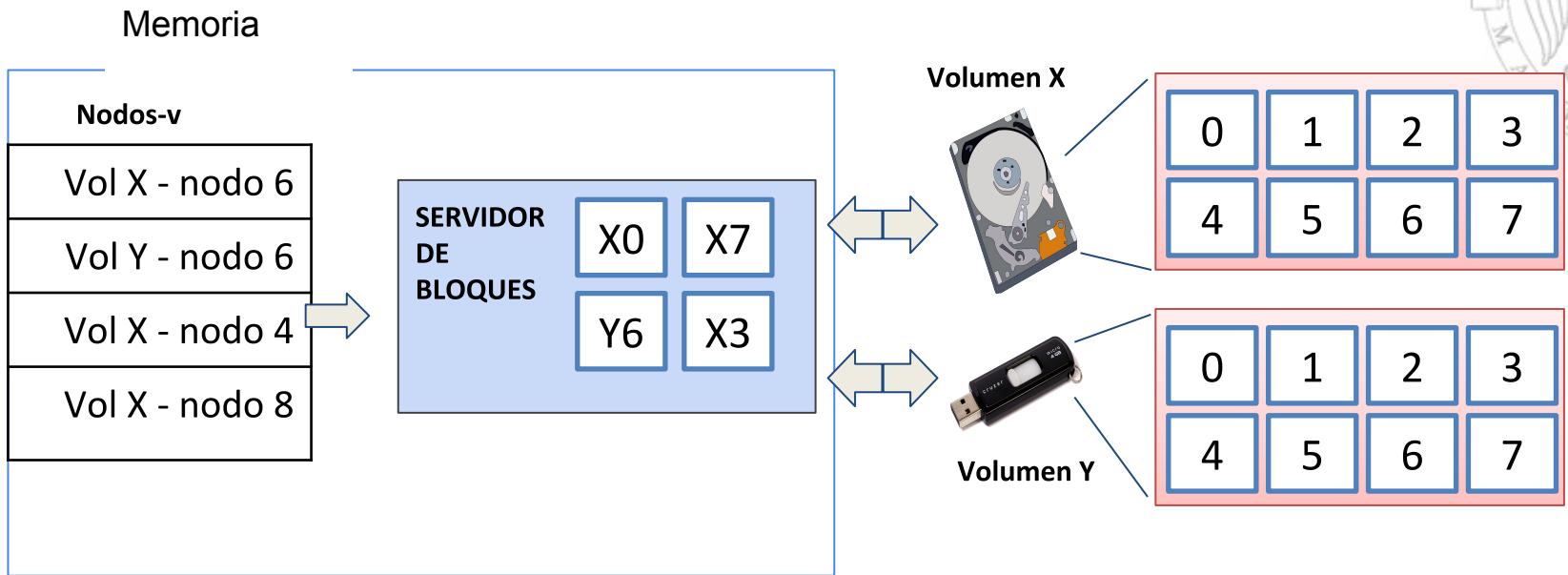


¿Tabla de nodos-v en memoria?

- Subconjunto (*cache*) de los nodos-i de *todos* los dispositivos (discos, memoria flash...) montados en el sistema
 - La información almacenada no tiene por qué ser idéntica a la del nodo-i correspondiente
 - Puede tener más campos, punteros....
- Es una capa de abstracción sobre la organización física del sistema de ficheros
 - Permite montar un volumen *FAT* y otro *ext-2* de forma transparente al usuario
- Es una abstracción habitual en sistemas tipo UNIX
 - ¡¡Pero no es la única!!
 - En un sistema operativo que sólo soporte *FAT*, el equivalente es tener en memoria partes de la *FAT*



Servidor de bloques



- Acepta solicitudes del tipo: '*dame el bloque físico 73 del dispositivo X*'
- Almacena un subconjunto (cache) de bloques en memoria
- Aisla la gestión de *nodos-v* del acceso a los dispositivos físicos concretos



