

# Lección 4

## Sistemas de ficheros

Diseño de Sistemas Operativos  
Grado en Ingeniería Informática

# Objetivos

---

1. Conocimiento de los métodos de gestión interna de recursos en un sistema operativo.
2. Motivación de logro de soluciones mejores.

# Lecturas recomendadas

---

## Base



1. Carretero 2007:
  1. Cap.9

## Recomendada



1. Tanenbaum 2006(en):
  1. Cap.5
2. Stallings 2005:
  1. Parte tres
3. Silberschatz 2006:
  1. Cap. 10, 11 y 12

# A recordar...

---

1. Estudiar la teoría asociada.
  - ▶ Estudiar el material asociado a la bibliografía: las transparencias solo no son suficiente.
2. Repasar lo visto en clase.
  - ▶ Realizar el cuaderno de prácticas progresivamente.
3. Ejercitar las competencias.
  - ▶ Realizar las prácticas progresivamente.
  - ▶ Realizar todos los ejercicios posibles.

# Contenidos

---

1. Introducción
2. Marco de trabajo
3. Diseño y desarrollo de un sistema de ficheros
4. Aspectos complementarios

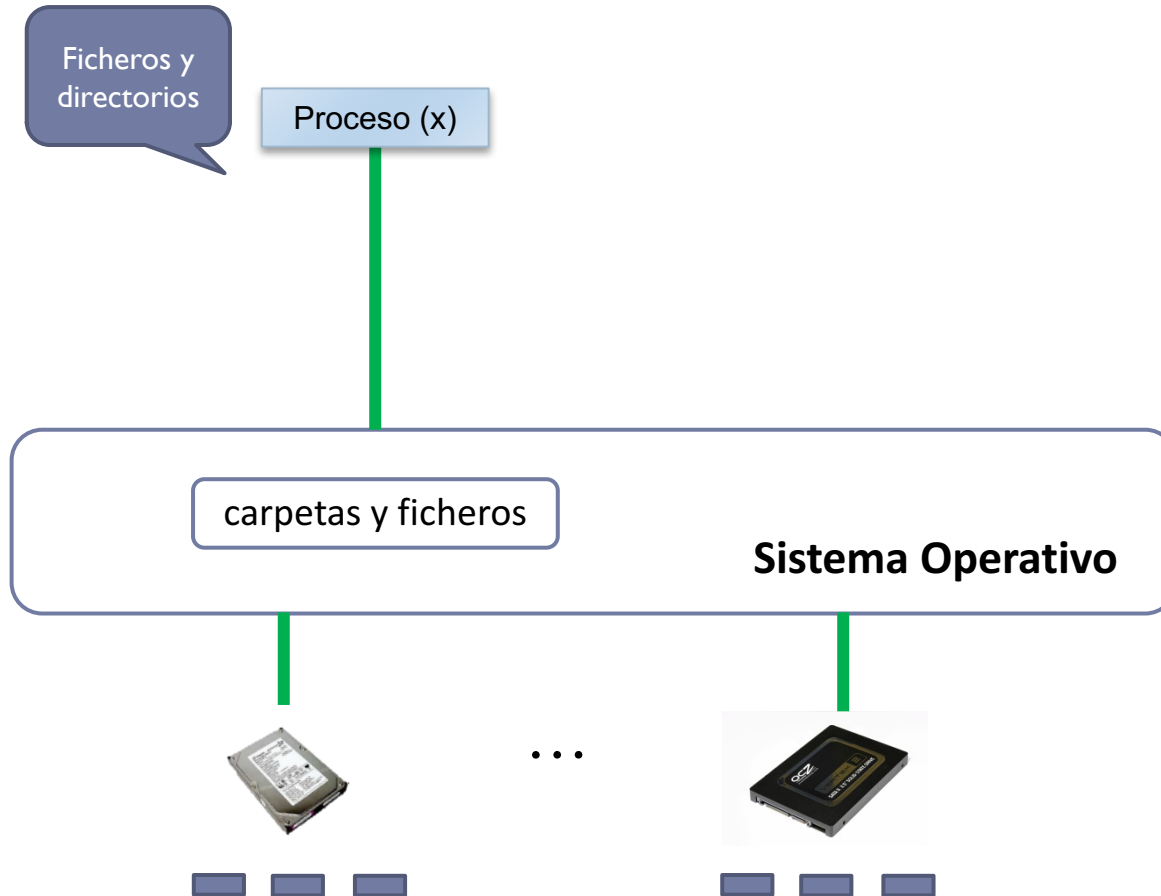
# Contenidos

---

1. **Introducción**
2. Marco de trabajo
3. Diseño y desarrollo de un sistema de ficheros
4. Aspectos complementarios

# (1/2) El S.O. integra una abstracción básica y genérica: sistema de ficheros

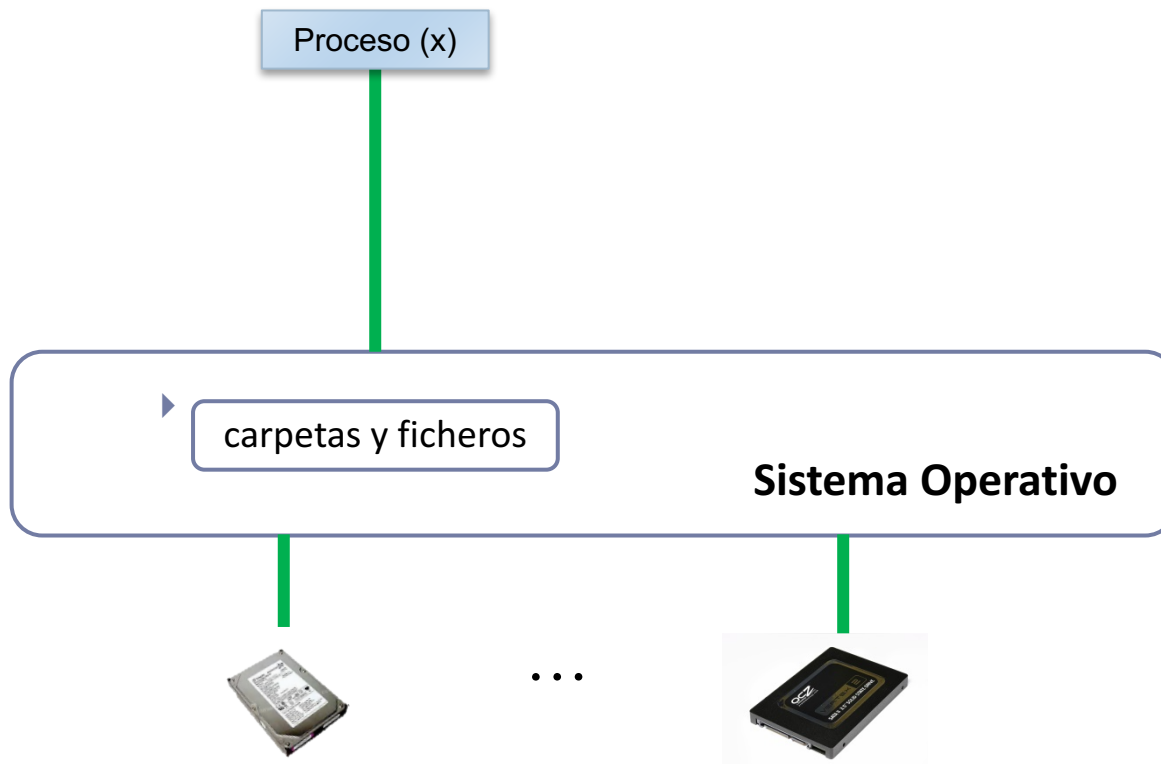
---



# Principales Características de un sistema de ficheros

---

- ▶ Incorporar una funcionalidad añadida para facilitar el manejo del almacenamiento secundario:

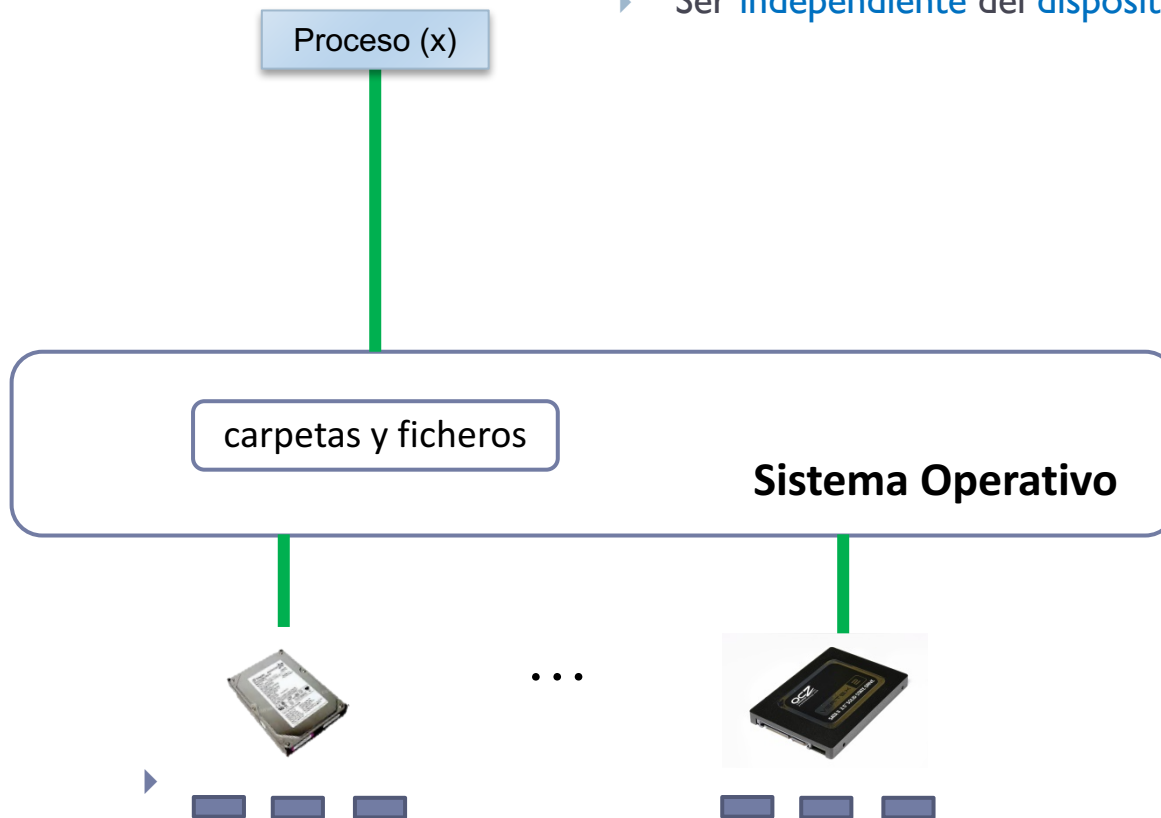




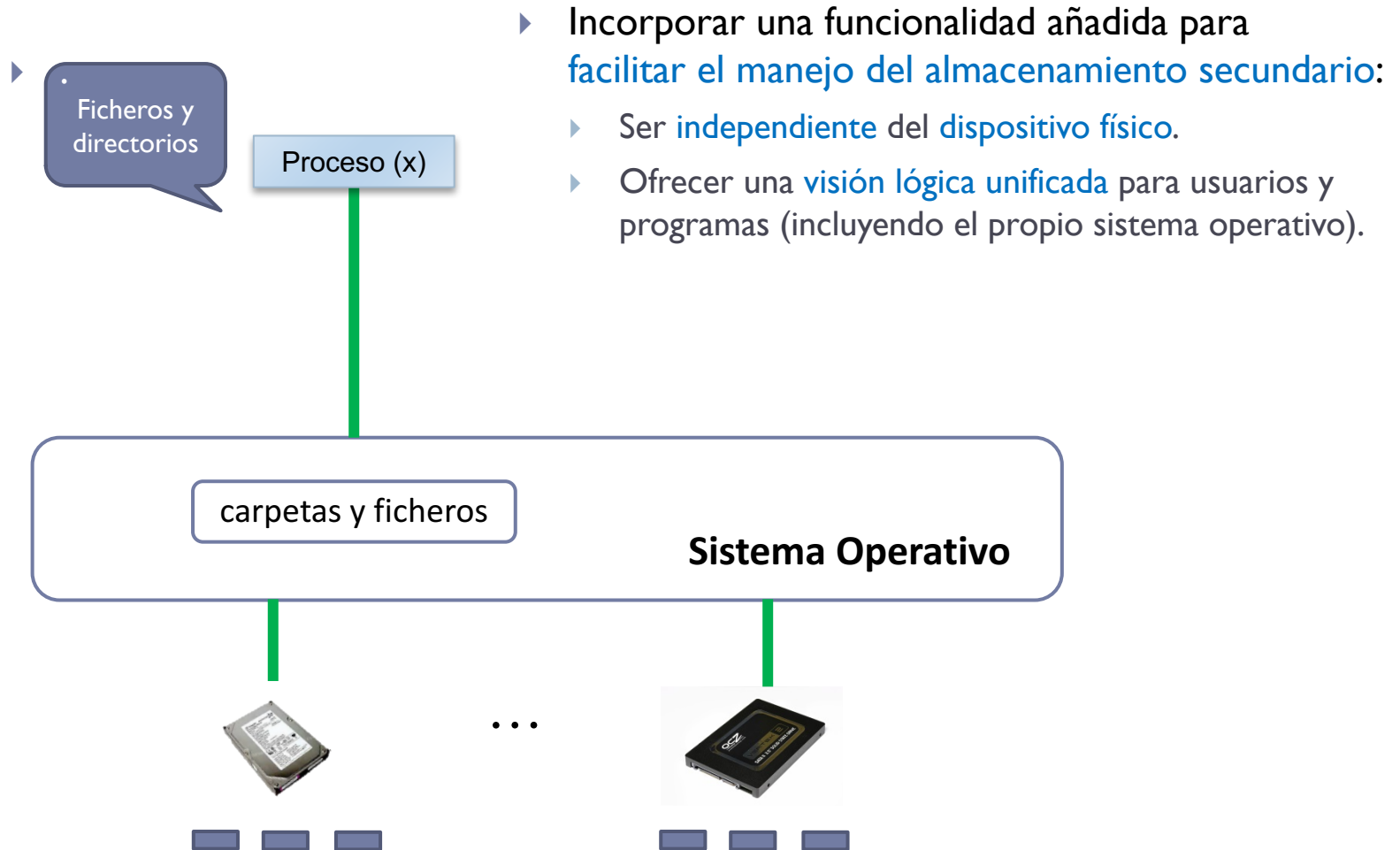
# Principales Características de un sistema de ficheros

---

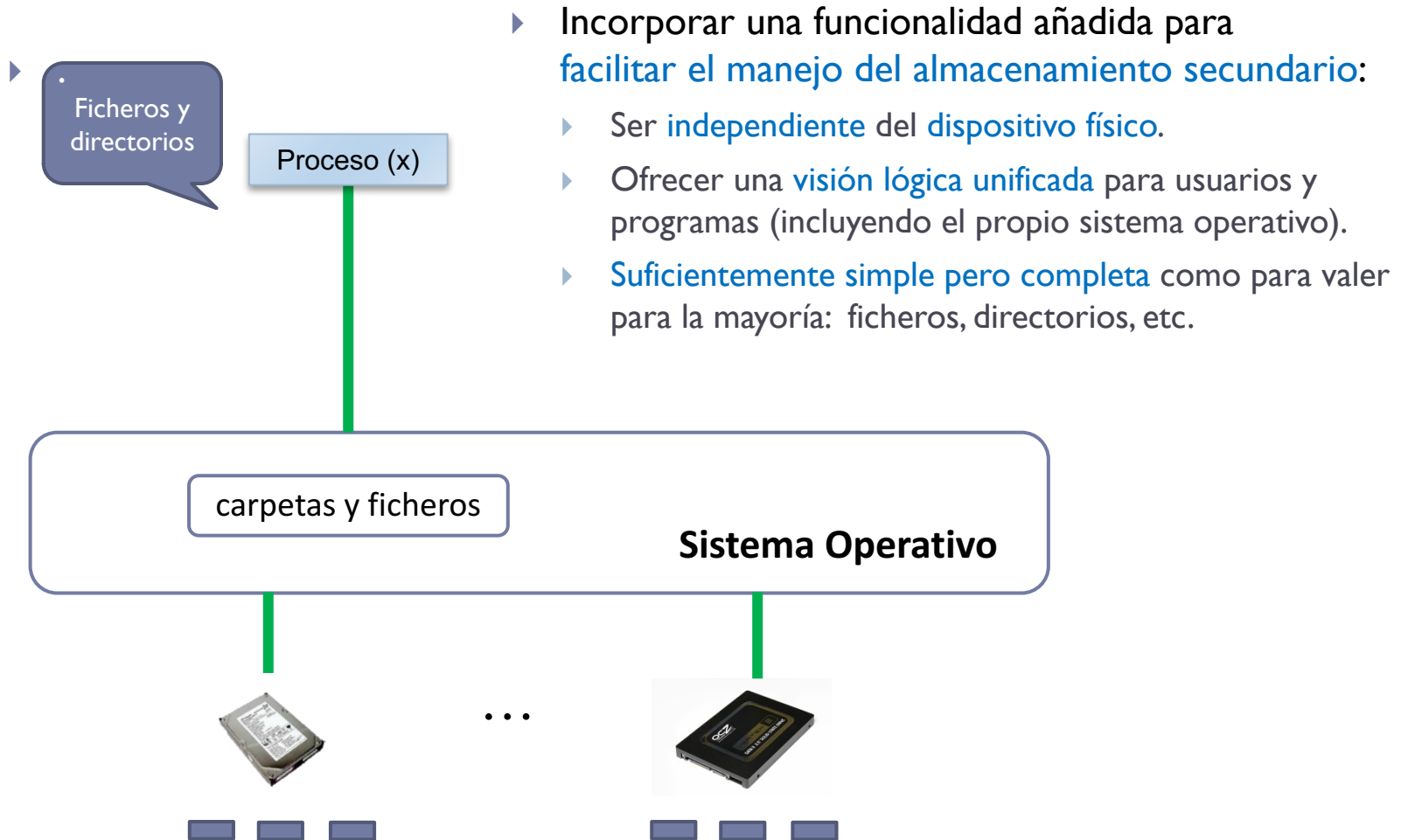
- ▶ Incorporar una funcionalidad añadida para **facilitar el manejo del almacenamiento secundario**:
  - ▶ Ser **independiente** del dispositivo físico.