Tabla de superbloques

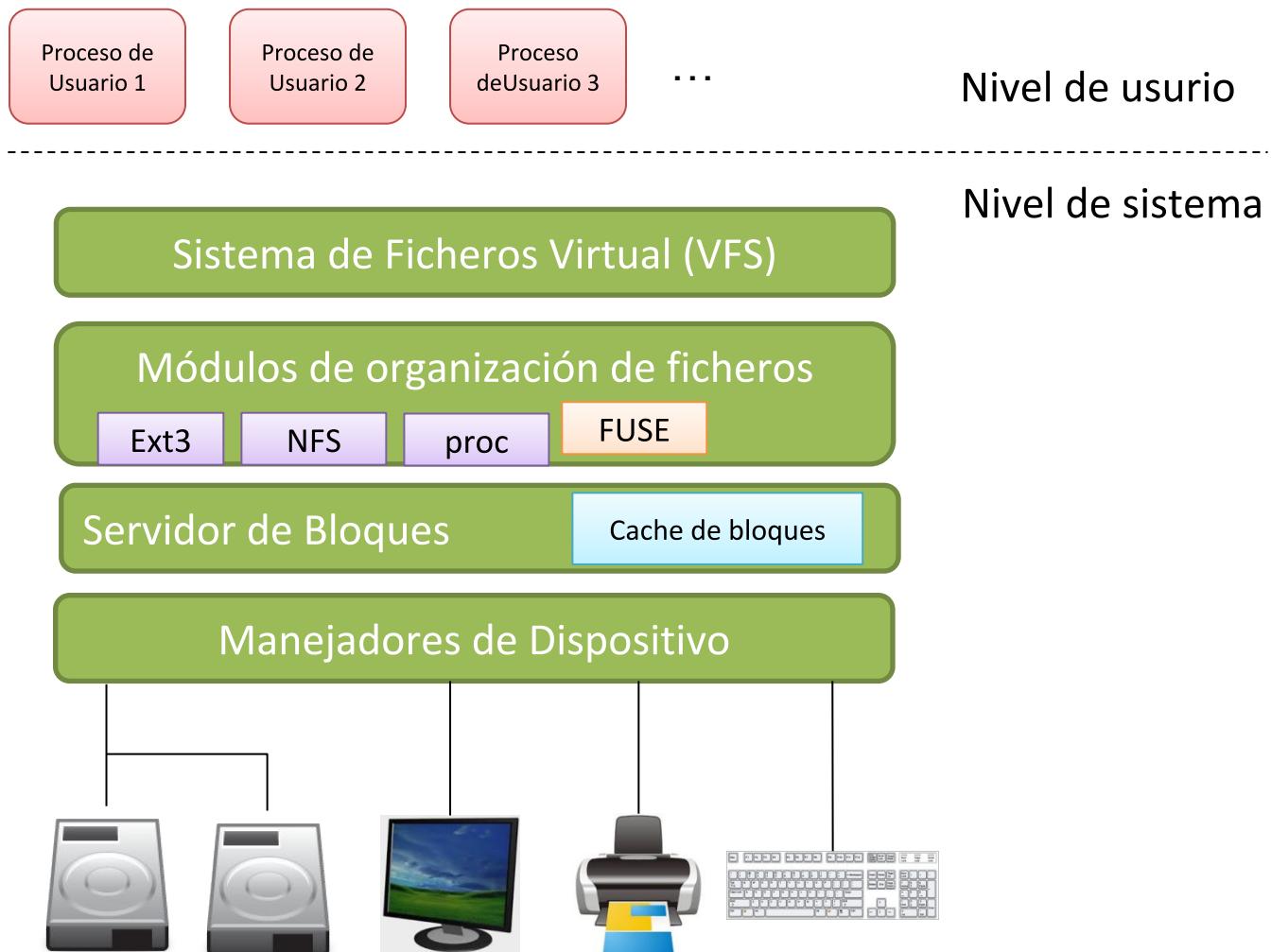
- Cada sistema de ficheros montado tiene su propio *superbloque*
 - Contiene meta-information sobre el sistema de ficheros
- En memoria, se mantiene un modelo del superbloque por cada sistema de ficheros montado
 - Da acceso a las operaciones concretas (*open, read, write...*) para ese sistema de ficheros
 - Indica el punto de montaje en el grafo de directorios y lo relacion con el volúmen/dispositivo físico



Resumen: tablas en memoria

1. Una ***tdda*** por proceso en ejecución
2. Una sola ***tabla de nodos-v*** (tipo *cache*) en el sistema
3. Una sola ***tabla intermedia*** (TFA) en el sistema
4. Una *cache* de bloques de datos, con bloques de todos los dispositivos
5. Tabla de superbloques (una en el sistema)
 - a. Una entrada con el superbloque de cada sistema de ficheros montado
6. *Extra*: una *cache* de nombres para acelerar traducciones
 - a. Especie de *diccionario* que, dada una cadena de caracteres que representa una ruta, nos indica el *nodo-v* que le corresponde

Estructura del servidor de ficheros



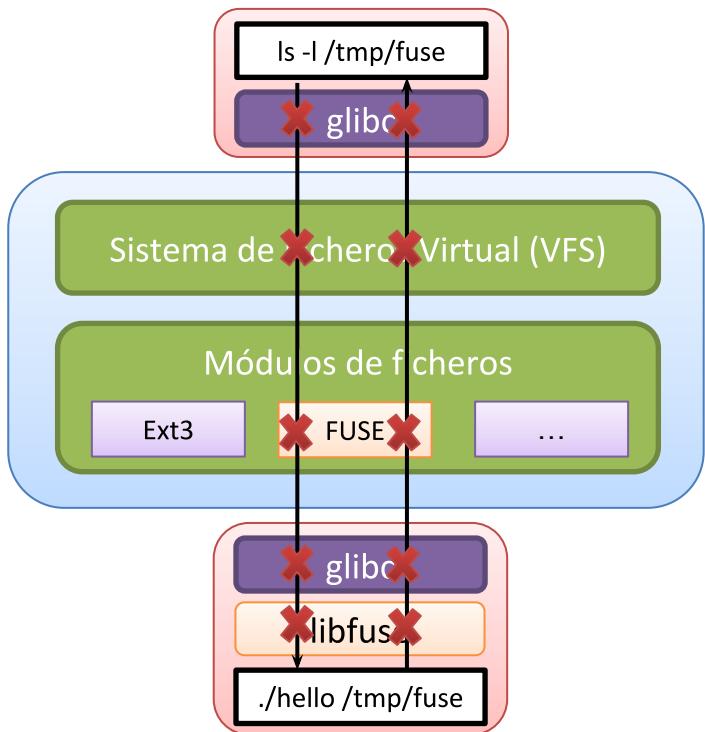


Ejemplo - FUSE

- RECUERDA: # mount -t ext3 /dev/hdb2 /mnt/unidad



- FUSE significa **Filesystem in USErspace**.



FUSE funciona exactamente de la misma manera.

- 1) Módulo para el kernel que se registra como un manejador de un sistema de archivos cualquiera.
- 2) Cuando queremos acceder a una unidad FUSE montada, el VFS le redirige la llamada al módulo de FUSE, y FUSE nos redirige la llamada a nuestro programa controlador
→ Es una capa entre el Kernel y nuestro programa en el espacio de usuario.



Ejemplo - FUSE

- ¿Cómo se usa? → Creamos controlador (archivo .c)
 1. Hay que incluir fuse.h y enlazar con libfuse
 2. Declarar estructura llamada **fuse_operations**
 - Contiene punteros a funciones que serán llamados por cada operación
 3. Se termina el programa con la llamada a fuse_main
- Normas generales
 - Las funciones deben devolver 0 en caso de éxito y un número negativo indicando el error en caso de fallo.
 - Excepción: Las llamadas a write/read deben devolver un número positivo indicando los bytes leídos, 0 en caso de EOF o número negativo en caso de error.