# Principales Características de un sistema de ficheros

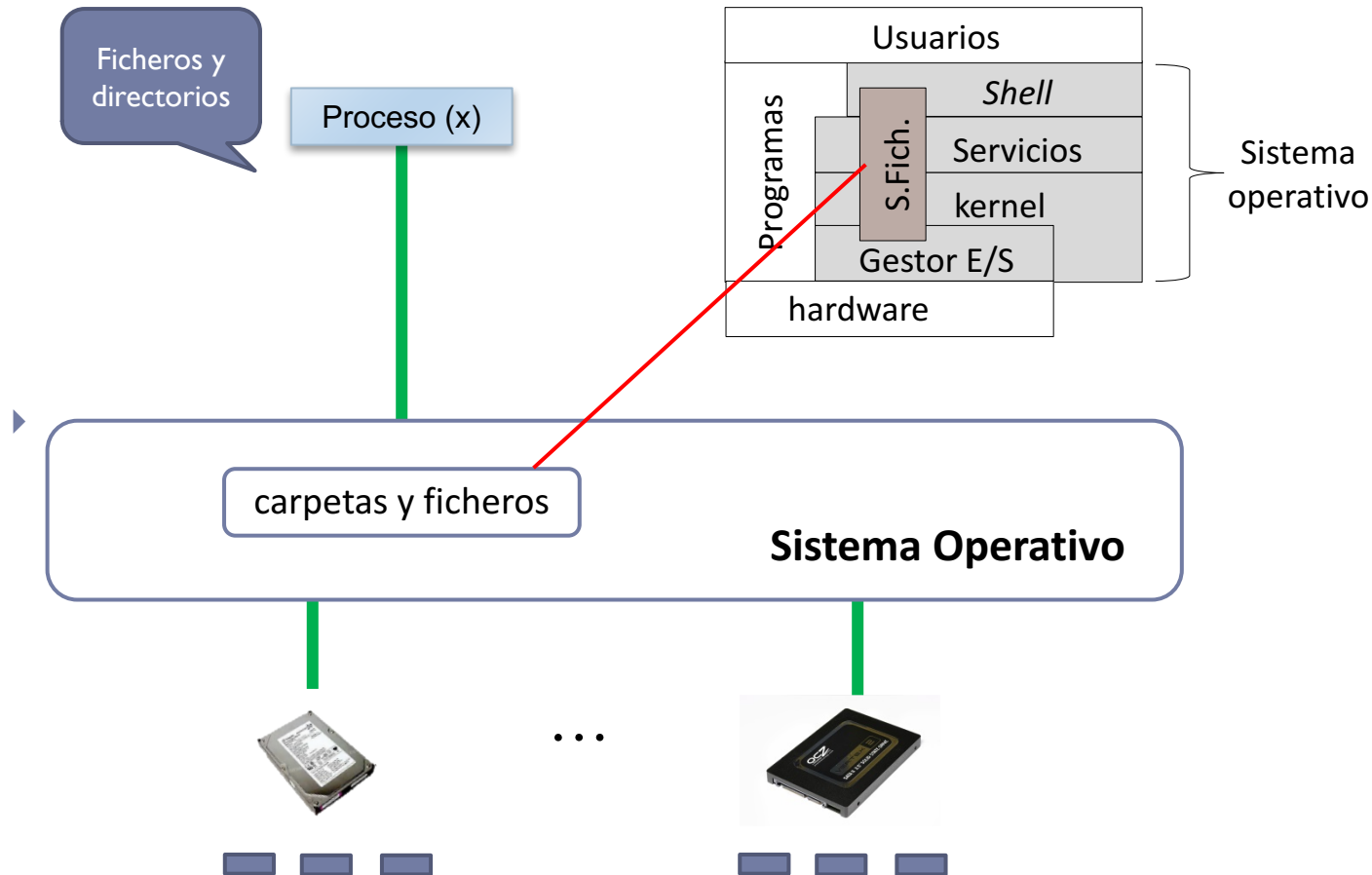


# Principales Características de un sistema de ficheros

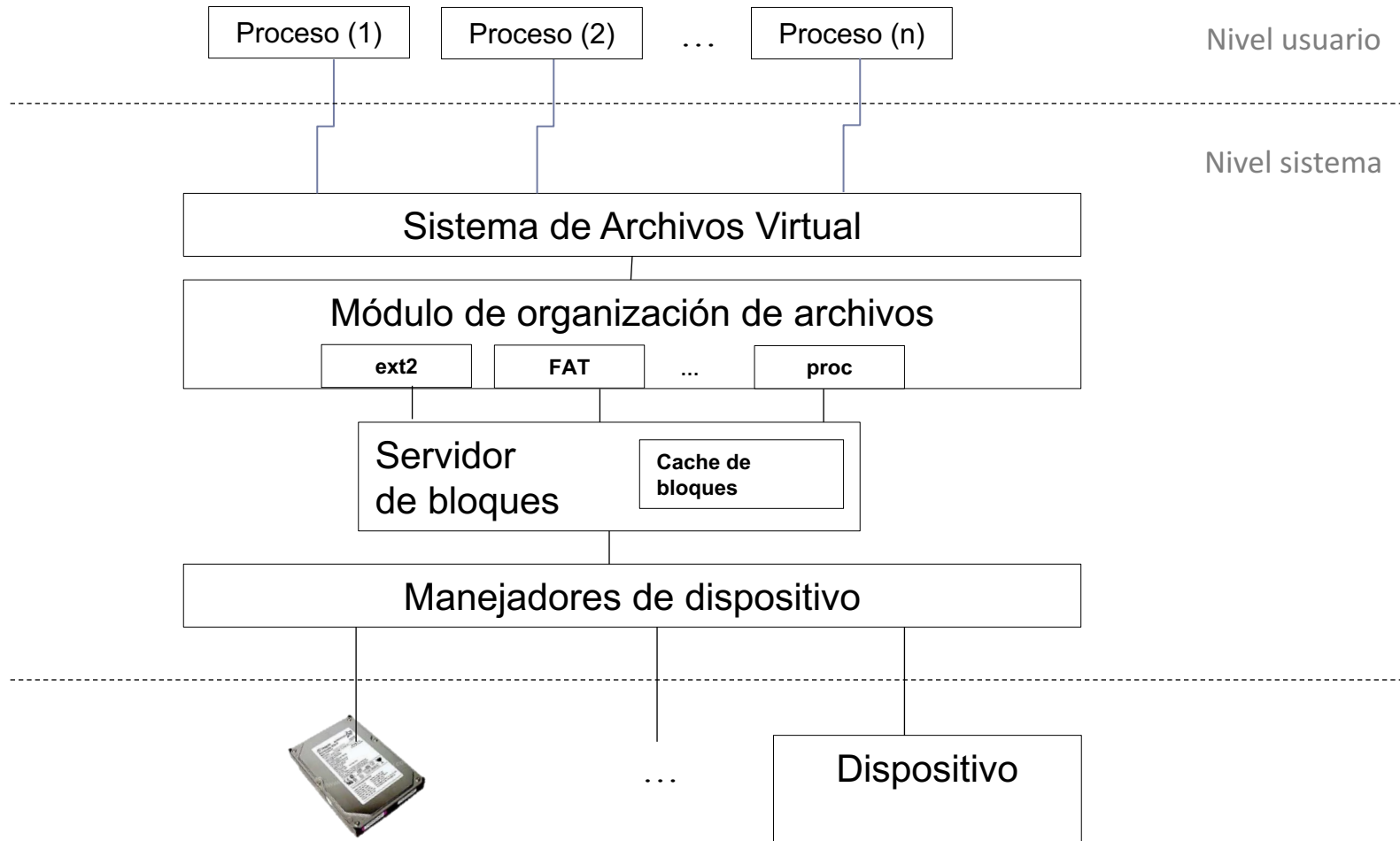


# Principales Características de un sistema de ficheros

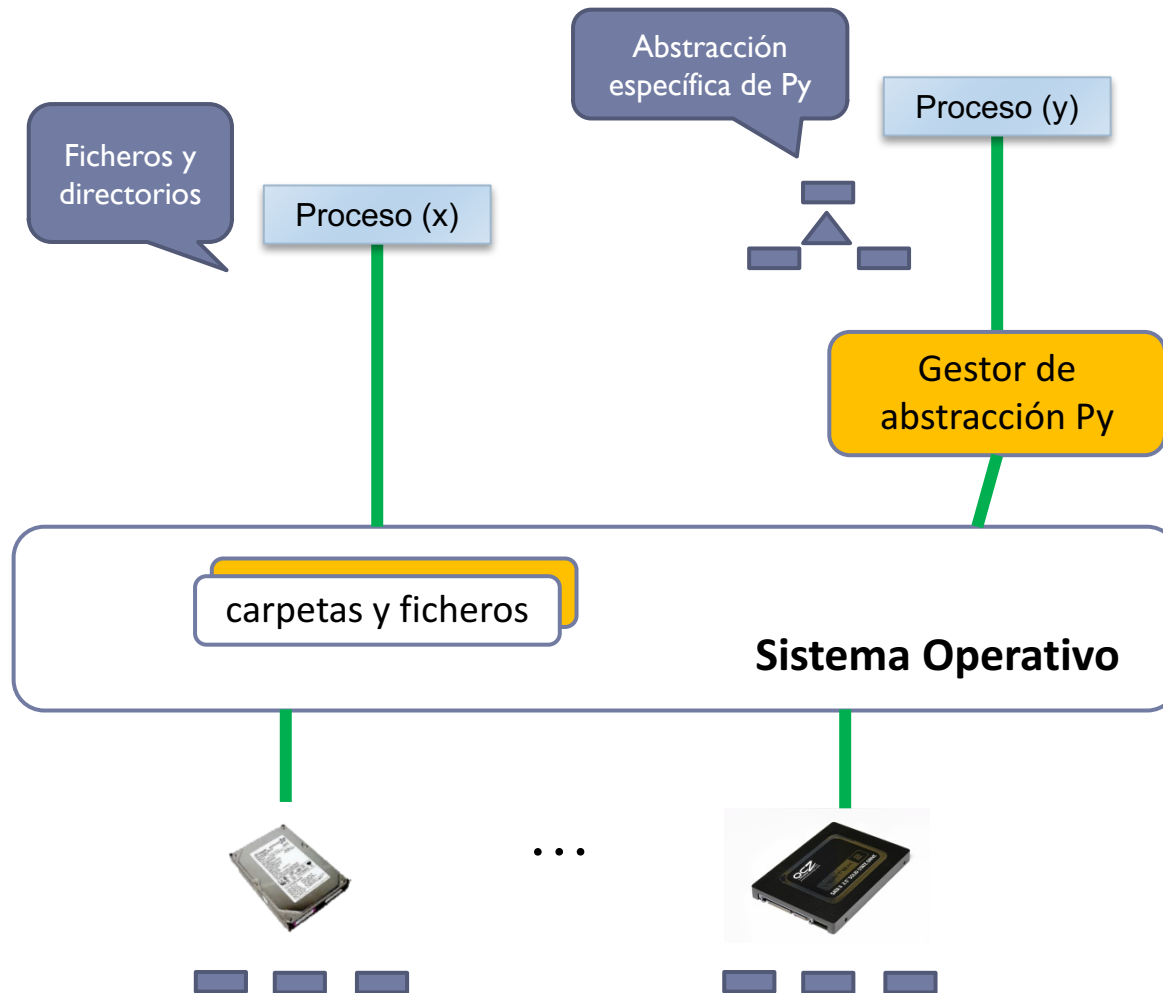
- ▶ Arquitectura transversal al sistema operativo:



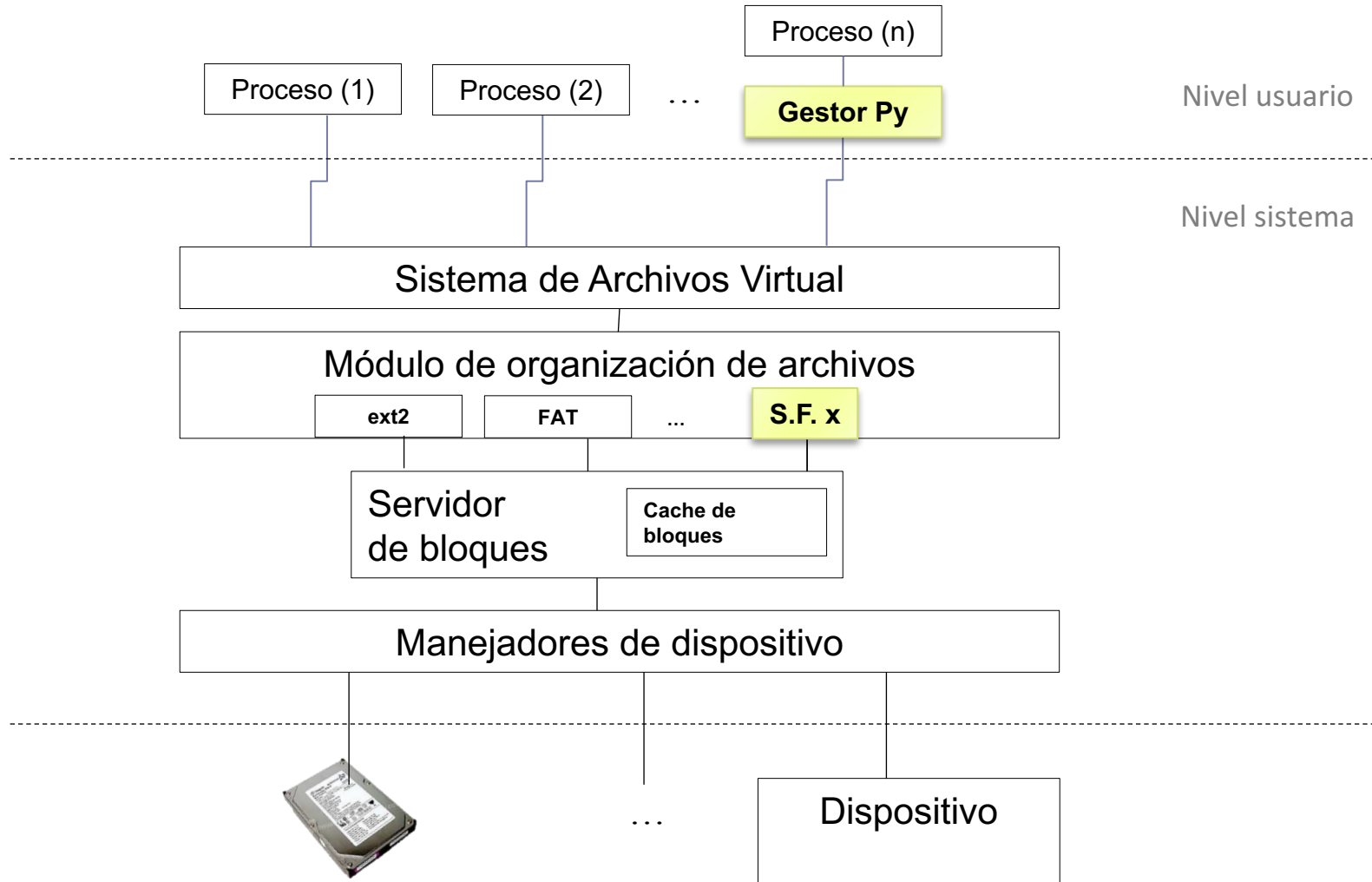
# Arquitectura de los sistemas de ficheros



## (2/2) El S.O. da soporte para construir hasta otros sistemas de almacenamiento

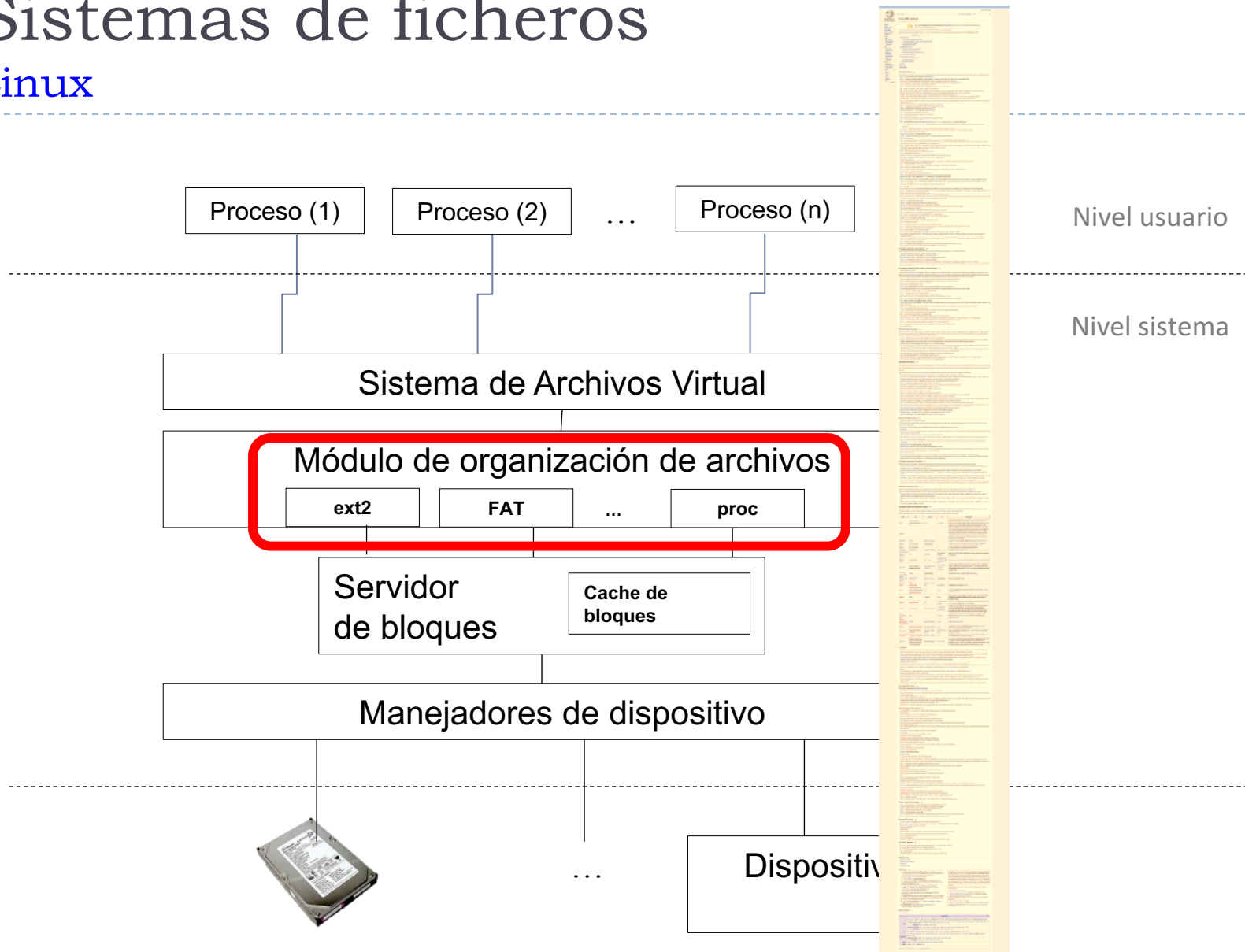


# Arquitectura ampliable con sistemas de ficheros y gestores externos



# Sistemas de ficheros

## Linux





# Contenidos

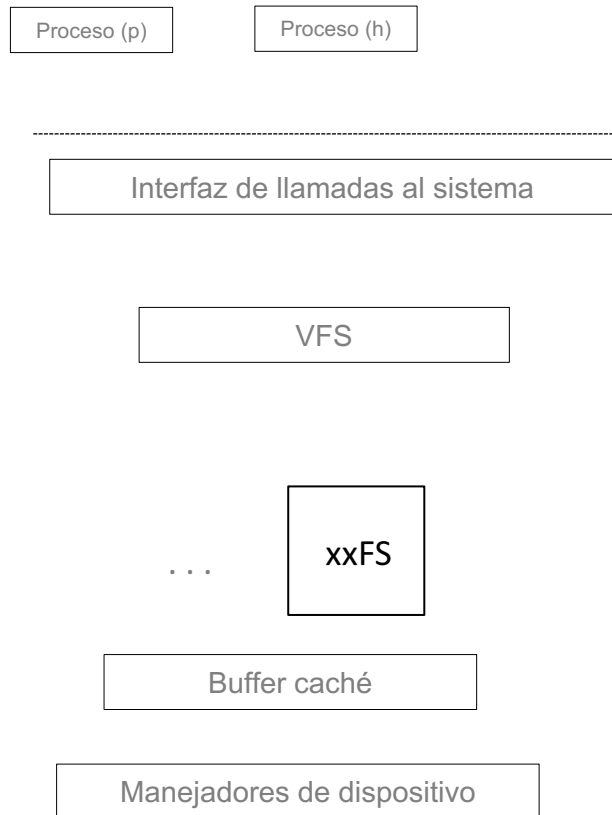
---

1. Introducción
2. **Marco de trabajo**
3. Diseño y desarrollo de un sistema de ficheros
4. Aspectos complementarios

# Organización del sistema de ficheros

## principales aspectos: Linux

---

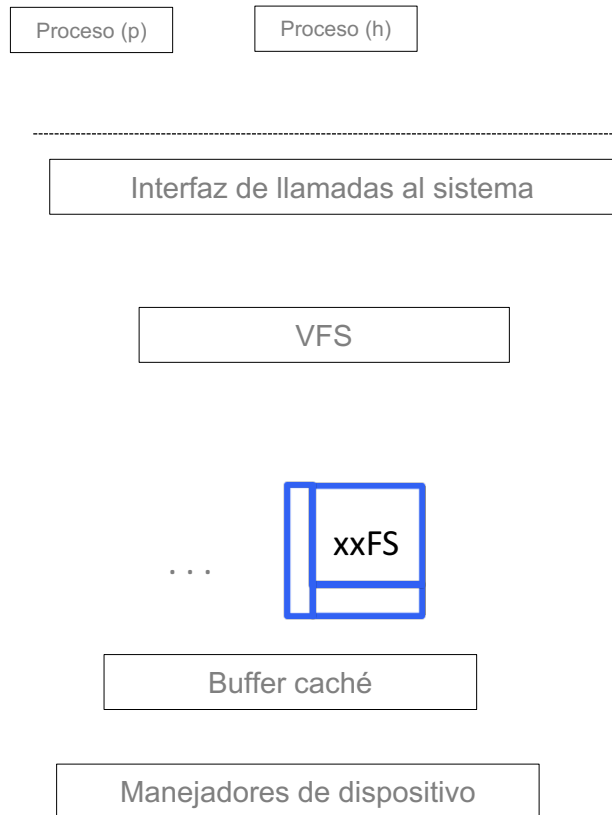


- ▶ Estructura en capas tipo UNIX.
- ▶ Principales componentes:
  - ▶ Interfaz de llamadas al sistema
  - ▶ VFS: *Virtual File System*
  - ▶ xxFS: sistema de ficheros específico
  - ▶ Buffer caché: caché de bloques
  - ▶ Manejadores de dispositivos: *drivers*

# Organización del sistema de ficheros

sin *framework* de apoyo, en kernel. Ej.: *simplefs*

---



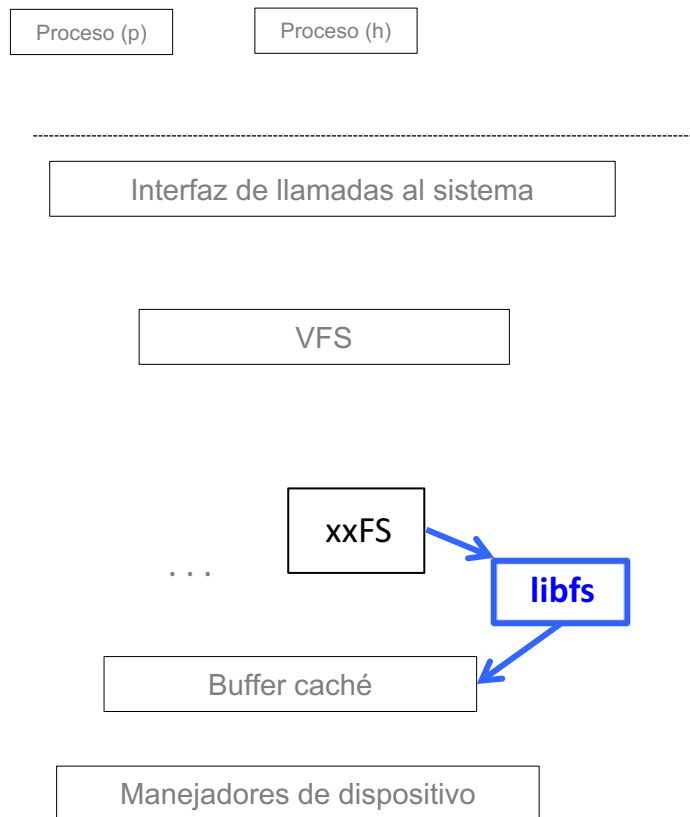
## ▶ Interfaz a escribir:

- ▶ **register**: dar de alta el sist. de ficheros
- ▶ ...
- ▶ **open**: abrir sesión de un fichero
- ▶ **read**: leer datos
- ▶ ...
- ▶ **namei**: convierte una ruta a inodo
- ▶ **iget**: lee el inodo
- ▶ **bmap**: calcula el bloque dado un offset
- ▶ ...

# Organización del sistema de ficheros

con *framework* de apoyo, en kernel: **libfs**

---

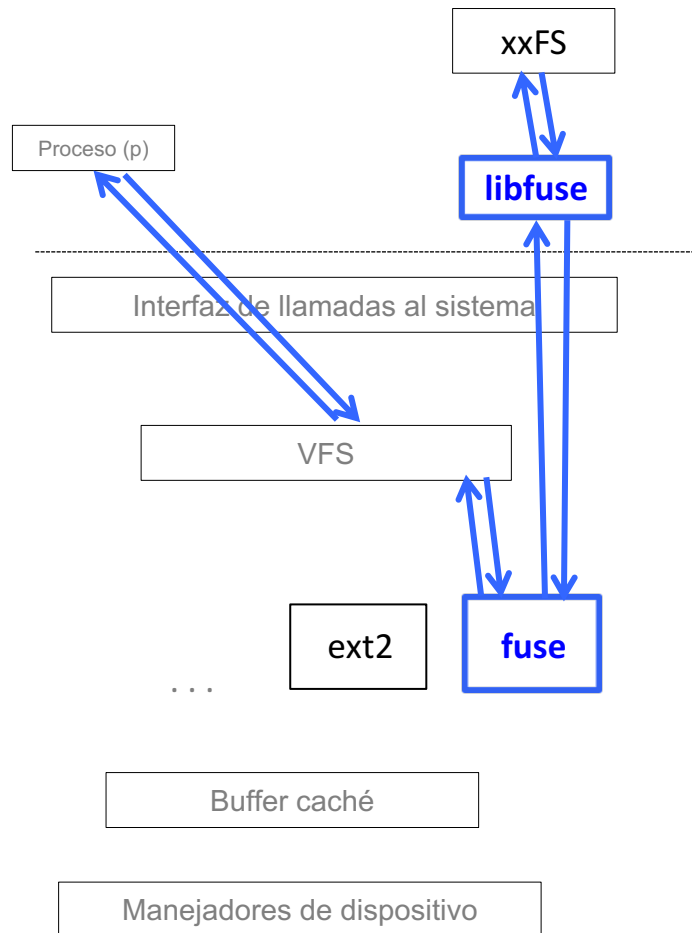


## ▶ Interfaz a completar/escribir: **libfs**

- ▶ **lfs\_fill\_super**: superbloque inicial
- ▶ **lfs\_create\_file**: crear fichero
- ▶ **lfs\_make\_inode**: inodo por defecto
- ▶ **lfs\_open**: abrir sesión de trabajo
- ▶ **lfs\_read\_file**: leer del fichero
- ▶ **lfs\_write\_file**: escribir al fichero
- ▶ ...

# Organización del sistema de ficheros

con *framework* de apoyo, en espacio de usuario: fuse

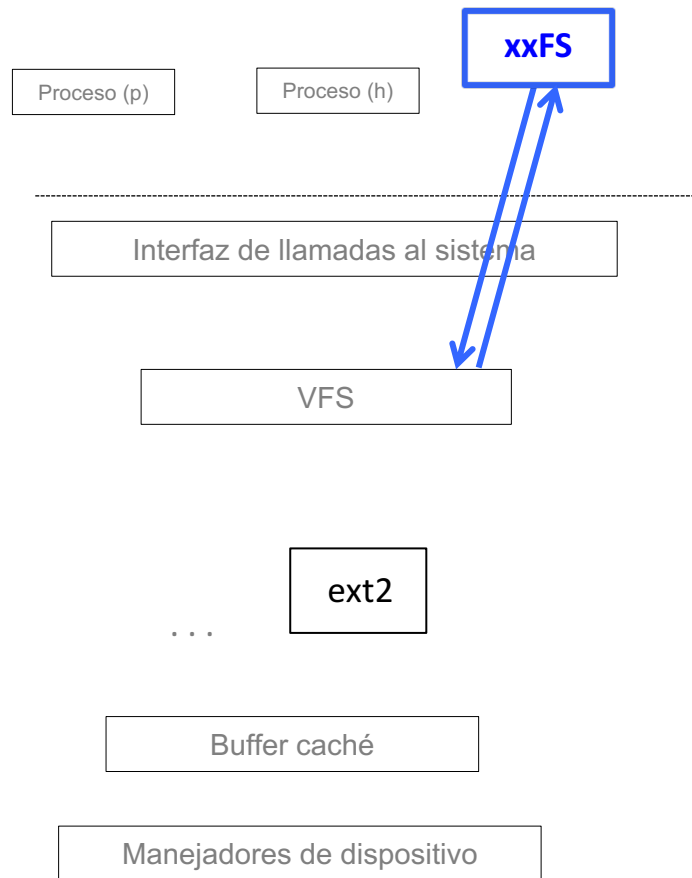


- Interfaz a completar/escribir:  
*File system in USer space*

```
struct fuse_operations {  
    int (*getattr) (const char *, struct stat *);  
    int (*readlink) (const char *, char *, size_t);  
    int (*getdir) (const char *, fuse_dirh_t, fuse_dirfil_t);  
    int (*mknod) (const char *, mode_t, dev_t);  
    ...  
    int (*listxattr) (const char *, char *, size_t);  
    int (*removexattr) (const char *, const char *);  
};
```

# Organización del sistema de ficheros

sin *framework* de apoyo, en espacio de usuario. Ej.: *mtools*



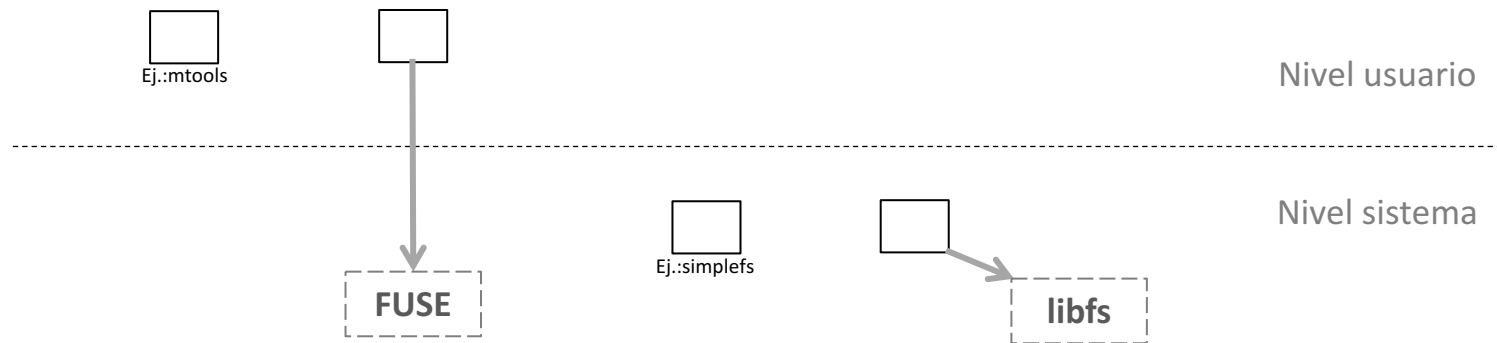
► Implementar la interfaz de un sistema de ficheros en espacio de usuario, y como biblioteca para otras aplicaciones:

- **open**: abrir sesión de un fichero
- **read**: leer datos
- ...
- **namei**: convierte una ruta a inodo
- **iget**: lee el inodo
- **bmap**: calcula el bloque dado un offset
- ...

# Principales alternativas para la organización de un sistema de ficheros

---

	Espacio de usuario	Espacio de kernel
<b>Con</b> <i>Framework</i> de apoyo	FUSE	libfs
<b>Sin</b> <i>Framework</i> de apoyo	Ej.: mtools	Ej.: simplefs



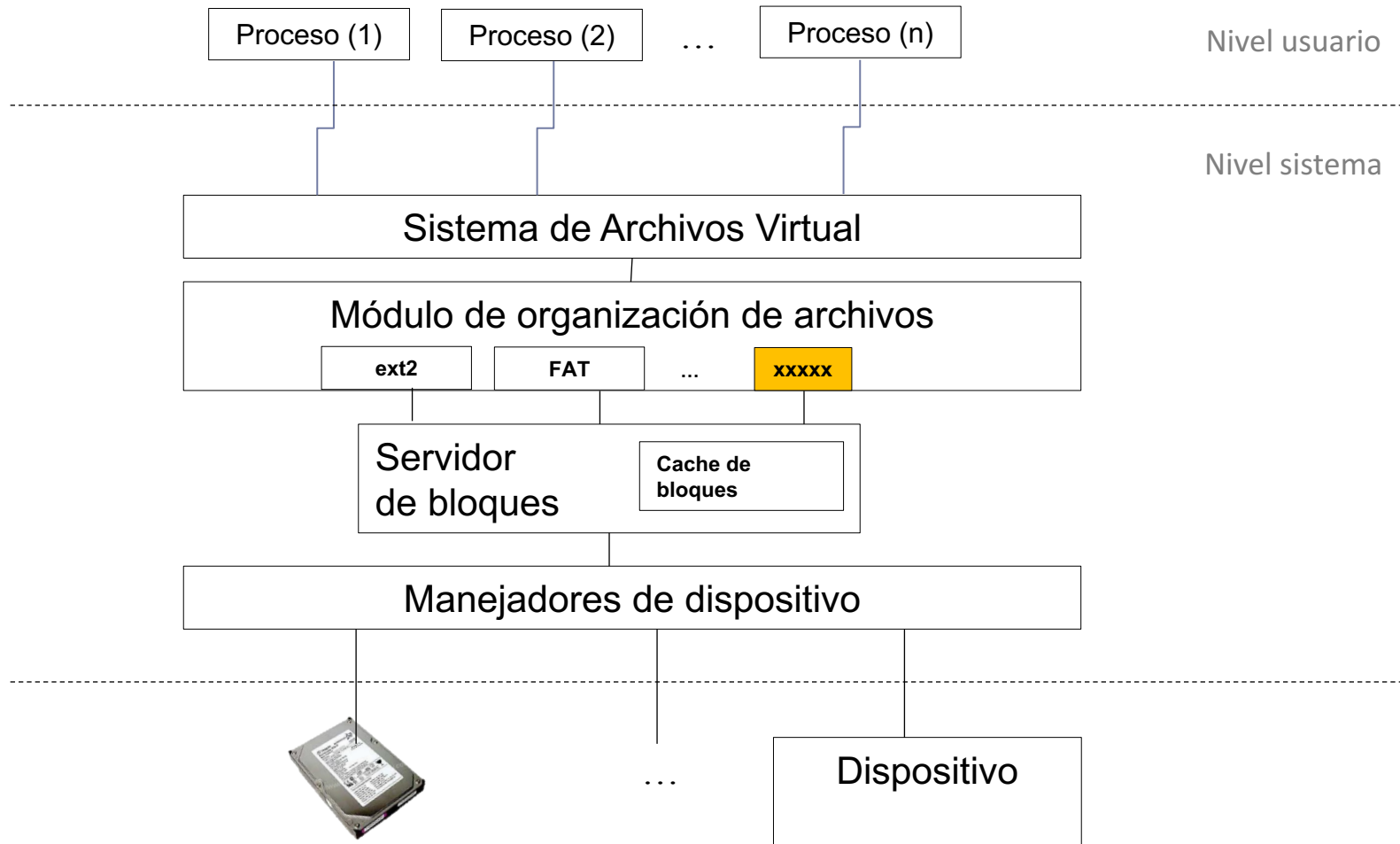
# Contenidos

---

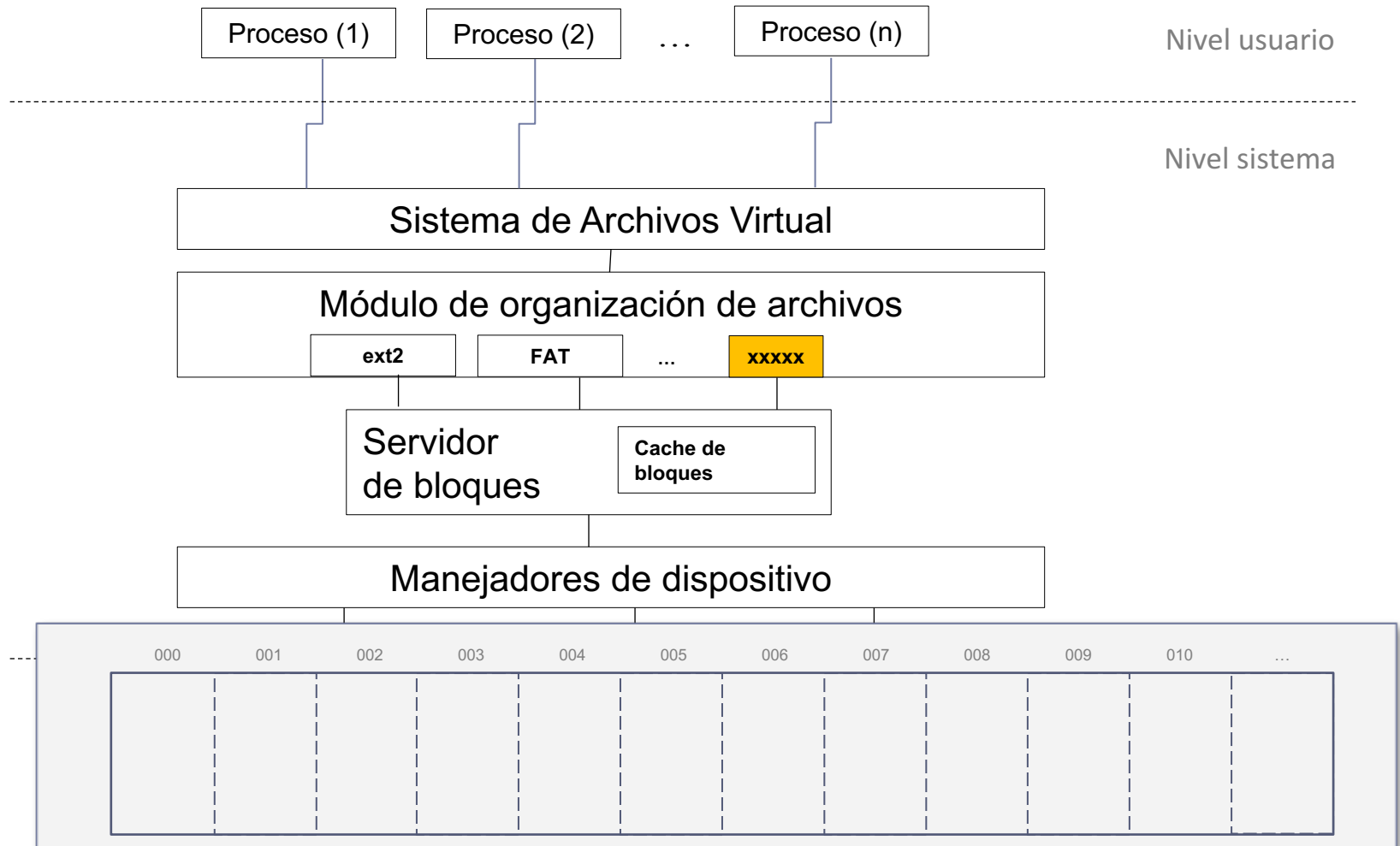
1. Introducción
2. Marco de trabajo
3. **Diseño y desarrollo de un sistema de ficheros**
4. Aspectos complementarios



# Aspectos a tener en cuenta de arquitectura de un sistema de ficheros...



# Aspectos a tener en cuenta de arquitectura de un sistema de ficheros...



# Aspectos a tener en cuenta de arquitectura de un sistema de ficheros...

- ▶ **getblk:** busca/reserva en caché un bloque de un v-nodo, con desplazamiento y tamaño dado.
- ▶ **brelease:** libera un buffer y lo pasa a la lista de libres.
- ▶ **bwrite:** escribe un bloque de la caché a disco.
- ▶ **bread:** lee un bloque de disco a caché.
- ▶ **breada:** lee un bloque (y el siguiente) de disco a caché.

ario

ema

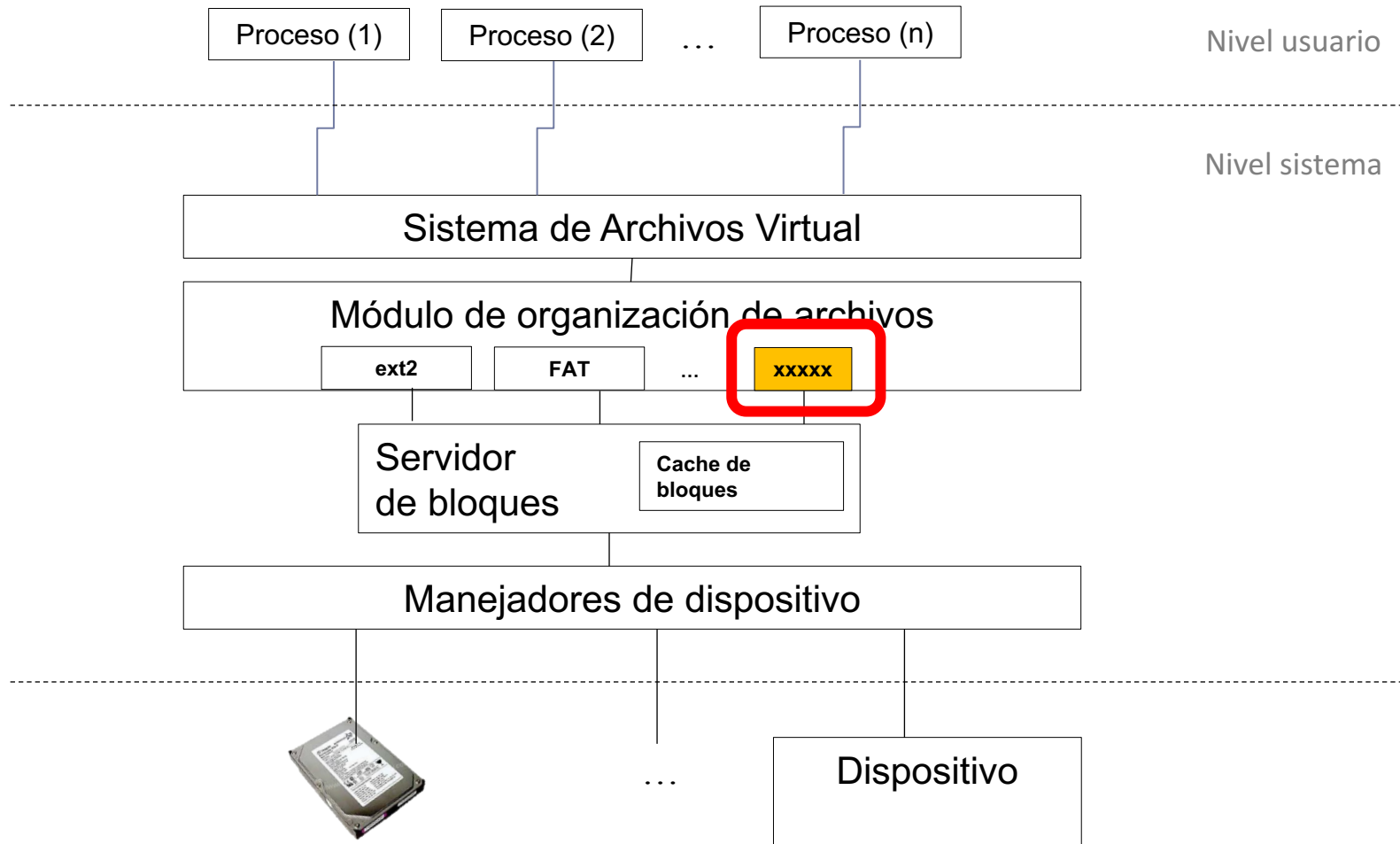
Servidor  
de bloques

Cache de  
bloques

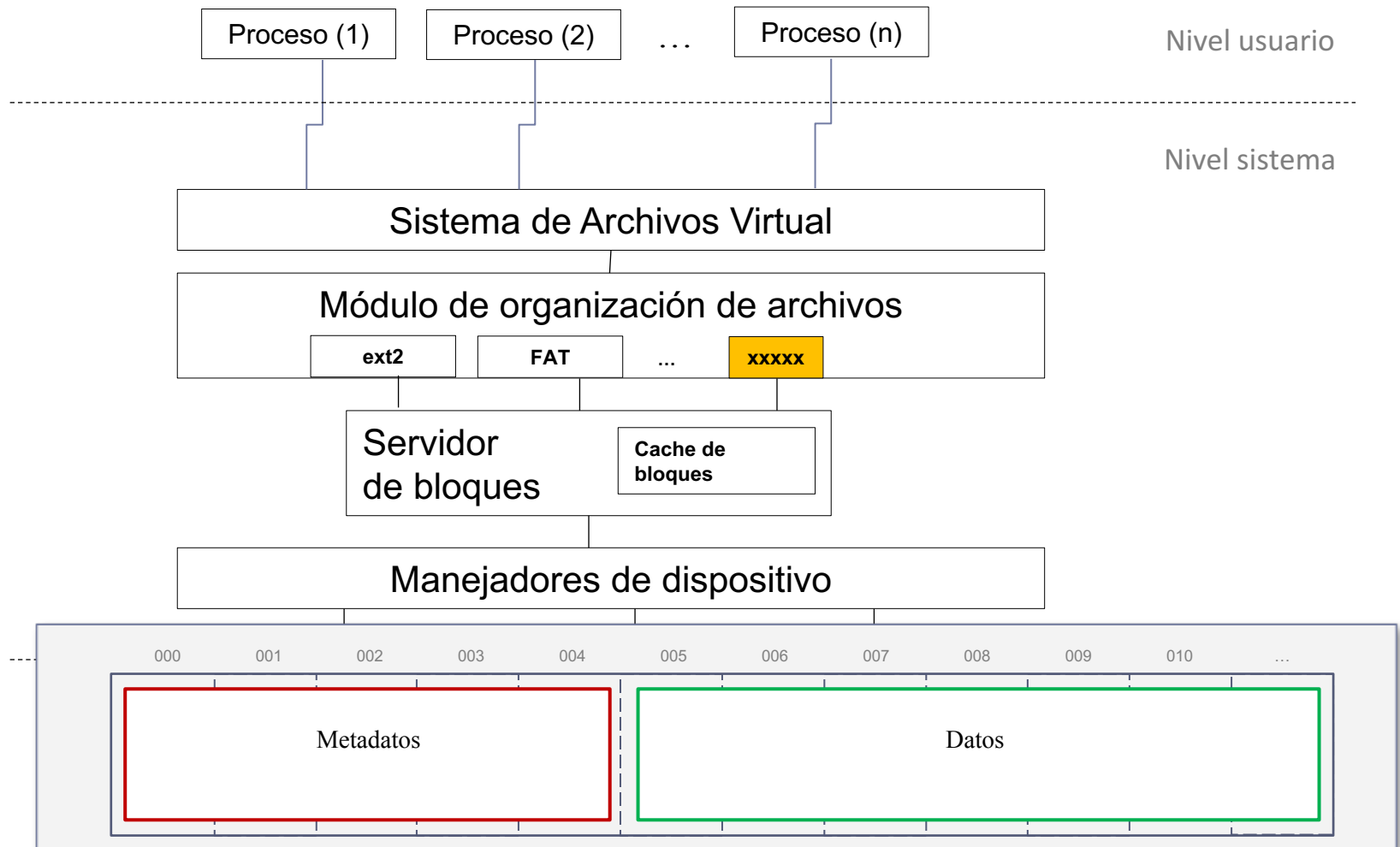
Manejadores de dispositivo

000 001 002 003 004 005 006 007 008 009 010 ...

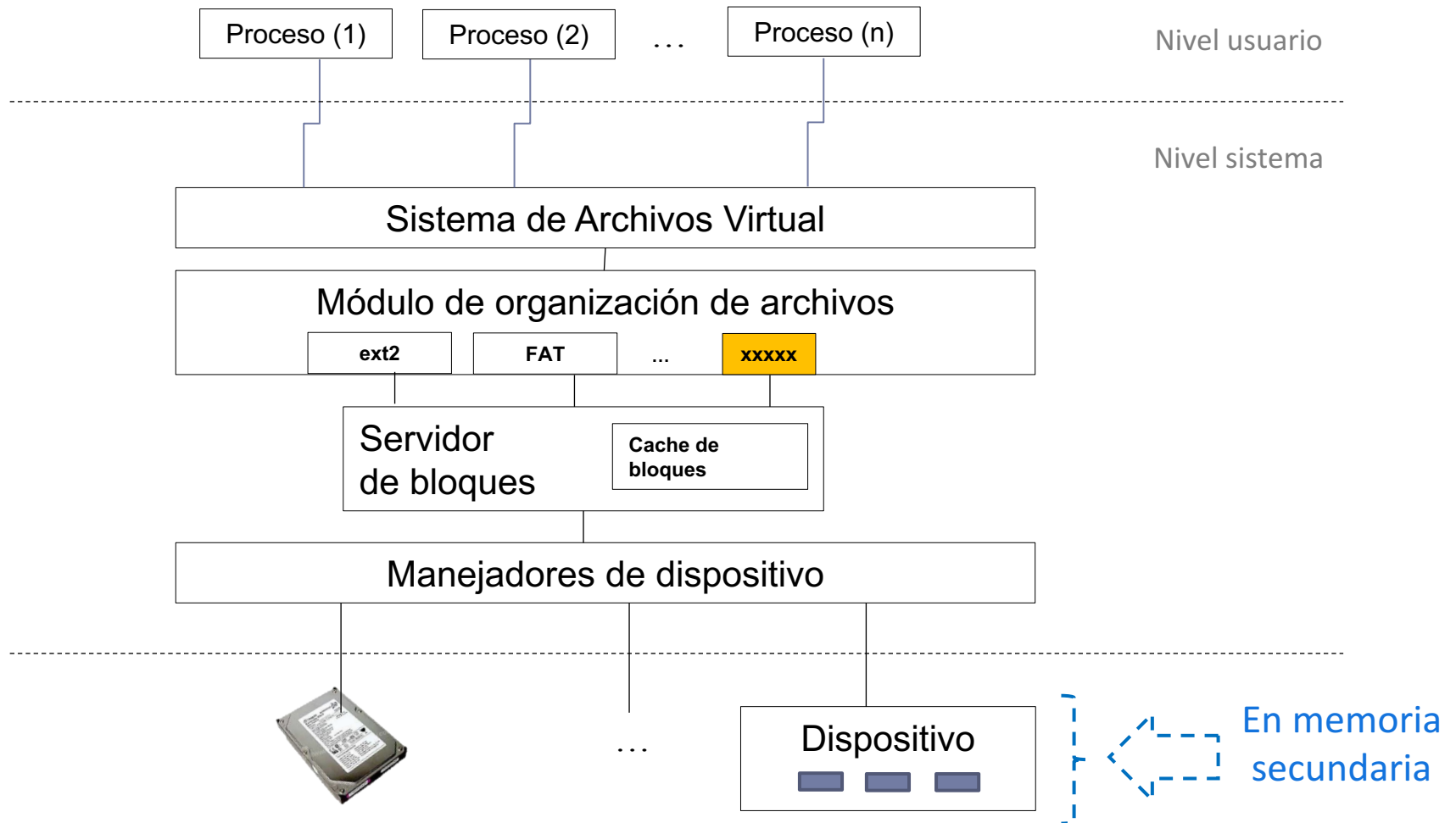
# Aspectos a tener en cuenta de arquitectura de un sistema de ficheros...