Ejemplo - FUSE

- Miembros básicos de la estructura **fuse_operations**:

```
$ more /usr/include/fuse/fuse.h
```

```
int (*getattr) (const char *, struct stat *);
```

- Función llamada cuando se quieren obtener los atributos de un archivo, p.e. cuando se le hace un “stat” a un archivo se llama a esta función. El primer parámetro indica la ruta del archivo. El segundo parámetro es la estructura stat a llenar.

```
int (*open) (const char *, struct fuse_file_info *);
```

- Se llama al abrir un archivo. El primer parámetro es la ruta del archivo, el segundo parámetro contiene información acerca de los flags de apertura. Permite devolver un handler, pero no es un parámetro necesario para un sistema de archivos sencillo y se puede ignorar. Lo único que hay que hacer es comprobar si se puede abrir el archivo, en caso afirmativo se devuelve cero.

```
int (*read) (const char *, char *, size_t, off_t, struct fuse_file_info *);
```

- Se llama al leer un archivo. El primer parámetro es la ruta del archivo, el segundo es el buffer donde almacenar los datos, el tercer parámetro es la cantidad de bytes a leer, el cuarto el desplazamiento y el quinto es el mismo que el de open. Se devuelve la cantidad de bytes leídos.

```
int (*readdir) (const char *, void *, fuse_fill_dir_t, off_t, struct fuse_file_info *);
```

- Se utiliza para leer un directorio. El primer parámetro es la ruta del directorio, el segundo una estructura que hay que llenar, el tercero es una función usada para llenar la estructura del segundo parámetro y los otros dos se pueden ignorar para ejemplos sencillos.



Ejemplo - FUSE

- Más miembros de la estructura **fuse_operations**:

```
int (*mknod) (const char *, mode_t, dev_t);
int (*unlink) (const char *);
int (*rename) (const char *, const char *);
int (*truncate) (const char *, off_t);
int (*write) (const char *, const char *, size_t, off_t, struct
  fuse_file_info *);
```



Ejemplo - FUSE

■ Un ejemplo sencillo (1/4):

```
#include <fuse.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>

static const char *hello_str = "Hello World!\n";
static const char *hello_path = "/hello";

static int hello_getattr(const char *path, struct stat *stbuf, t
```



El path que nos pasa FUSE siempre es un path "absoluto" pero referenciado al sistema de archivos.



Ejemplo - FUSE

- Un ejemplo sencillo (2/4):

```
static int hello_open(const char *path, struct fuse_file_info *fi) {
    if (strcmp(path, hello_path) != 0)
        return -ENOENT;
    if ((fi->flags & 3) != O_RDONLY)
        return -EACCES;
    return 0;
}

static int hello_read(const char *path, char *buf, size_t size, off_t offset,
                     struct fuse_file_info *fi) {
    size_t len;
    (void) fi;
    if(strcmp(path, hello_path) != 0)
        return -ENOENT;
    len = strlen(hello_str);
    if (offset < len) {
        if (offset + size > len)
            size = len - offset;
        memcpy(buf, hello_str + offset, size);
    } else
        size = 0;
    return size;
}
```



Ejemplo - FUSE

- Un ejemplo sencillo (3/4):

```
static int hello_readdir(const char *path, void *buf, fuse_fill_dir_t filler,
                        off_t offset, struct fuse_file_info *fi) {
    (void) offset;
    (void) fi;
    if (strcmp(path, "/") != 0)
        return -ENOENT;
    filler(buf, ".", NULL, 0);
    filler(buf, "..", NULL, 0);
    filler(buf, hello_path + 1, NULL, 0);
    return 0;
}
static struct fuse_operations hello_oper = {
    .getattr = hello_getattr,
    .readdir = hello_readdir,
    .open     = hello_open,
    .read     = hello_read,
};

int main(int argc, char *argv[]) {
    return fuse_main(argc, argv, &hello_oper);
}
```



Ejemplo - FUSE

- Un ejemplo sencillo (4/4):

```
$ gcc hello.c -o hello -lfuse -D_FILE_OFFSET_BITS=64 -DFUSE_USE_VERSION=22
$ mkdir /tmp/fuse
$ ./hello /tmp/fuse
$ ls -l /tmp/fuse
    total 0
        -r--r--r--  1 root root 13 Jan  1 1970 hello
$ cat /tmp/fuse/hello
    Hello World!
$ fusermount -u /tmp/fuse
```