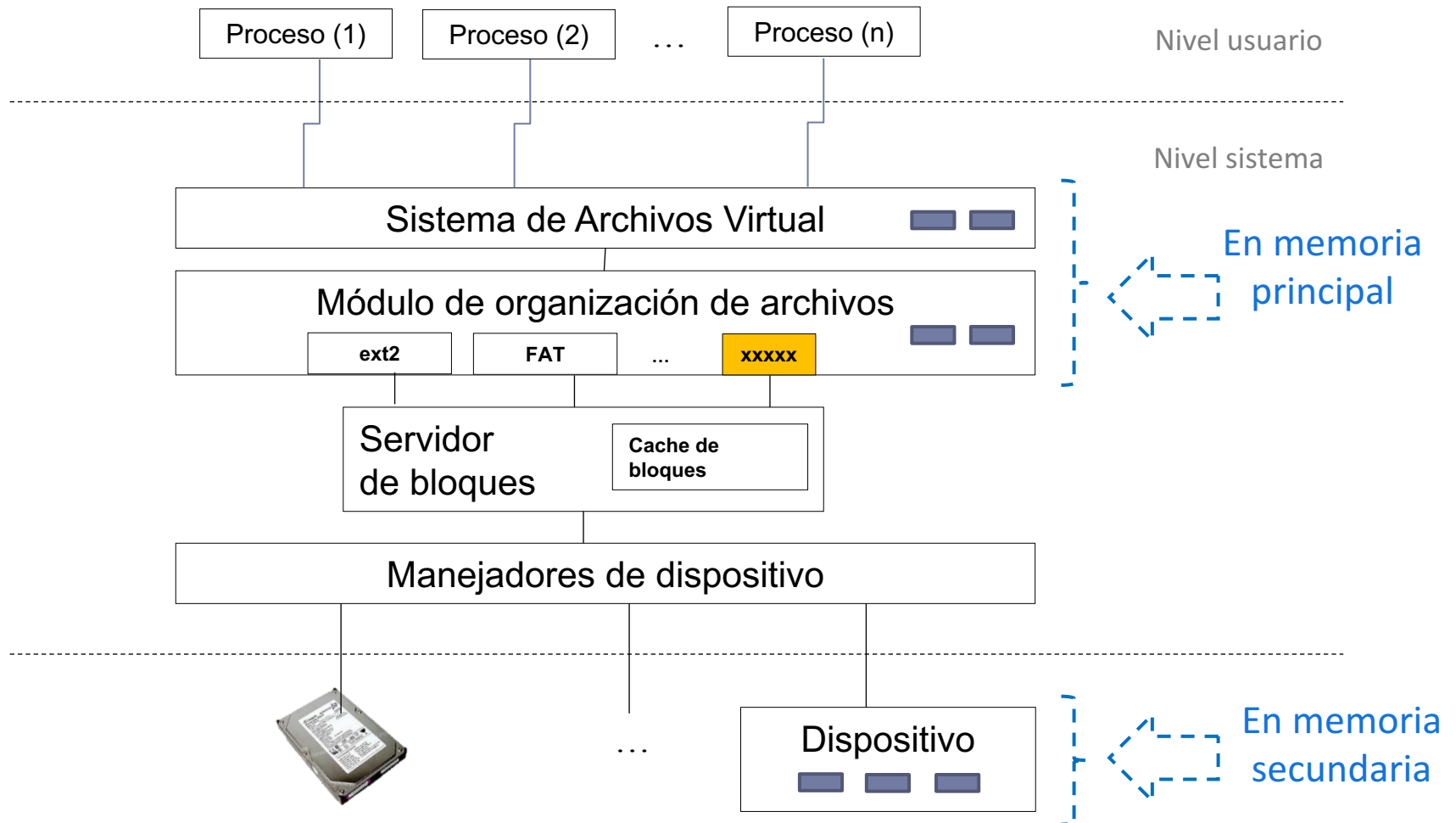
# (1) Estructuras de datos en disco...



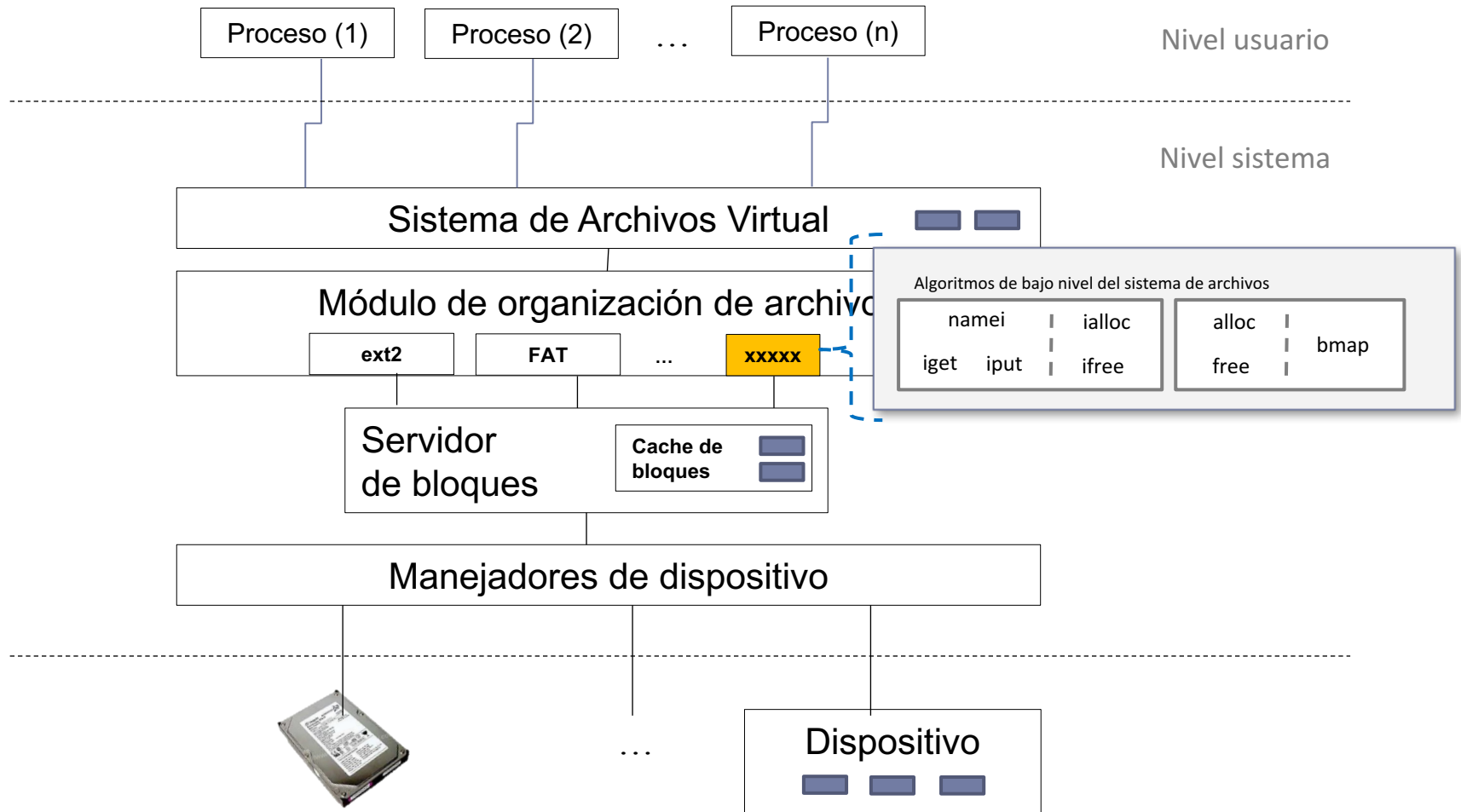
# (1) Estructuras de datos en disco...



## (2) Estructuras de datos en memoria...

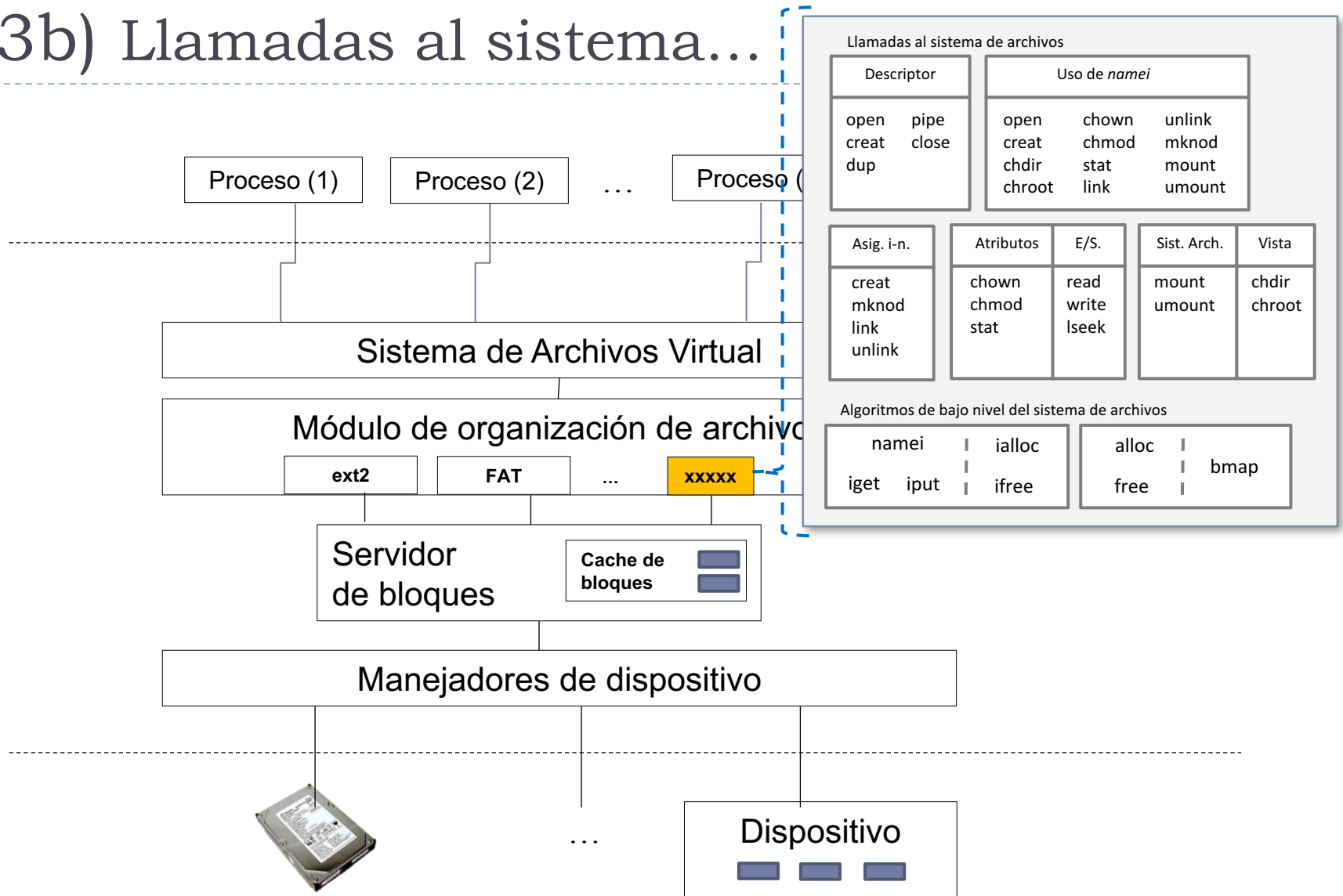


# (3a) Gestión de estructuras disco/memoria...

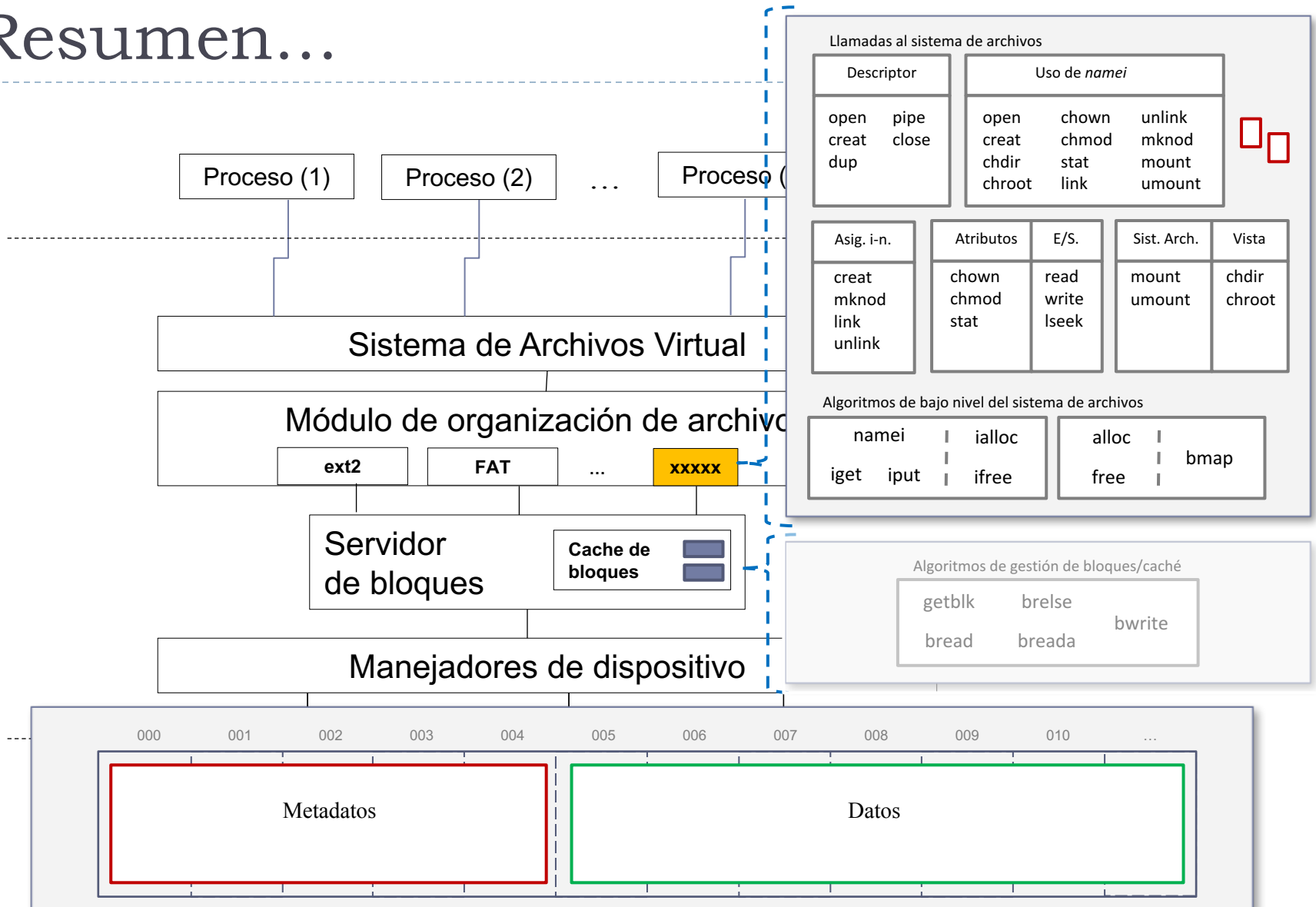




# (3b) Llamadas al sistema...



# Resumen...



# Resumen...

## Llamadas al sistema de archivos

Descriptor	Uso de <i>namei</i>	Asig. i-n.	Atributos	E/S.	Sist. Arch.	Vista
open pipe	open chown unlink	creat	chown	read	mount	chdir
creat close	creat chmod mknod	mknod	chmod	write	umount	chroot
dup	chdir stat mount	link	stat	lseek		
	chroot link umount	unlink				

xx

## Algoritmos de bajo nivel del sistema de archivos

namei	ialloc	alloc	bmap
iget	ifree	free	

d-entradas



punteros de posición

ficheros abiertos

montajes



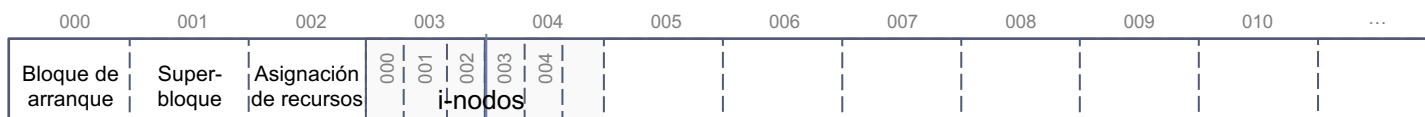
i-nodos en uso



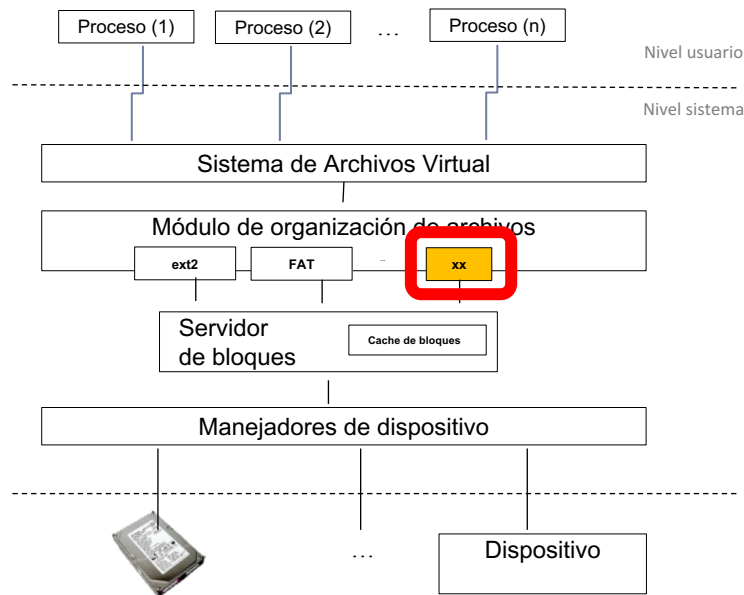
## Algoritmos de gestión de bloques/caché

getblk	brelse	bread	breada	bwrite
--------	--------	-------	--------	--------

módulos de s. ficheros

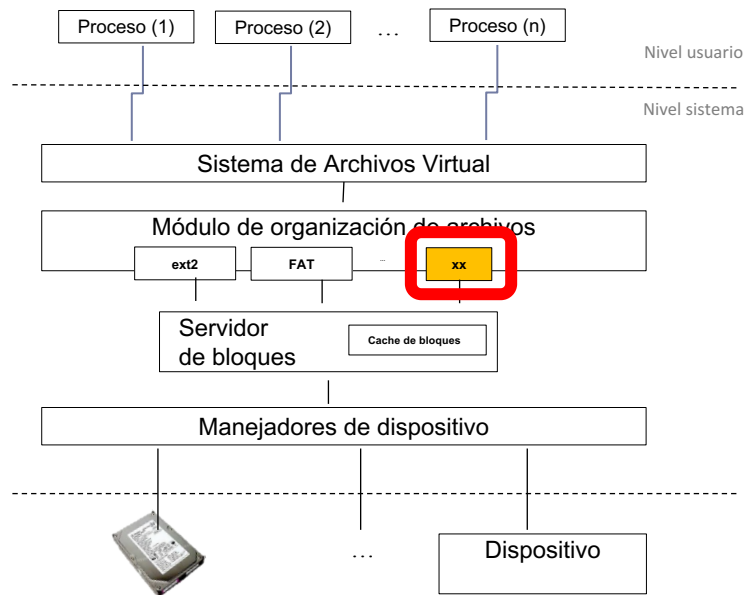


# Aspectos a tener en cuenta para añadir un sistema de ficheros...



- ▶ (0) Requisitos del sistema.
- ▶ (1) Estructuras en disco.
- ▶ (2) Estructuras en memoria.
- ▶ Caché de bloques.
- ▶ (3a) Funciones de gestión de estructuras disco/memoria.
- ▶ (3b) Funciones de llamadas al sistema.

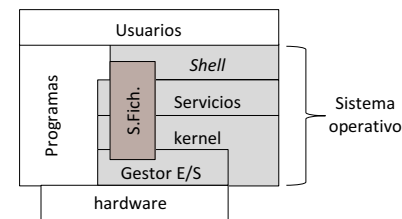
# Aspectos a tener en cuenta para añadir un sistema de ficheros...



- ▶ **(0) Requisitos del sistema.**
- ▶ **(1) Estructuras en disco.**
- ▶ **(2) Estructuras en memoria.**
- ▶ **Caché de bloques.**
- ▶ **(3a) Funciones de gestión de estructuras disco/memoria.**
- ▶ **(3b) Funciones de llamadas al sistema.**

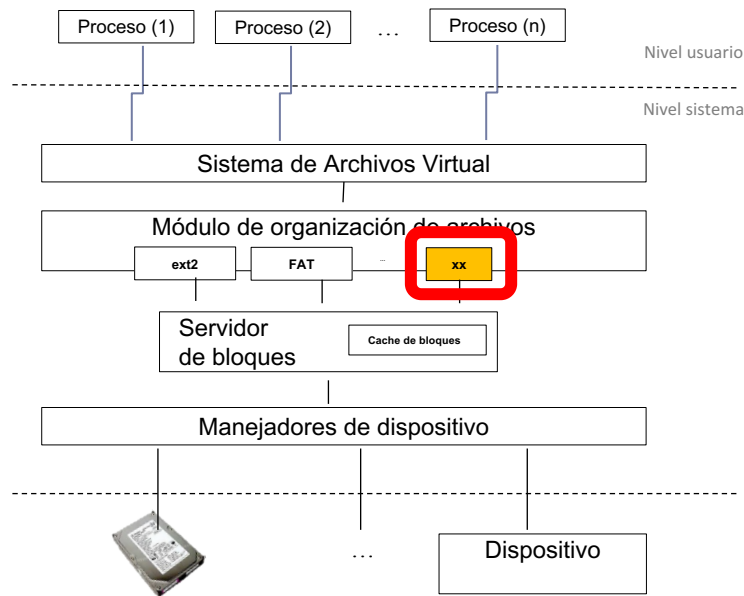
# (0) ~~Objetivos~~ requisitos principales

ej.: sistema de ficheros tipo Unix



- ▶ Lograr la **persistencia de los datos del usuario**, buscando **minimizar el impacto en el rendimiento y en el espacio para metadatos**.
- ▶ Los procesos usarán una **interfaz de trabajo segura**, sin acceso directo a la información usada en el kernel.
- ▶ **Compartir el puntero de posición de ficheros** entre procesos con relación de parentesco.
- ▶ Poder tener **una sesión de trabajo con un fichero/directorio** para actualizar la información que contiene.
- ▶ Poder tener **una sesión de trabajo con varios directorios** para poder recorrer sus entradas.
- ▶ **Llevar la pista de los sistemas de ficheros presentes** en el kernel, **y llevar la pista de los puntos donde están siendo usados**.

# Aspectos a tener en cuenta para añadir un sistema de ficheros...



- ▶ (0) Requisitos del sistema.
- ▶ **(1) Estructuras en disco.**
- ▶ (2) Estructuras en memoria.
- ▶ Caché de bloques.
- ▶ (3a) Funciones de gestión de estructuras disco/memoria.
- ▶ (3b) Funciones de llamadas al sistema.

# (1) Estructuras de datos en disco...

Llamadas al sistema de archivos

Descriptor	Uso de <i>namei</i>	Asig. i-n.	Atributos	E/S.	Sist. Arch.	Vista
open pipe	open chown unlink	creat	chown	read	mount	chdir
creat close	creat chmod mknod	mknod	chmod	write	umount	chroot
dup	chdir stat mount	link	stat	lseek		
	chroot link umount	unlink				

x

Algoritmos de bajo nivel del sistema de archivos

namei	ialloc	alloc	bmap
iget	ifree	free	

d-entradas



punteros de posición

ficheros abiertos

montajes



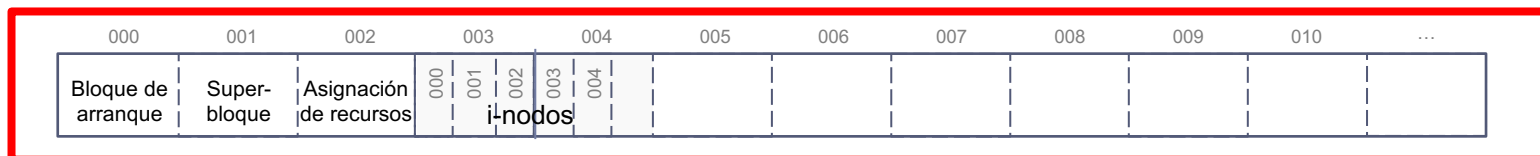
i-nodos en uso



Algoritmos de gestión de bloques/caché

getblk	brelse	bread	breada	bwrite
--------	--------	-------	--------	--------

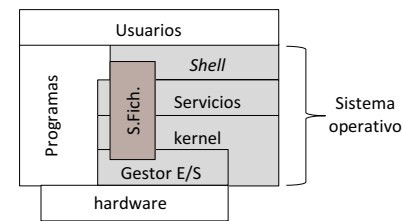
módulos de s. ficheros





# (0) ~~Objetivos~~ requisitos principales

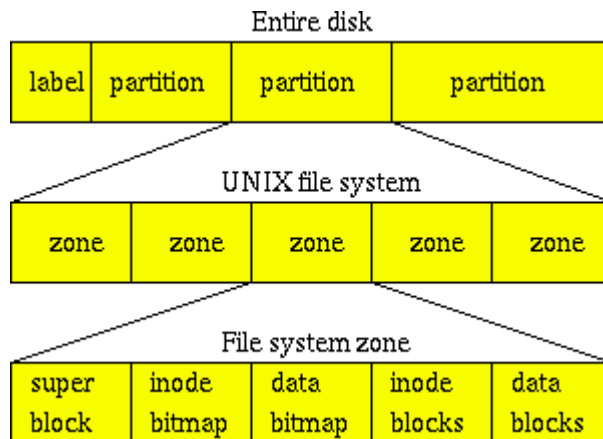
ej.: sistema de ficheros tipo Unix



- ▶ Lograr la **persistencia de los datos del usuario**, buscando **minimizar el impacto en el rendimiento y en el espacio para metadatos**.
- ▶ Los procesos usarán una **interfaz de trabajo segura**, sin acceso directo a la información usada en el kernel.
- ▶ **Compartir el puntero de posición de ficheros** entre procesos con relación de parentesco.
- ▶ Poder tener **una sesión de trabajo con un fichero/directorio** para actualizar la información que contiene.
- ▶ Poder tener **una sesión de trabajo con varios directorios** para poder recorrer sus entradas.
- ▶ **Llevar la pista de los sistemas de ficheros presentes** en el kernel, **y llevar la pista de los puntos donde están siendo usados**.

# Estructuras del sistema de ficheros

---



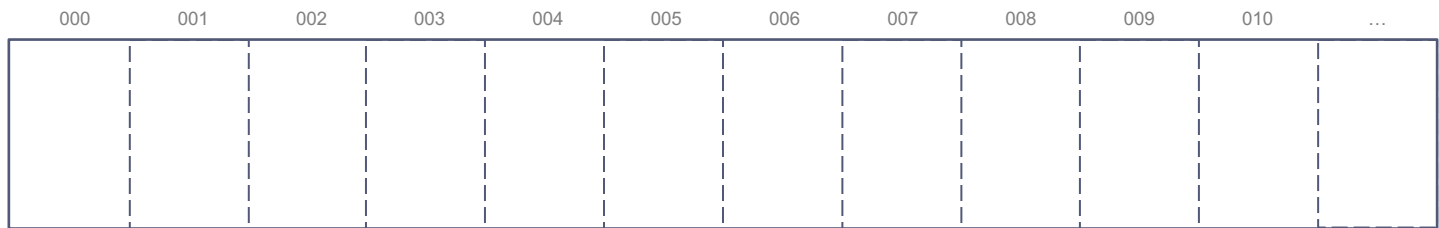
▶ UNIX/Linux

▶ FAT

# Sistema de ficheros: representación tipo Unix

---

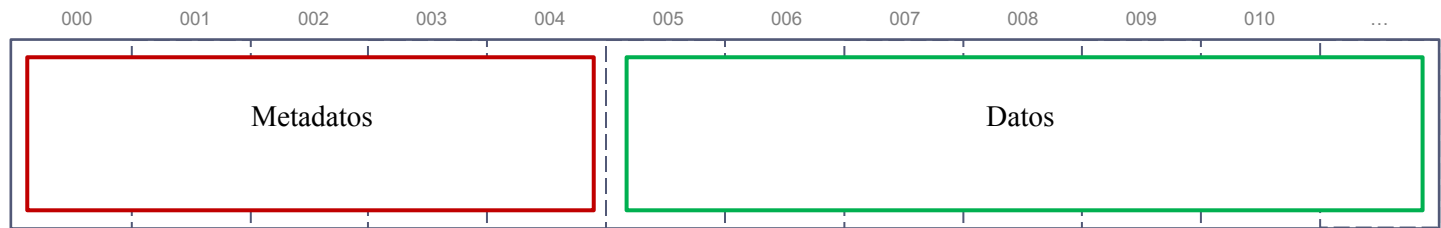
Disco lógico



# Sistema de ficheros: representación tipo Unix

---

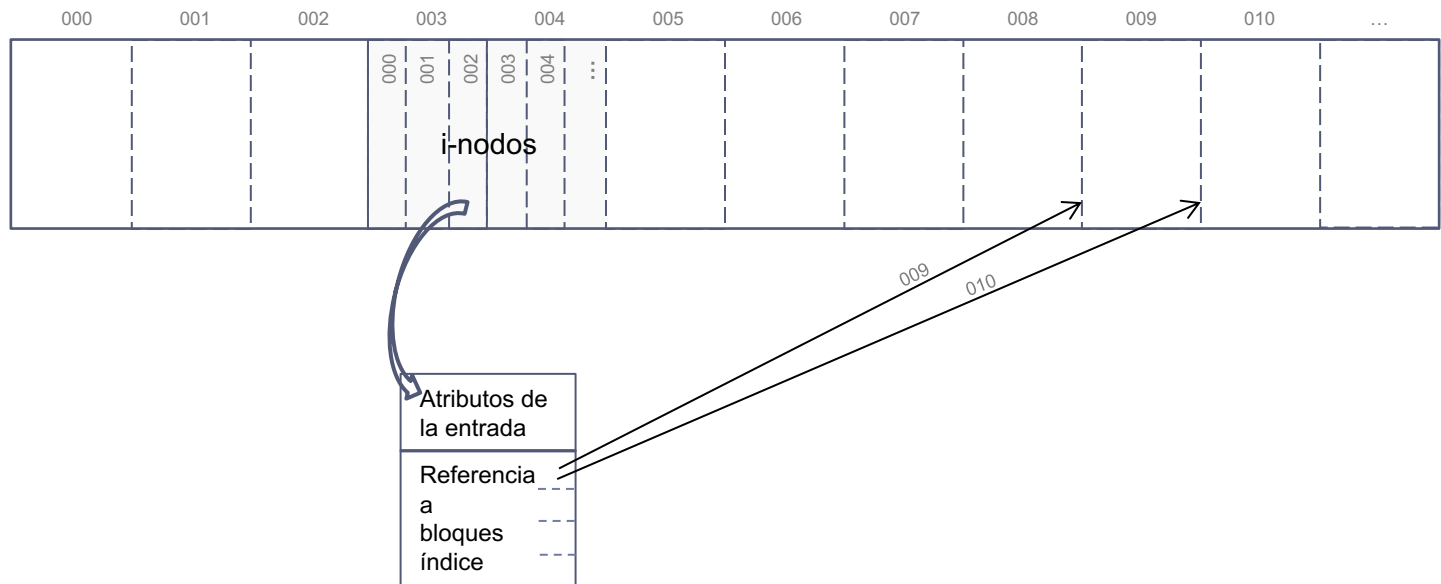
Disco lógico



# Sistema de ficheros: representación tipo Unix

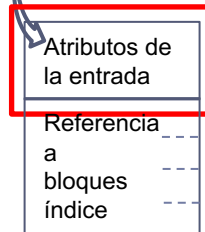
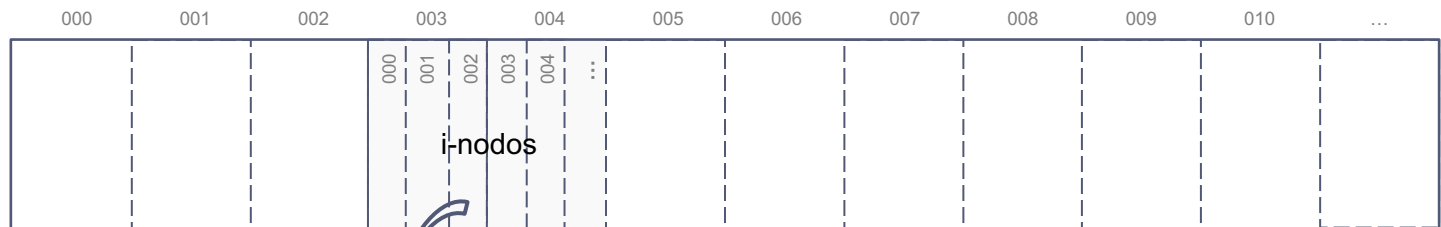
---

Disco lógico



# Sistema de ficheros: representación tipo Unix

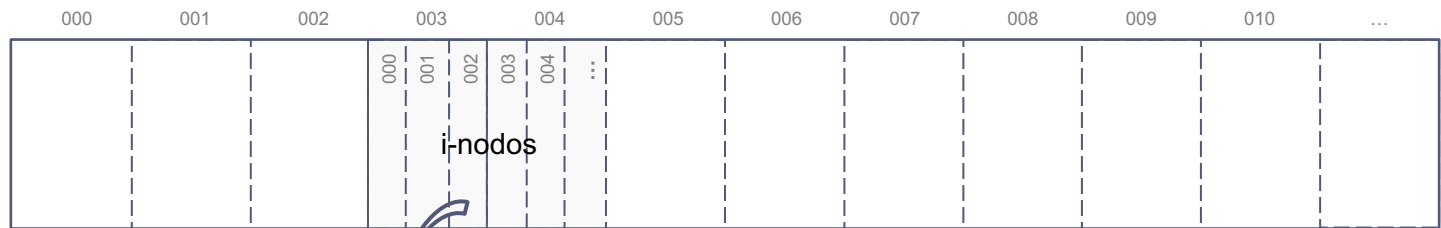
Disco lógico



- ▶ **Fechas:**
  - ▶ De creación, modificación, acceso, etc.
- ▶ **Tamaño:**
  - ▶ En bytes o bloques de disco usado.
- ▶ **Propietario y protección:**
  - ▶ Atributos, ACL, capacidades, etc.
- ▶ **Tipo de fichero, contador de enlace, etc.**

# Sistema de ficheros: representación tipo Unix

Disco lógico

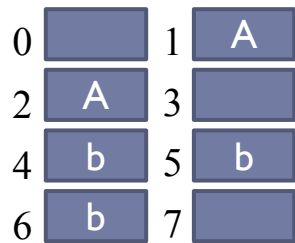


- ▶ **Asignación continua:**
  - ▶ Se asigna una lista de bloques consecutivos.
- ▶ **Asignación discontinua:**
  - ▶ Se asigna el primer bloque disponible.
  - ▶ **Mecanismo enlazado:**
    - ▶ Al final de cada bloque se indica el siguiente.
  - ▶ **Mecanismo indexado:**
    - ▶ Bloques con índices de todos los bloques de la entrada.

# Sistema de ficheros:

## representación de la asignación de recursos

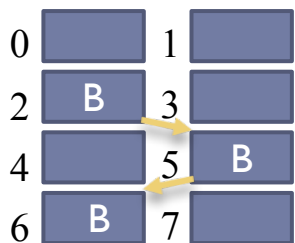
---



F	I	L
A	1	2
B	4	3

### ▶ Asignación **contigua**:

- ▶ Los bloques del ficheros están consecutivamente.
- ▶ Precisa: primero (I) y nº de bloques (L)
- ▶ Compactar.



F	I	L
B	2	3

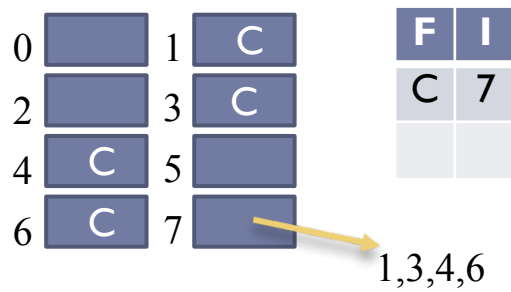
### ▶ Asignación **encadenada**:

- ▶ Cada bloque contiene la referencia al siguiente.
- ▶ Precisa: primero (I) y nº de bloques (L)
- ▶ Desfragmentar.



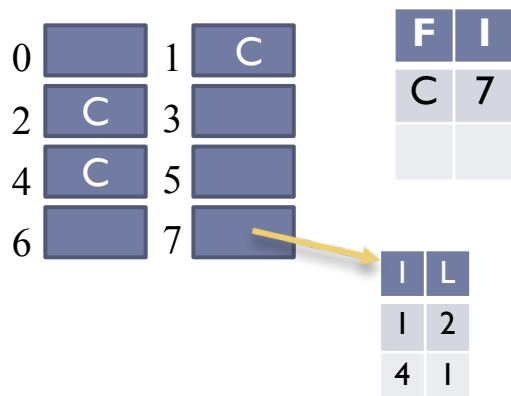
# Sistema de ficheros: representación de la asignación de recursos

---



## ▶ Asignación **indexada** (bloques):

- ▶ Se usa bloques con referencias a los bloques que contendrán los datos.
- ▶ Precisa: id. del 1<sup>er</sup> bloque índice.
- ▶ Desfragmentar.

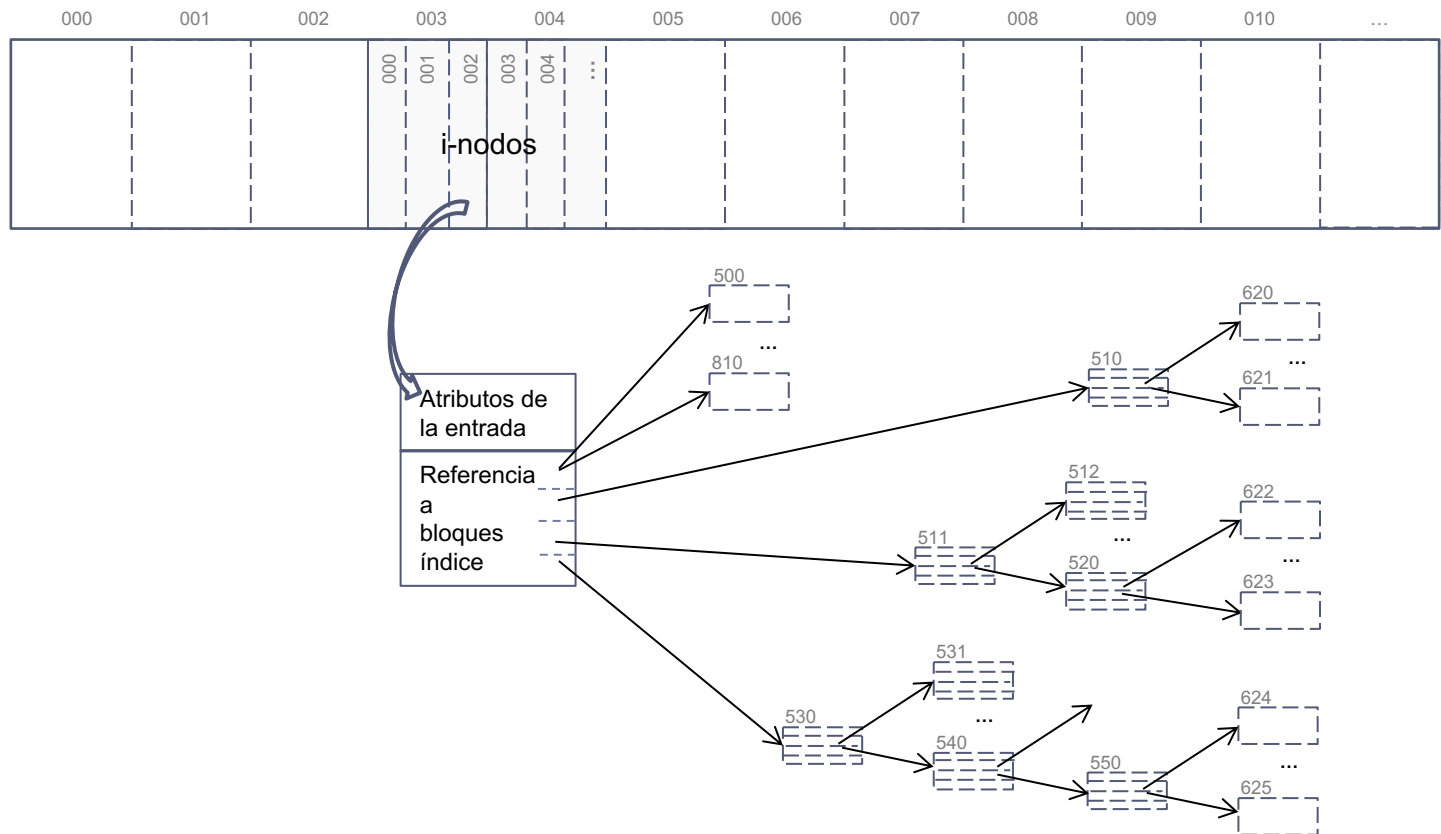


## ▶ Asignación **indexada** (extends):

- ▶ Se usa bloques con referencias al comienzo a los bloques que contendrán los datos.
- ▶ Precisa: id. del 1<sup>er</sup> bloque índice.
- ▶ Desfragmentar.

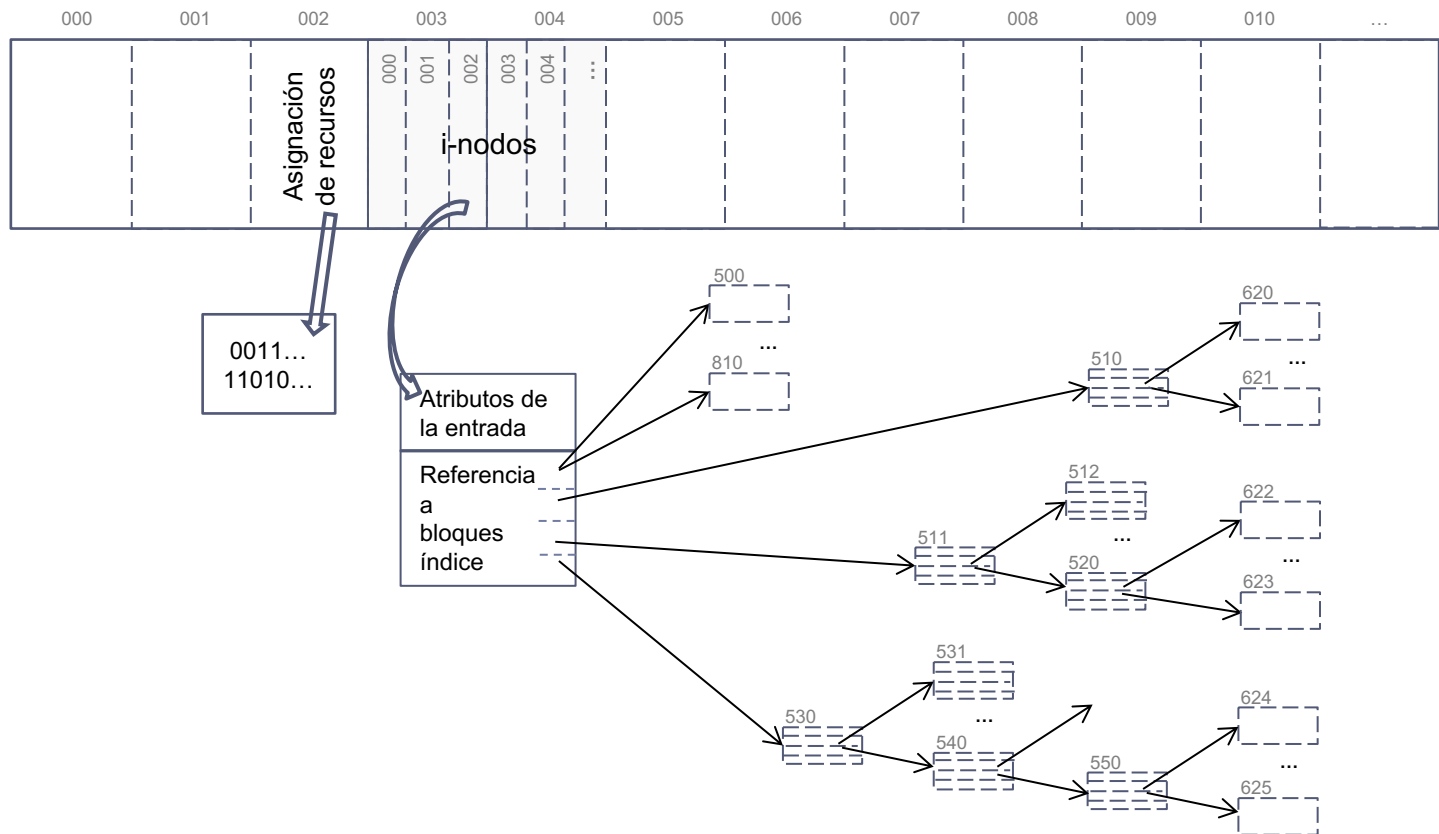
# Sistema de ficheros: representación tipo Unix

Disco lógico



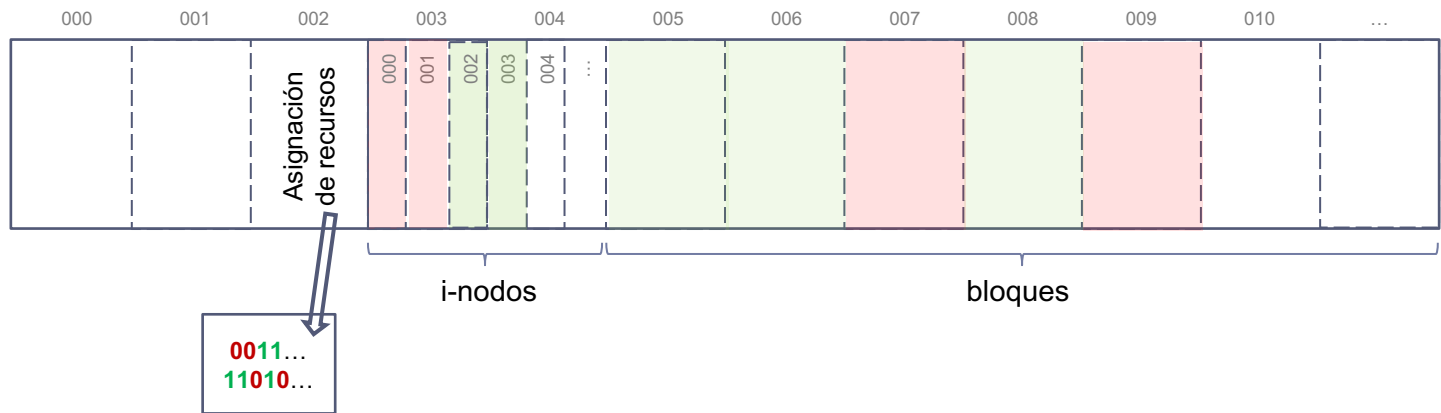
# Sistema de ficheros: representación tipo Unix

Disco lógico



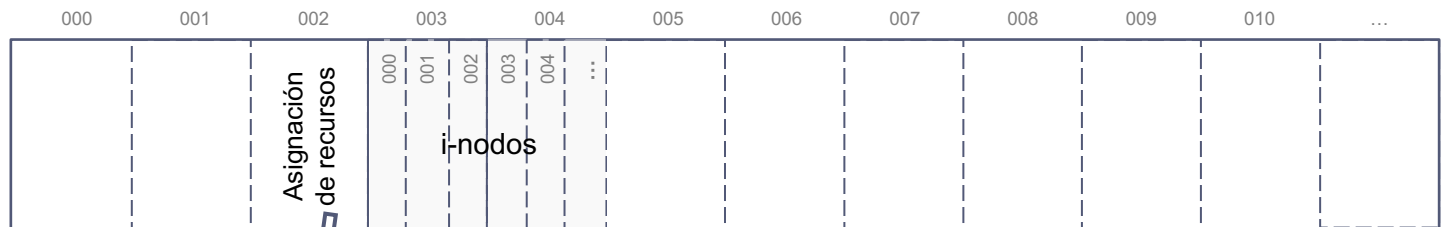
# Sistema de ficheros: representación tipo Unix

Disco lógico



# Sistema de ficheros: representación tipo Unix

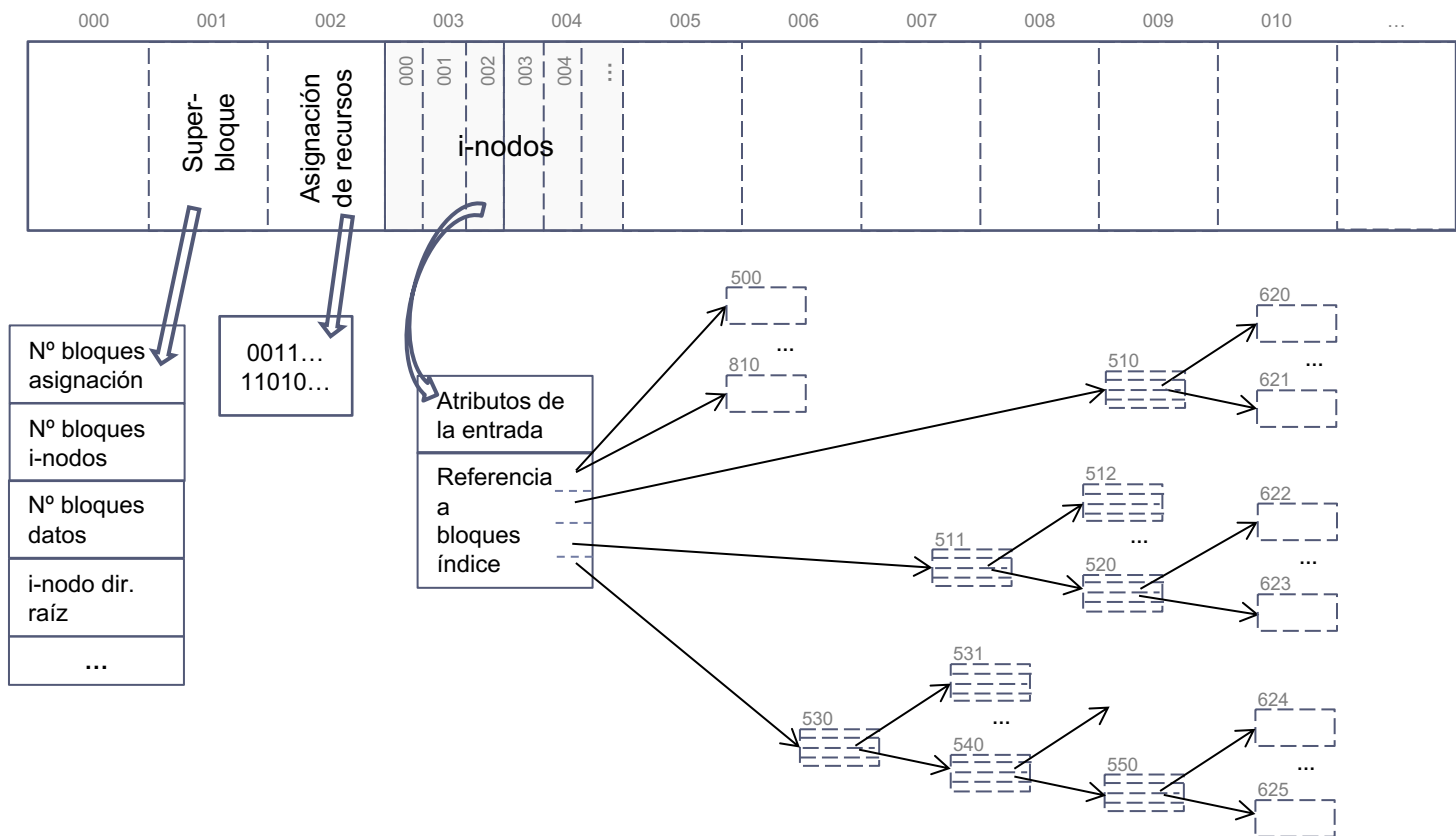
Disco lógico



- ▶ **Mapas de bits** o vectores de bits:
  - ▶ Vector con un bit por recurso existente.  
Si recurso libre entonces bit con valor 1, si ocupado valor es 0.
    - ▶ Fácil de implementar y sencillo de usar.
    - ▶ Eficiente si el dispositivo no está muy lleno o muy fragmentado.
- ▶ **Lista de recursos libres:**
  - ▶ Mantener enlazados en una lista todos los recursos disponibles manteniendo un apuntador al primer elemento de la lista.
    - ▶ Método no eficiente, excepto para dispositivos muy llenos y fragmentados
- ▶ **Indexación:**
  - ▶ Tabla índice de porciones libres.

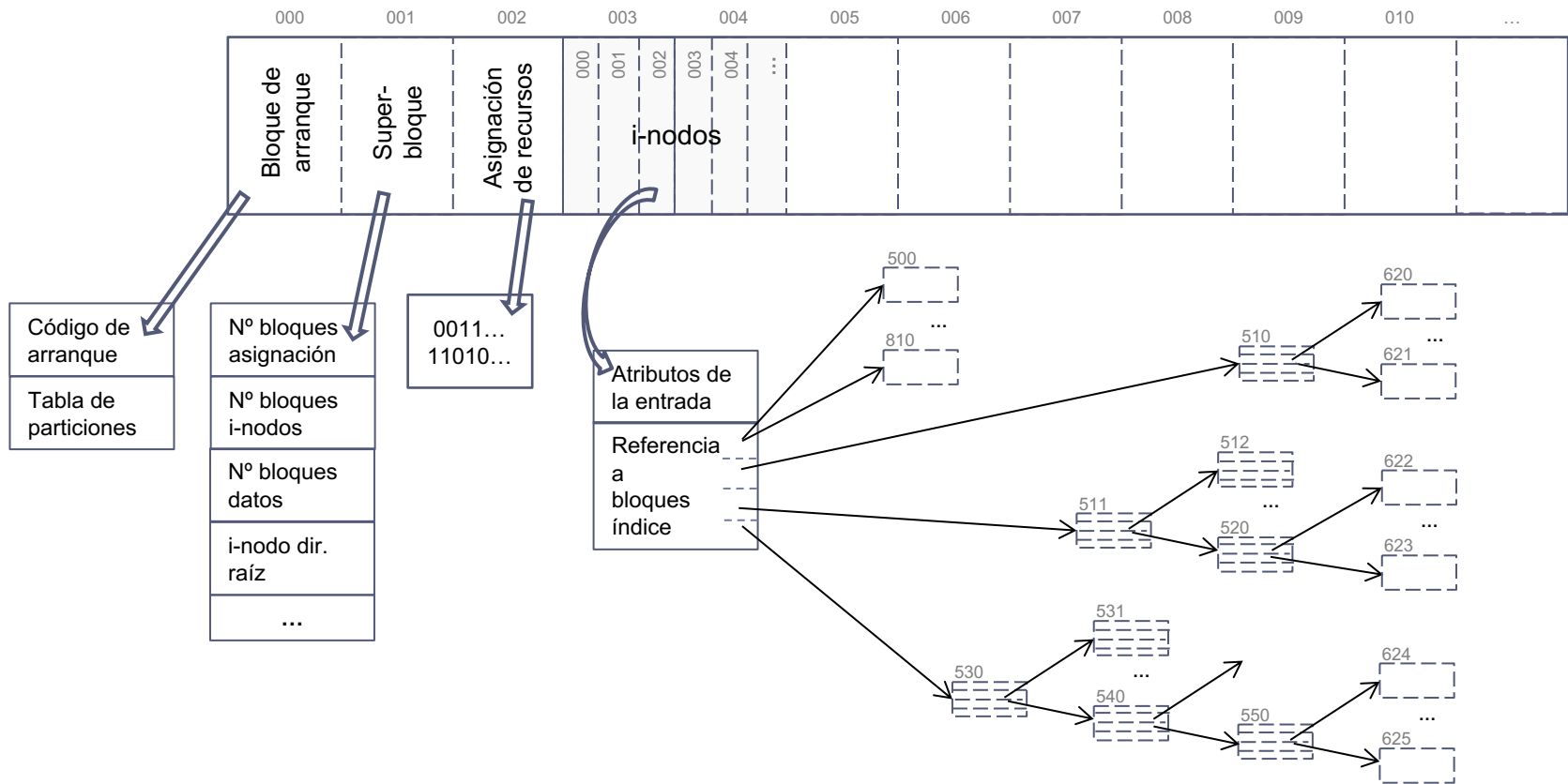
# Sistema de ficheros: representación tipo Unix

Disco lógico



# Sistema de ficheros: representación tipo Unix

Disco lógico



# Ejemplos de representaciones

---



▶ Ficheros



▶ Directorios

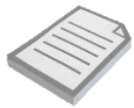


▶ Enlaces



# Ejemplos de representaciones

---



▶ Ficheros

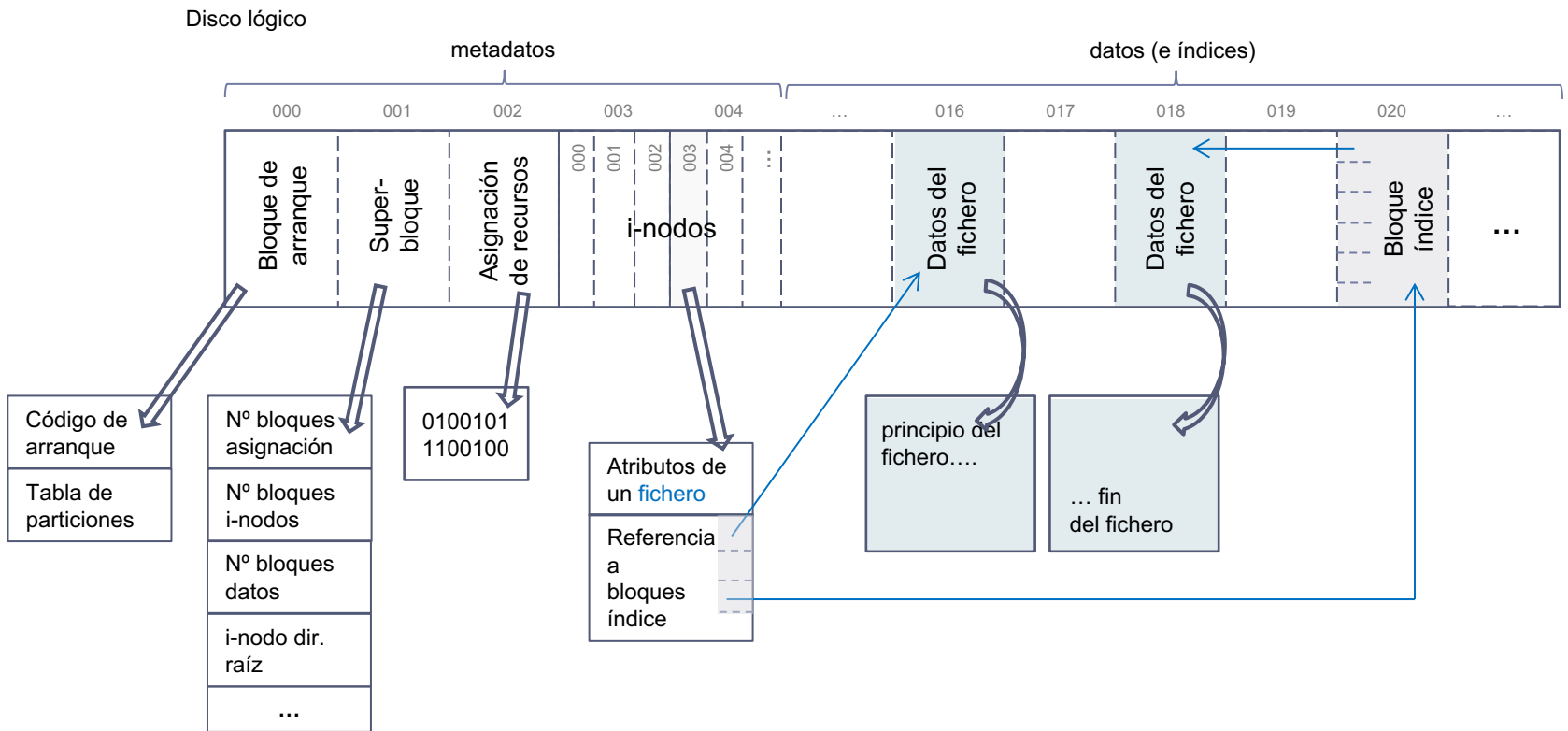


▶ Directorios



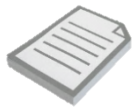
▶ Enlaces

# Sistema de ficheros: representación tipo Unix: **ficheros**



# Ejemplos de representaciones

---



▶ Ficheros

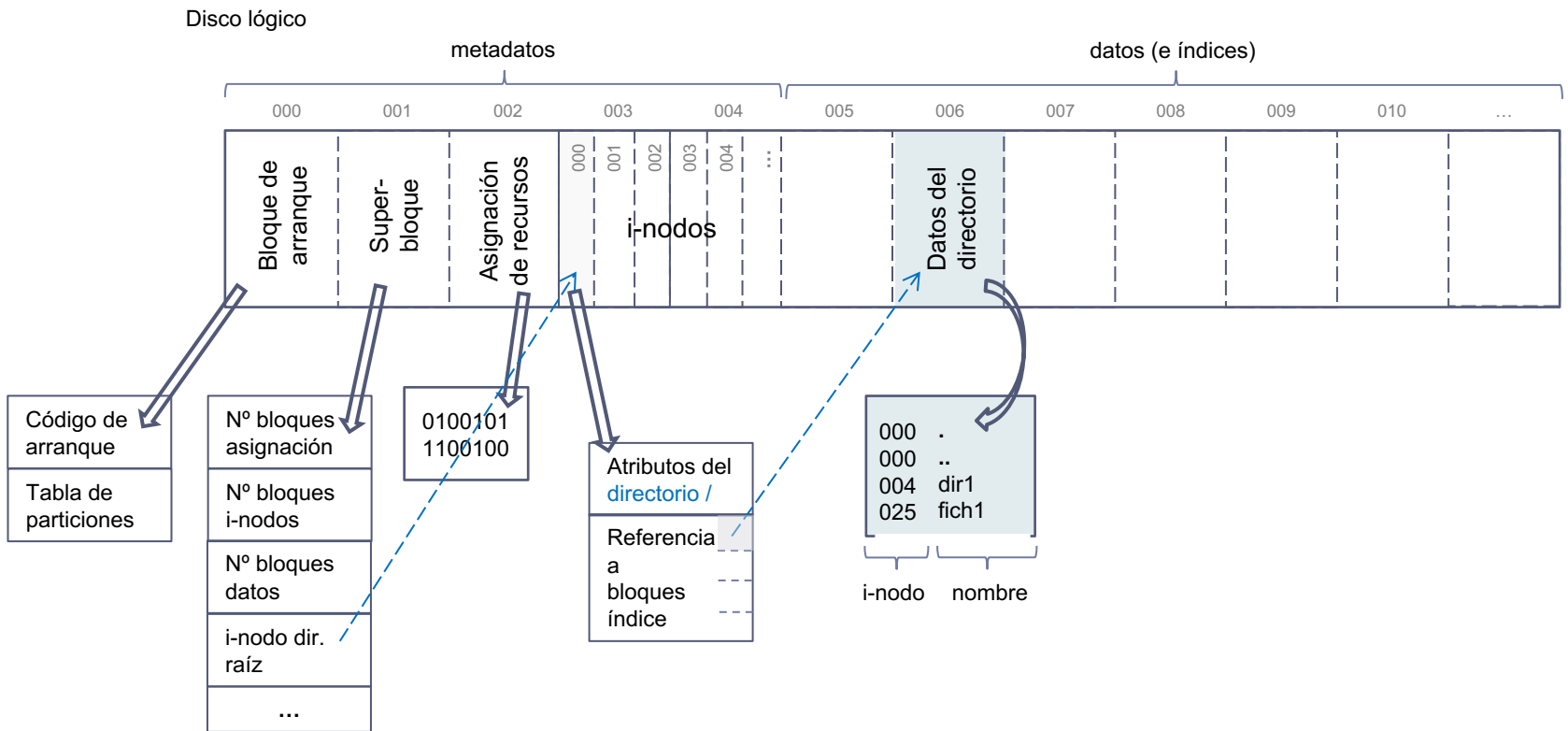


▶ Directorios

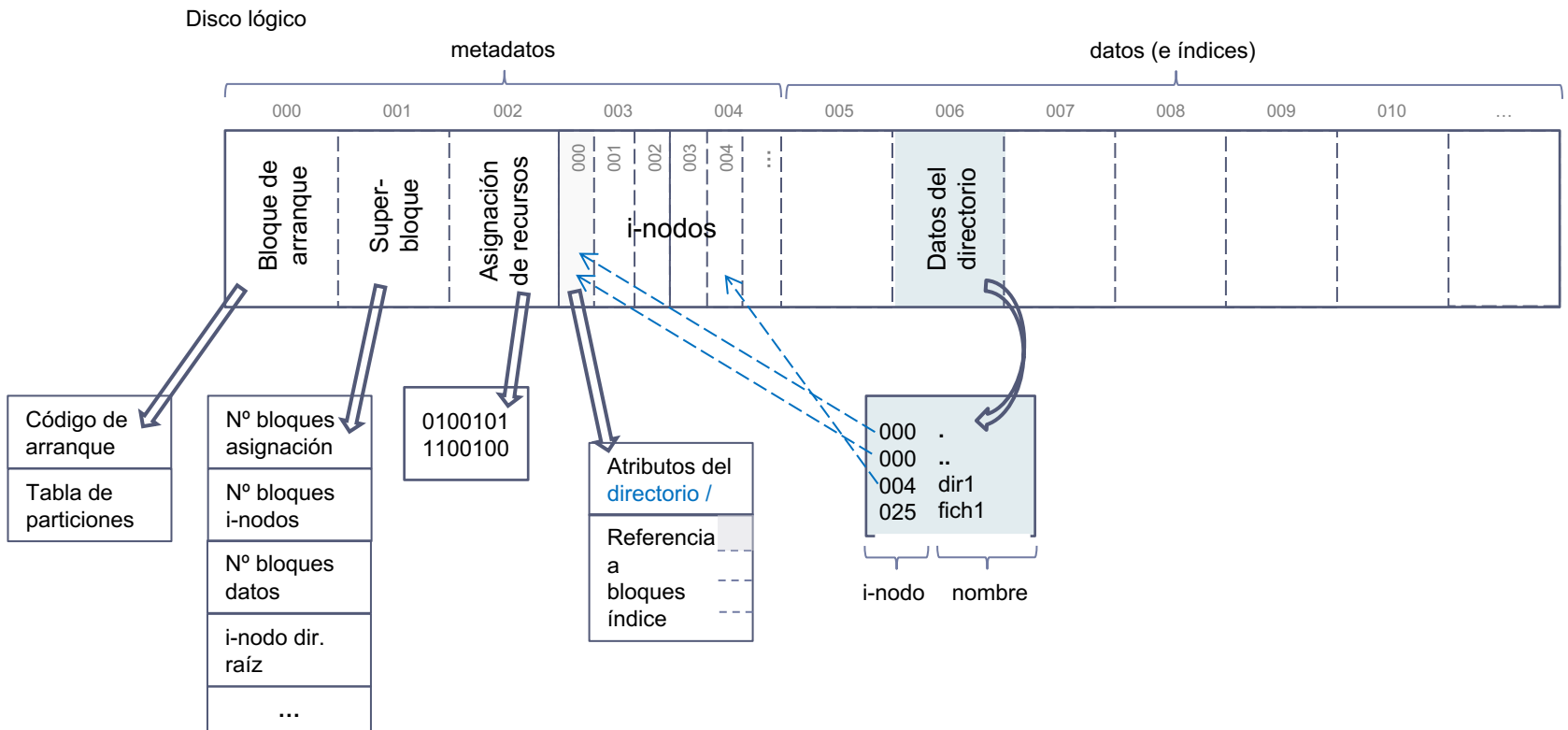


▶ Enlaces

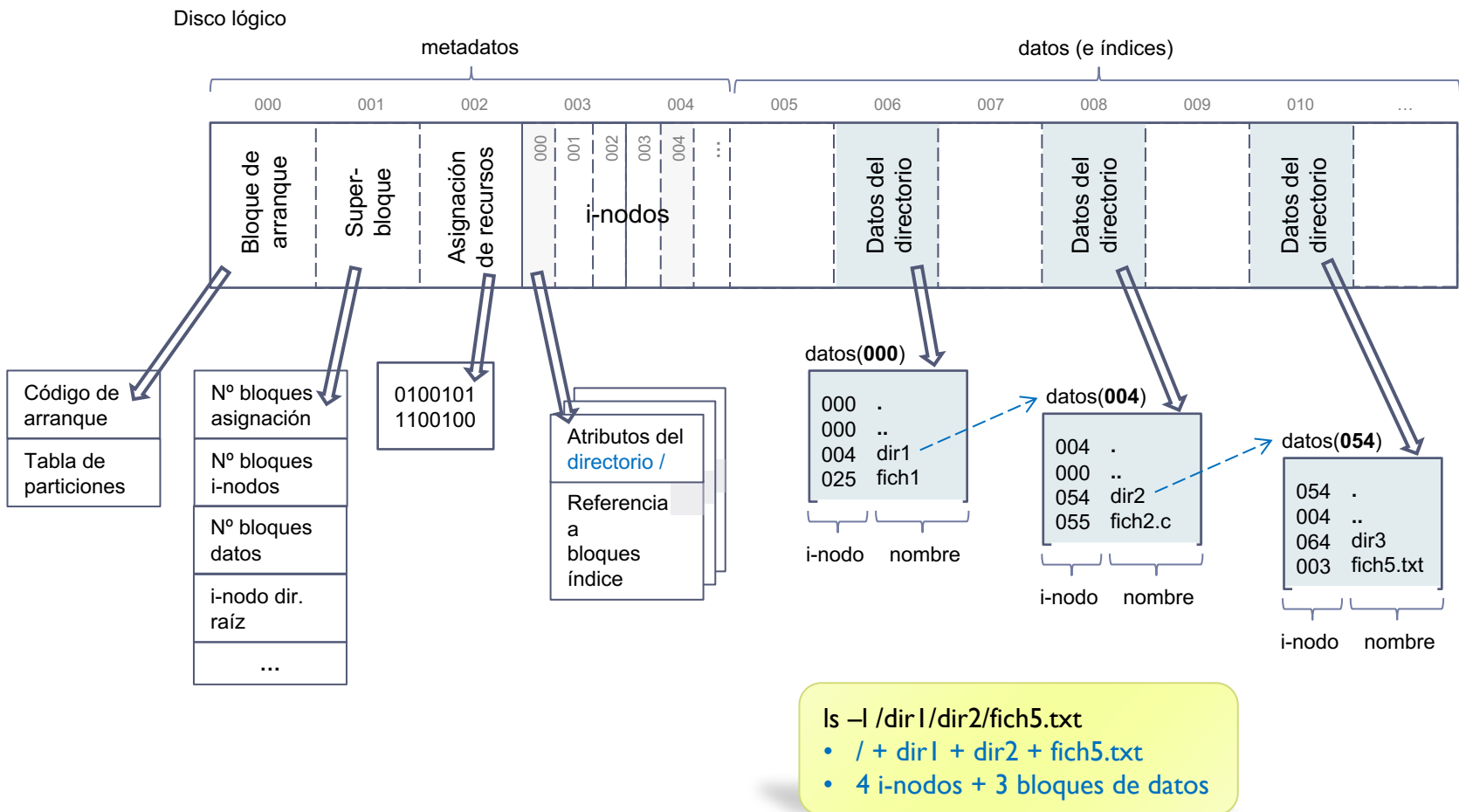
# Sistema de ficheros: representación tipo Unix: directorios



# Sistema de ficheros: representación tipo Unix: directorios

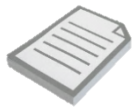


# Sistema de ficheros: representación tipo Unix: directorios



# Ejemplos de representaciones

---



▶ Ficheros



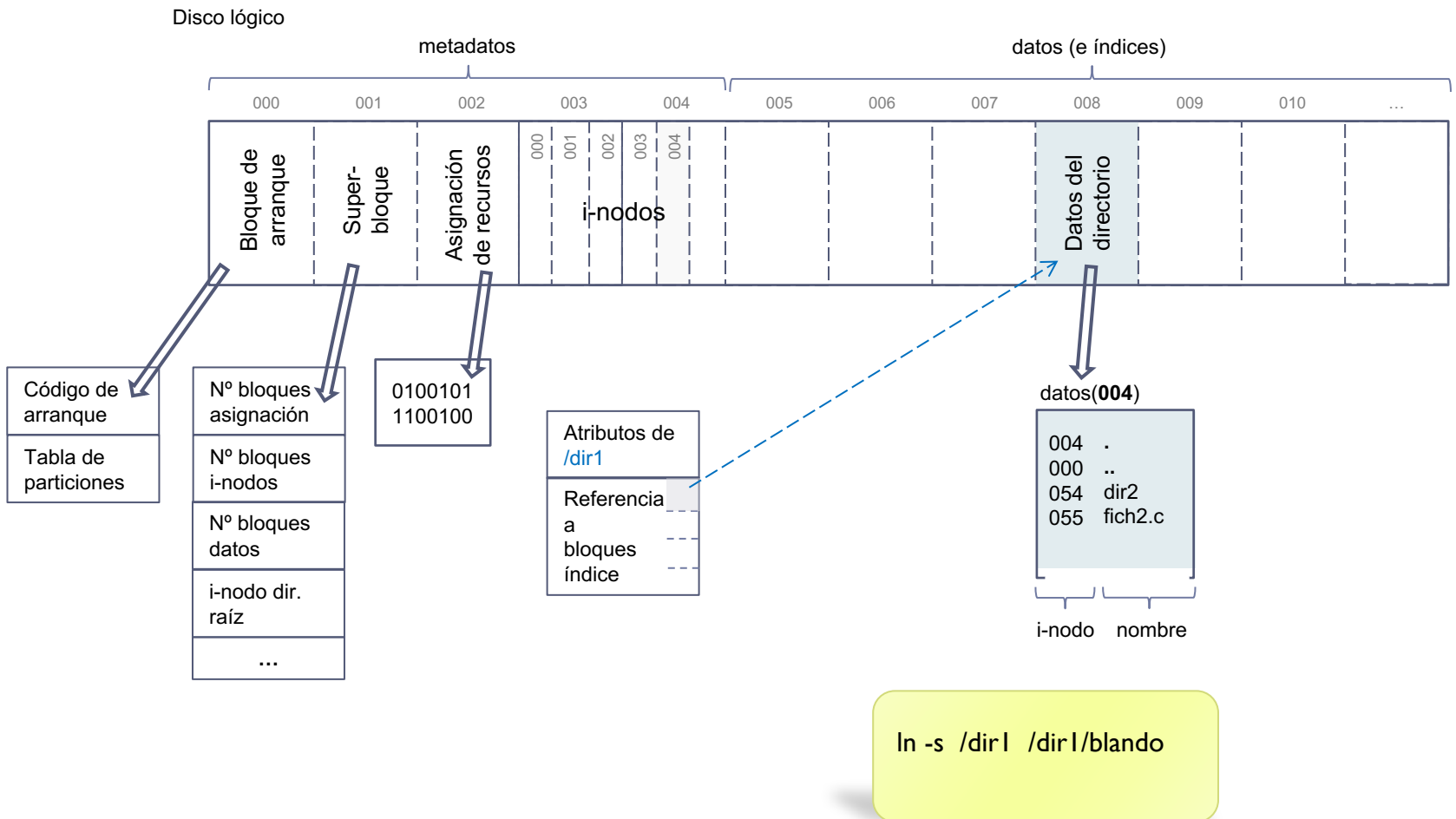
▶ Directorios



▶ Enlaces

# Sistema de ficheros:

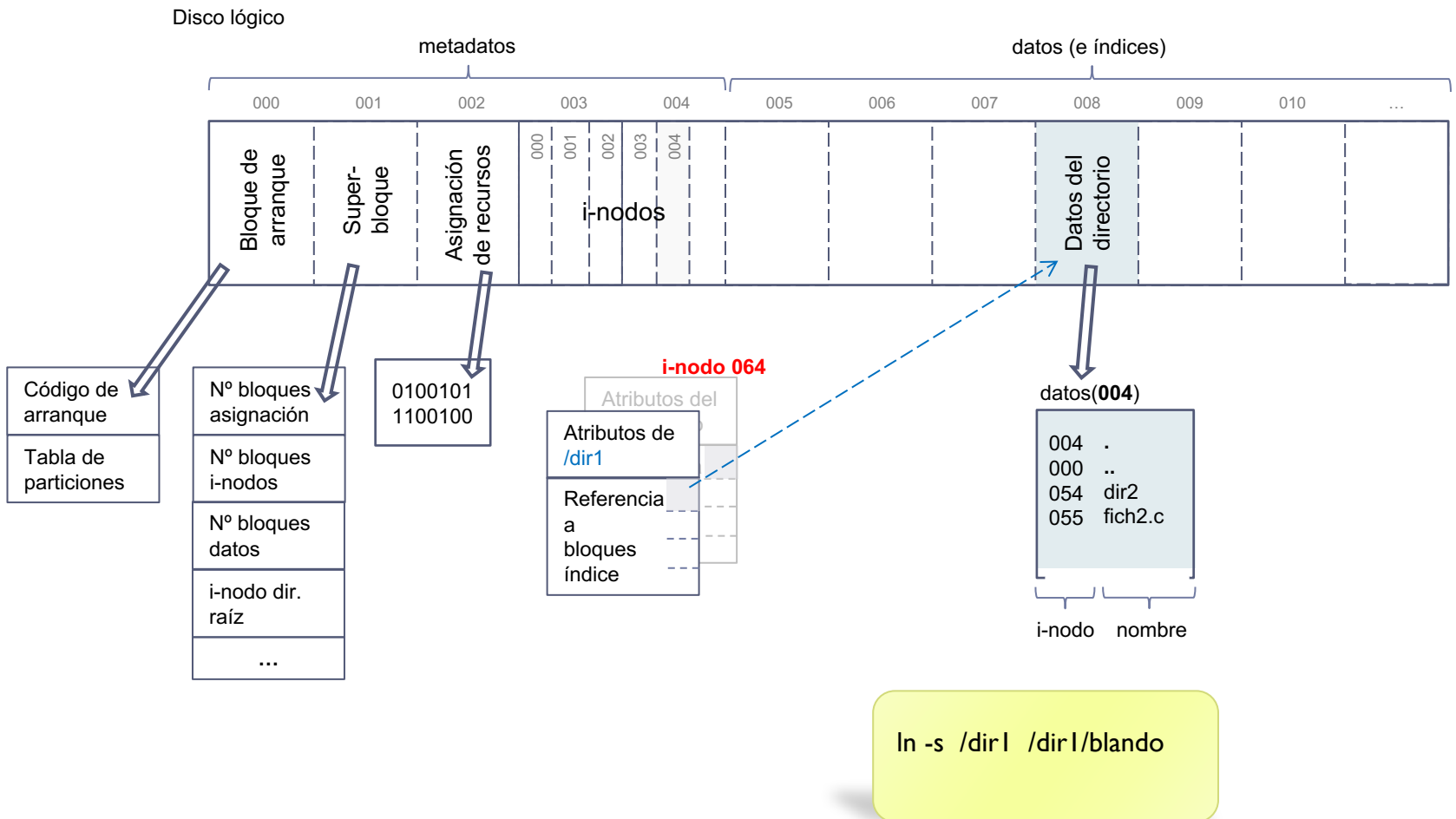
## representación tipo Unix: **enlace simbólico (o blando)**





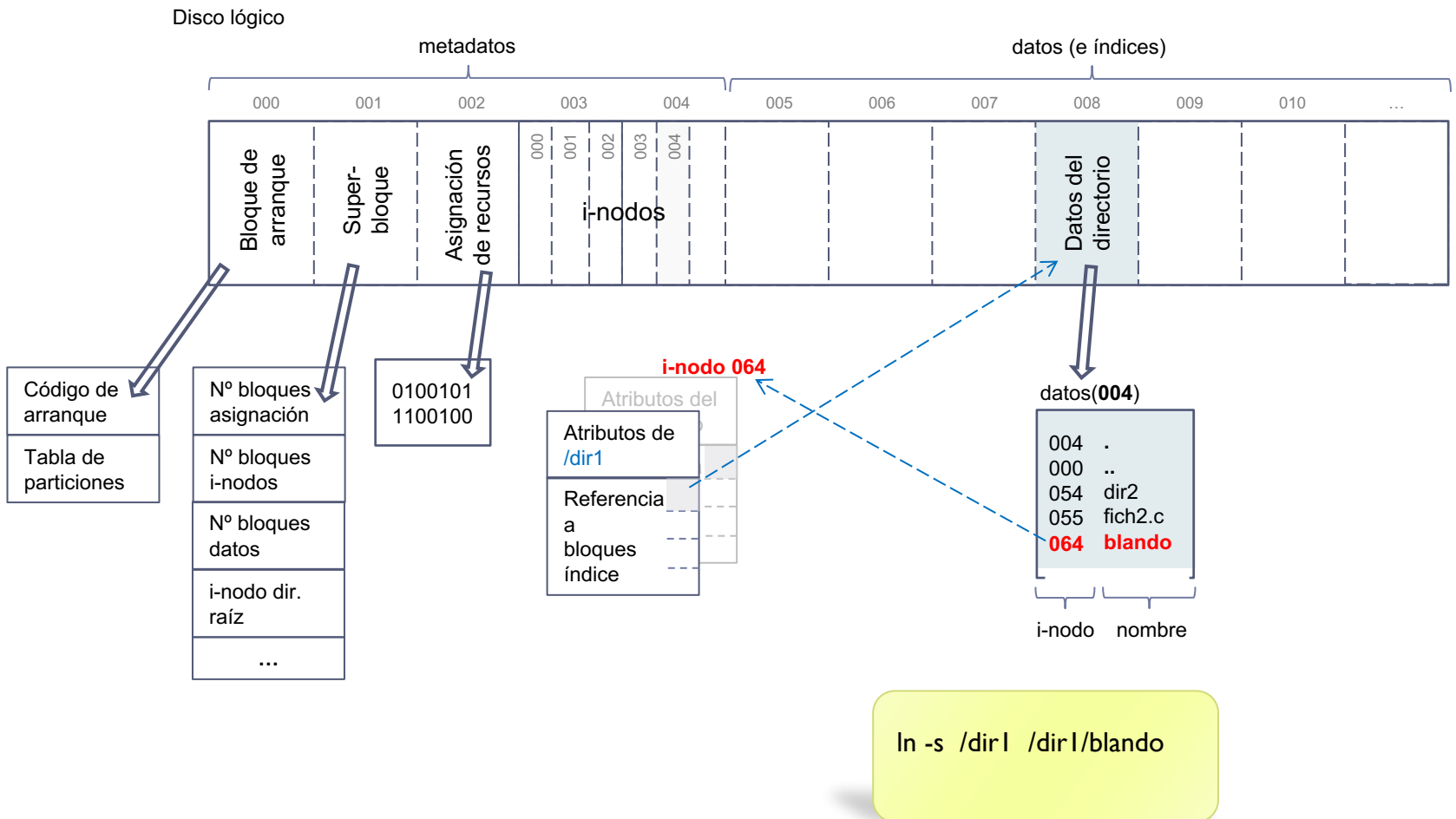
# Sistema de ficheros:

## representación tipo Unix: **enlace simbólico (o blando)**



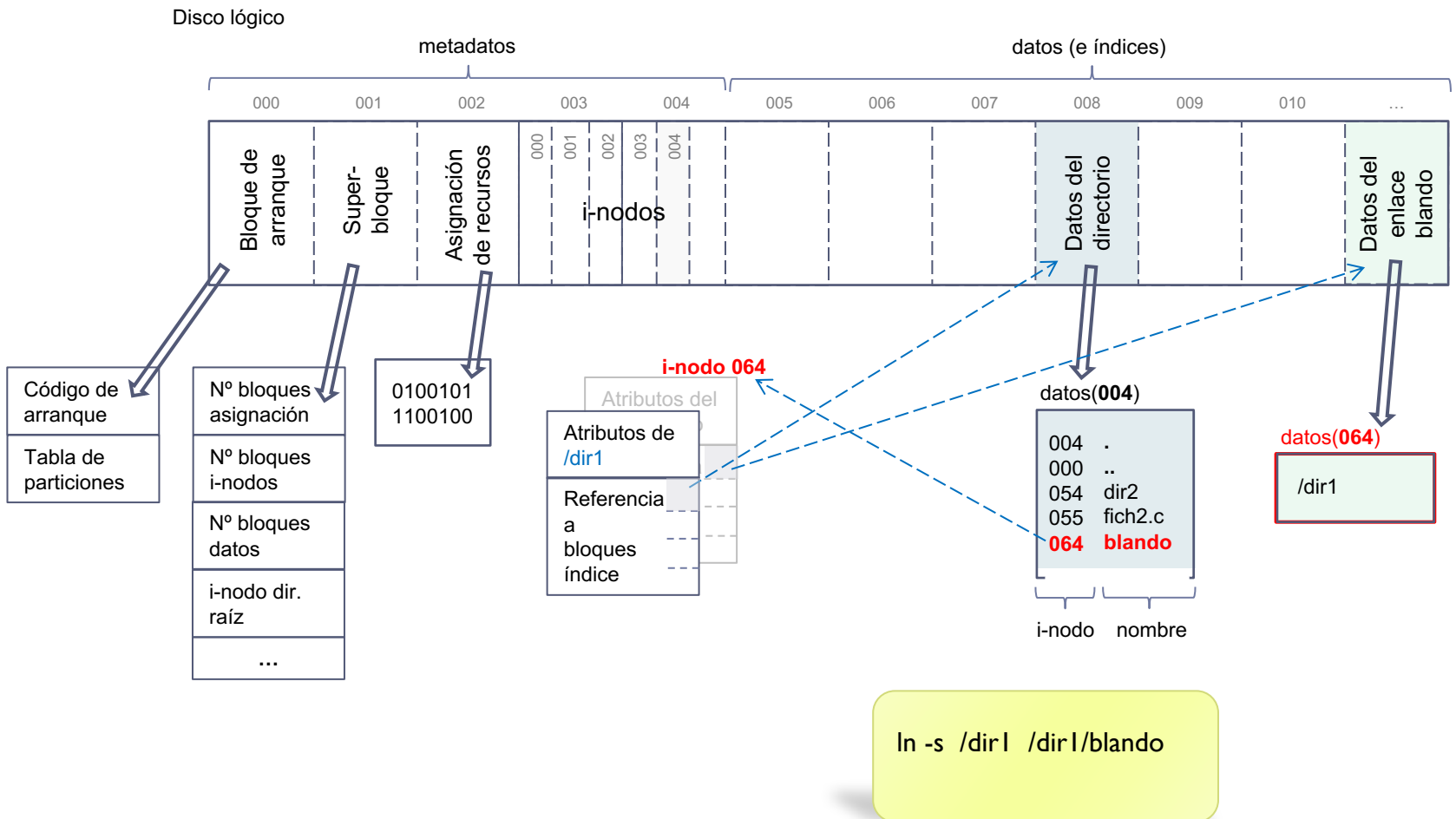
# Sistema de ficheros:

representación tipo Unix: **enlace simbólico (o blando)**

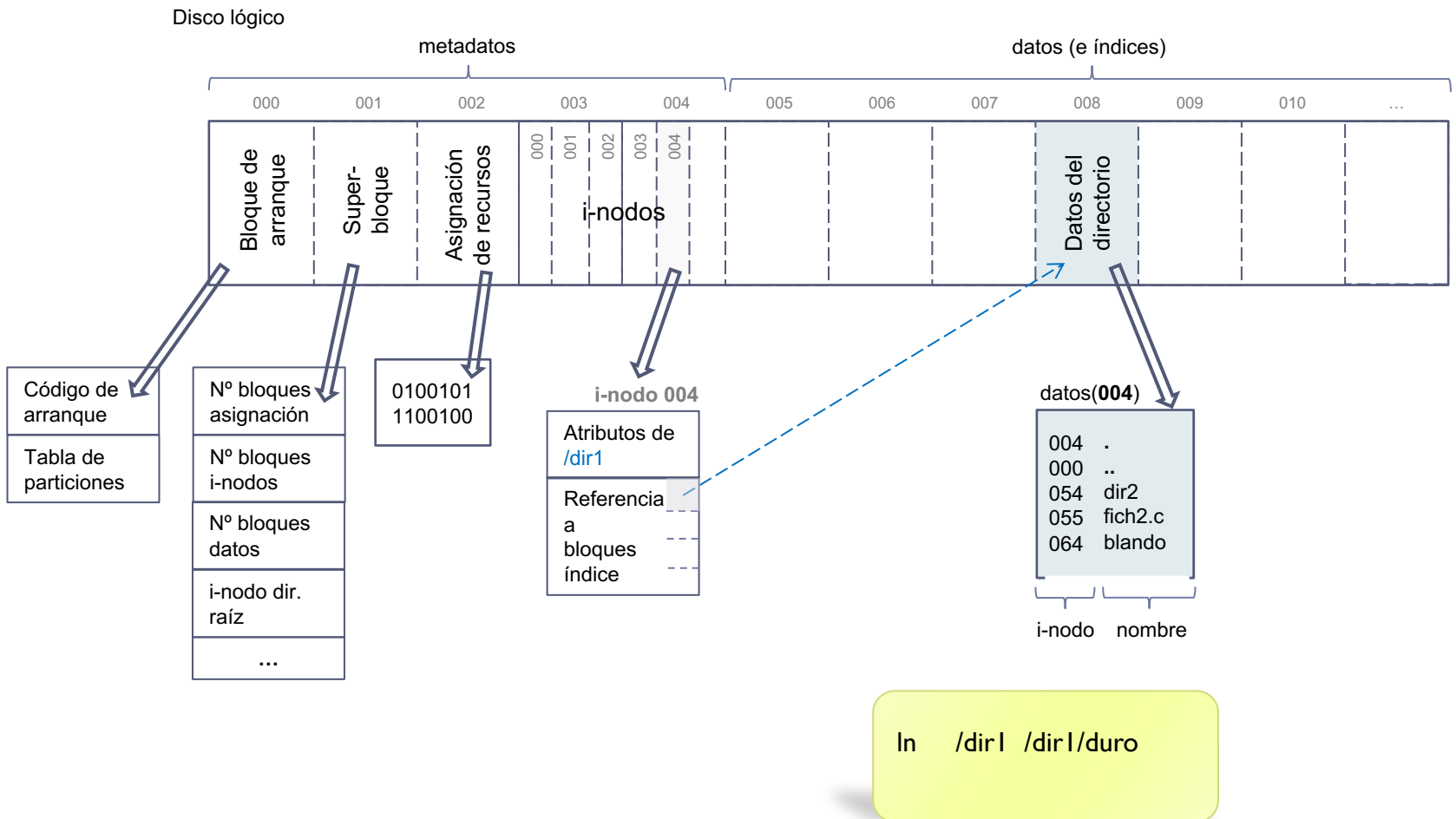


# Sistema de ficheros:

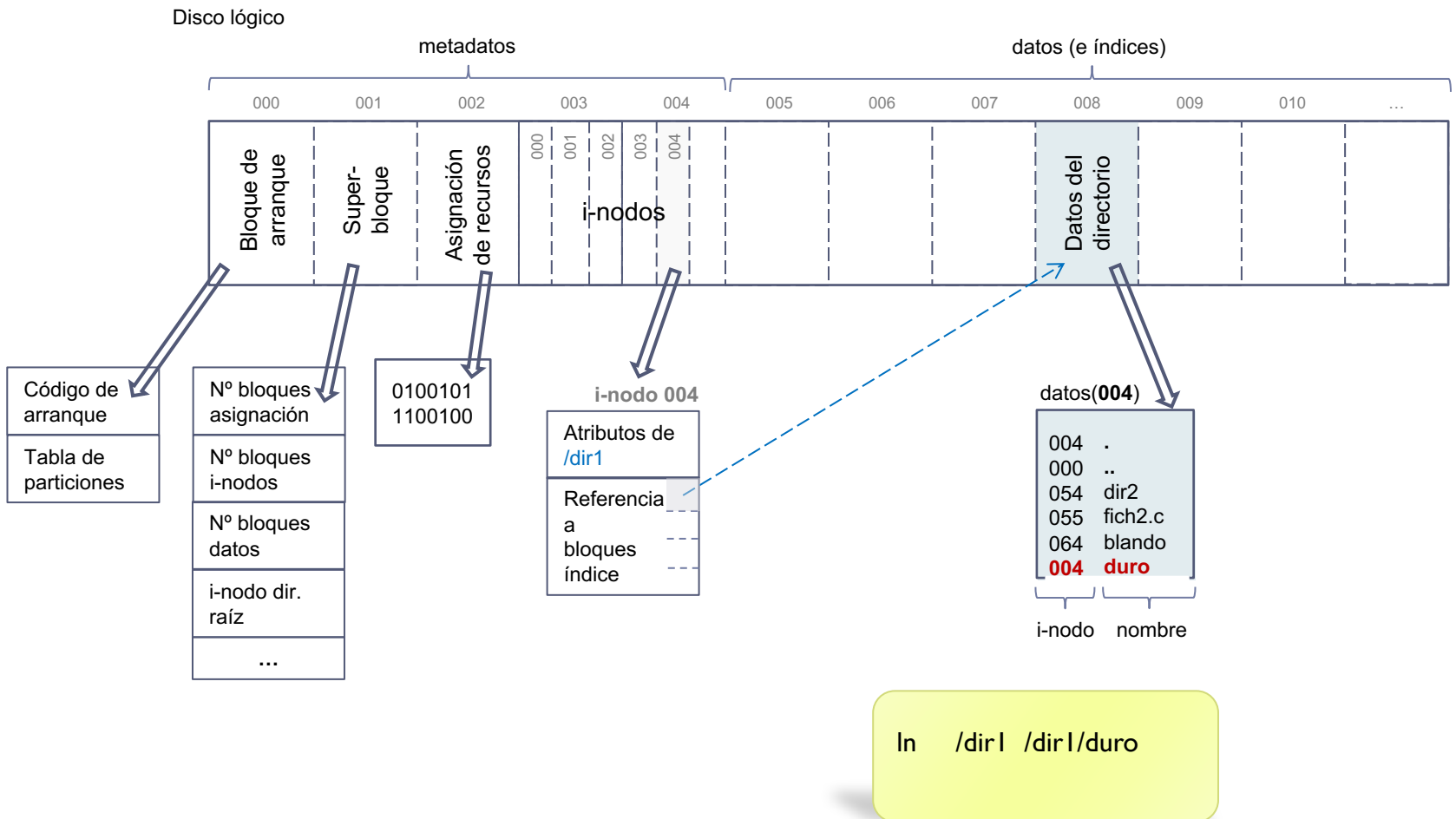
## representación tipo Unix: **enlace simbólico (o blando)**



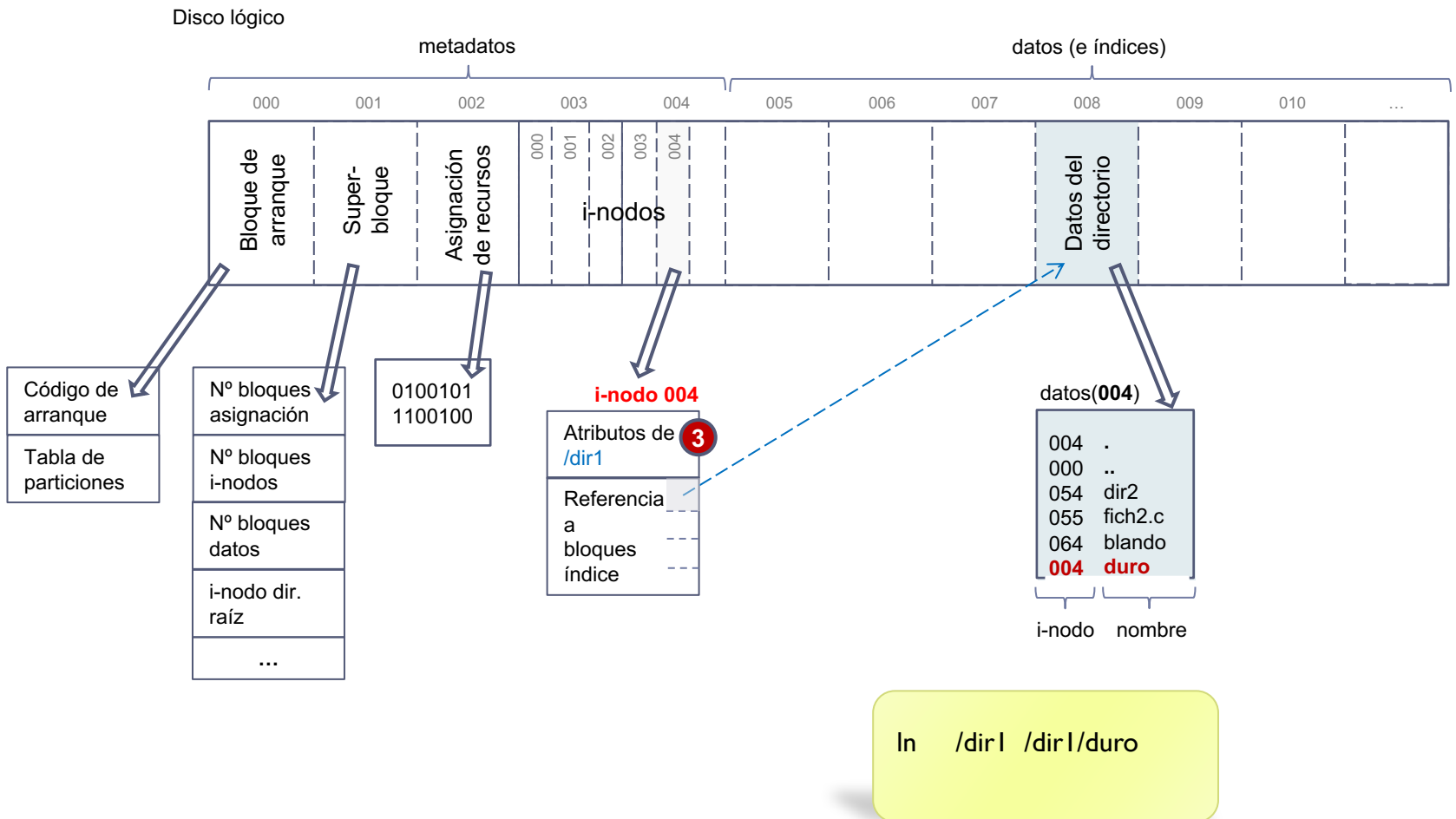
# Sistema de ficheros: representación tipo Unix: **enlace duro**



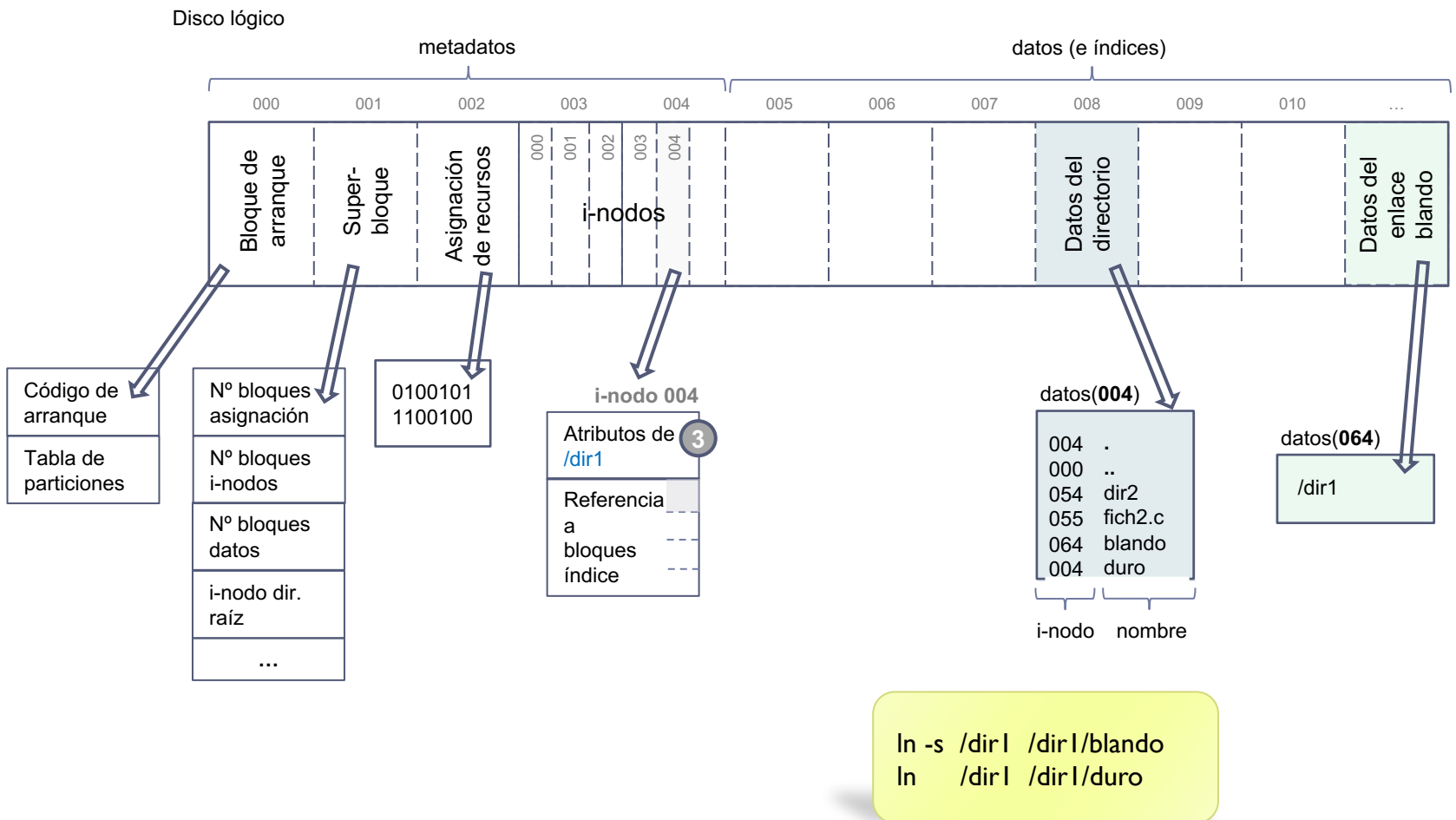
# Sistema de ficheros: representación tipo Unix: **enlace duro**



# Sistema de ficheros: representación tipo Unix: **enlace duro**

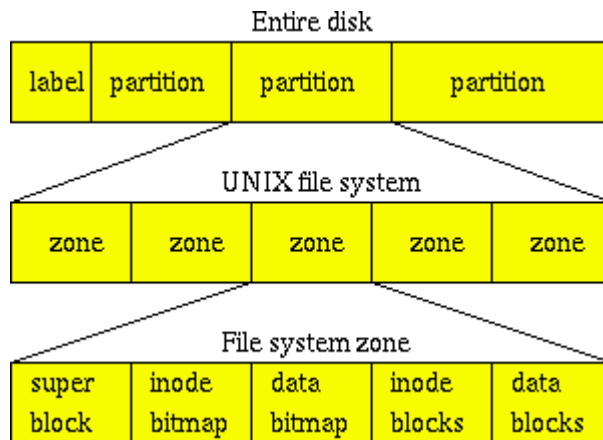


# Sistema de ficheros: enlace duro vs enlace simbólico



# Estructuras del sistema de ficheros

---



▶ UNIX/Linux

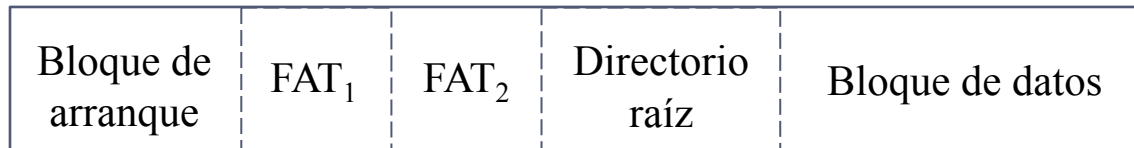
▶ FAT



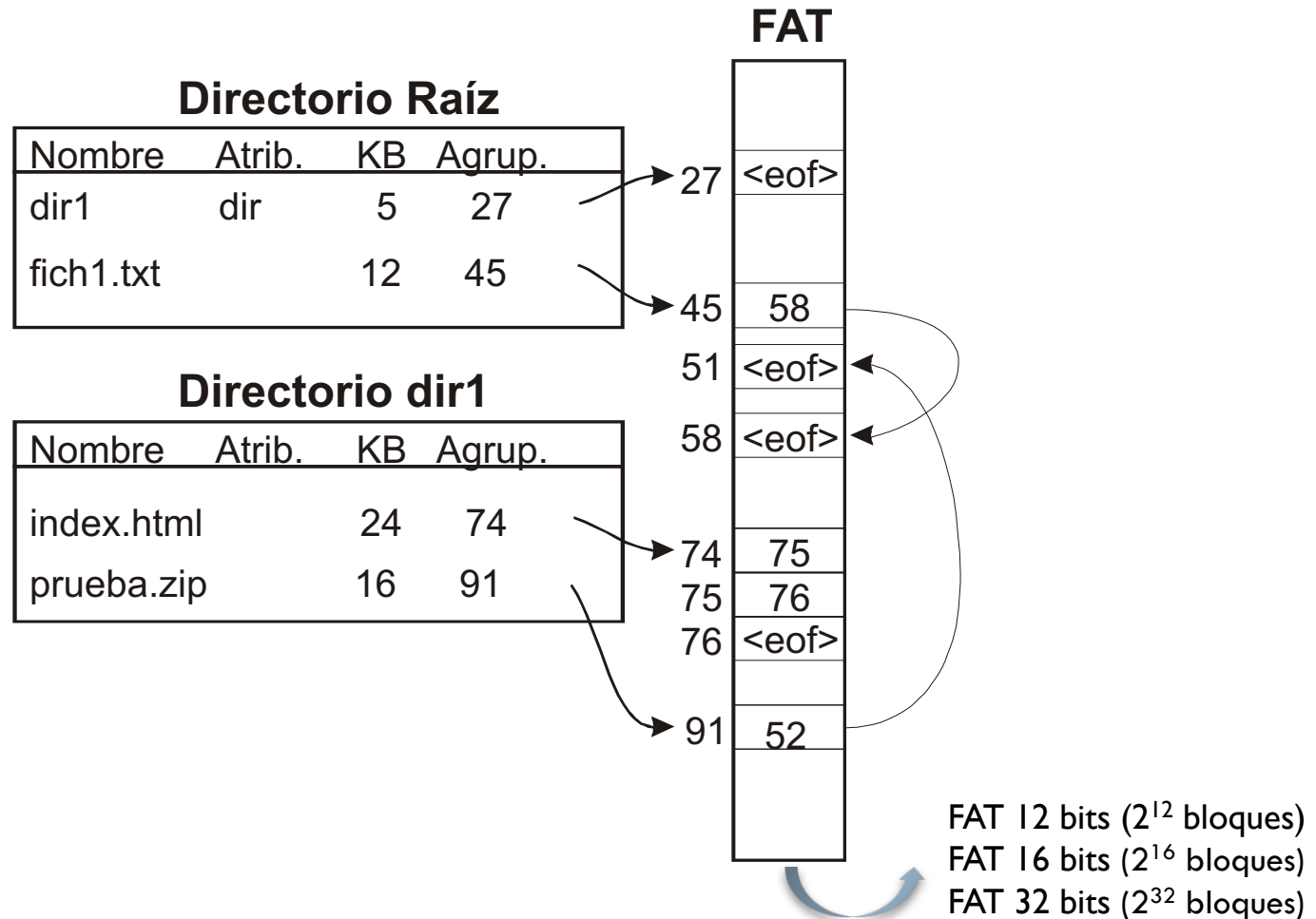
# Estructuras del sistema de ficheros:

## FAT

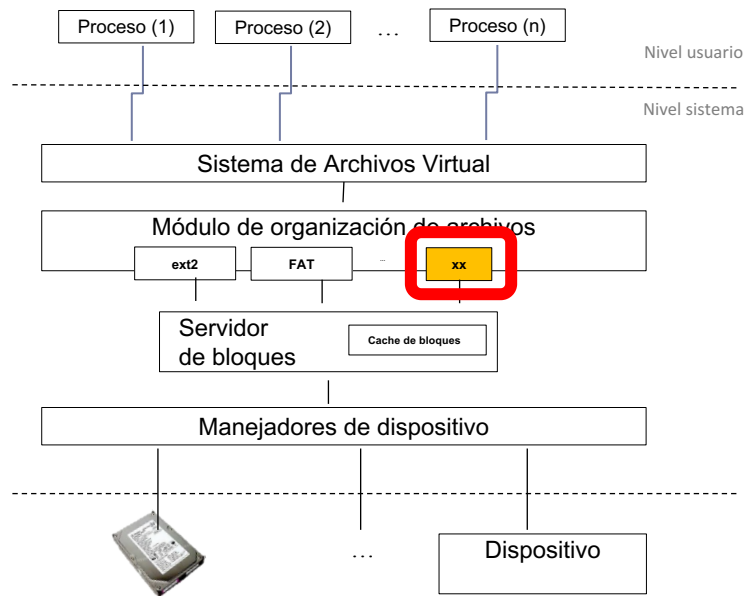
---



# Representación de ficheros y directorios: FAT



# Aspectos a tener en cuenta para añadir un sistema de ficheros...



- ▶ (0) Requisitos del sistema.
- ▶ (1) Estructuras en disco.
- ▶ **(2) Estructuras en memoria.**
- ▶ Caché de bloques.
- ▶ (3a) Funciones de gestión de estructuras disco/memoria.
- ▶ (3b) Funciones de llamadas al sistema.

## (2) Estructuras de datos en memoria...

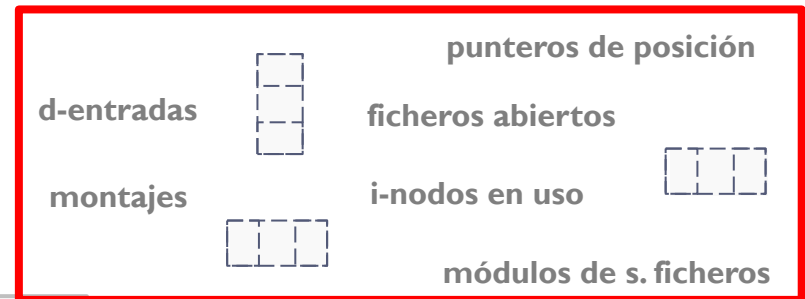
Llamadas al sistema de archivos

Descriptor	Uso de <i>namei</i>	Asig. i-n.	Atributos	E/S.	Sist. Arch.	Vista
open pipe	open chown unlink	creat	chown	read	mount	chdir
creat close	creat chmod mknod	mknod	chmod	write	umount	chroot
dup	chdir stat mount	link	stat	lseek		
	chroot link umount	unlink				

x

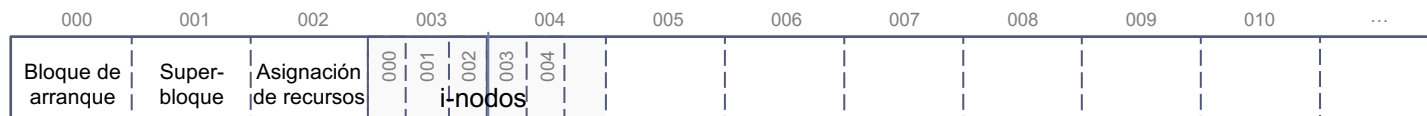
Algoritmos de bajo nivel del sistema de archivos

namei	ialloc	alloc	bmap
iget	ifree	free	



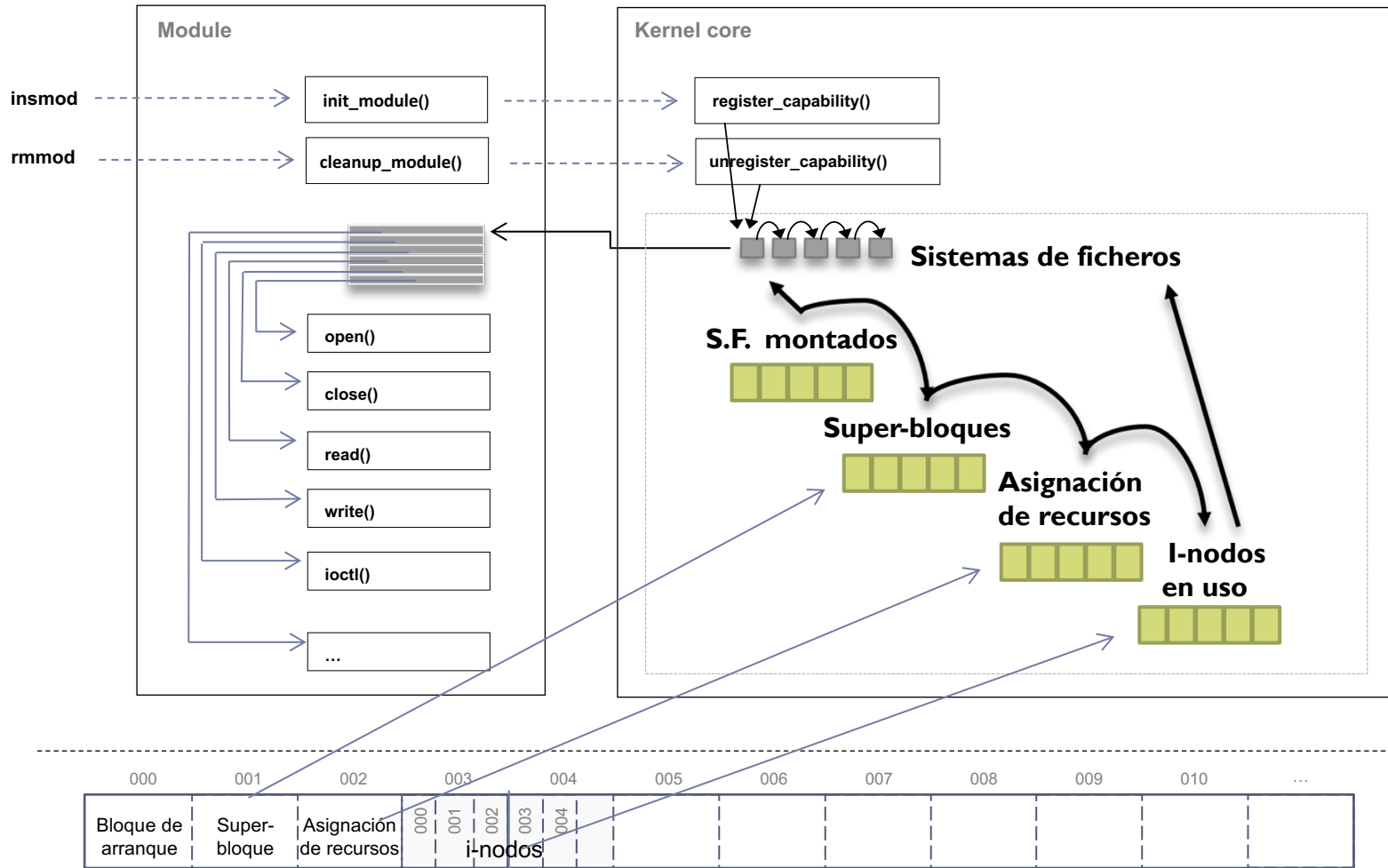
Algoritmos de gestión de bloques/caché

getblk	brelease	bread	breada	bwrite
--------	----------	-------	--------	--------



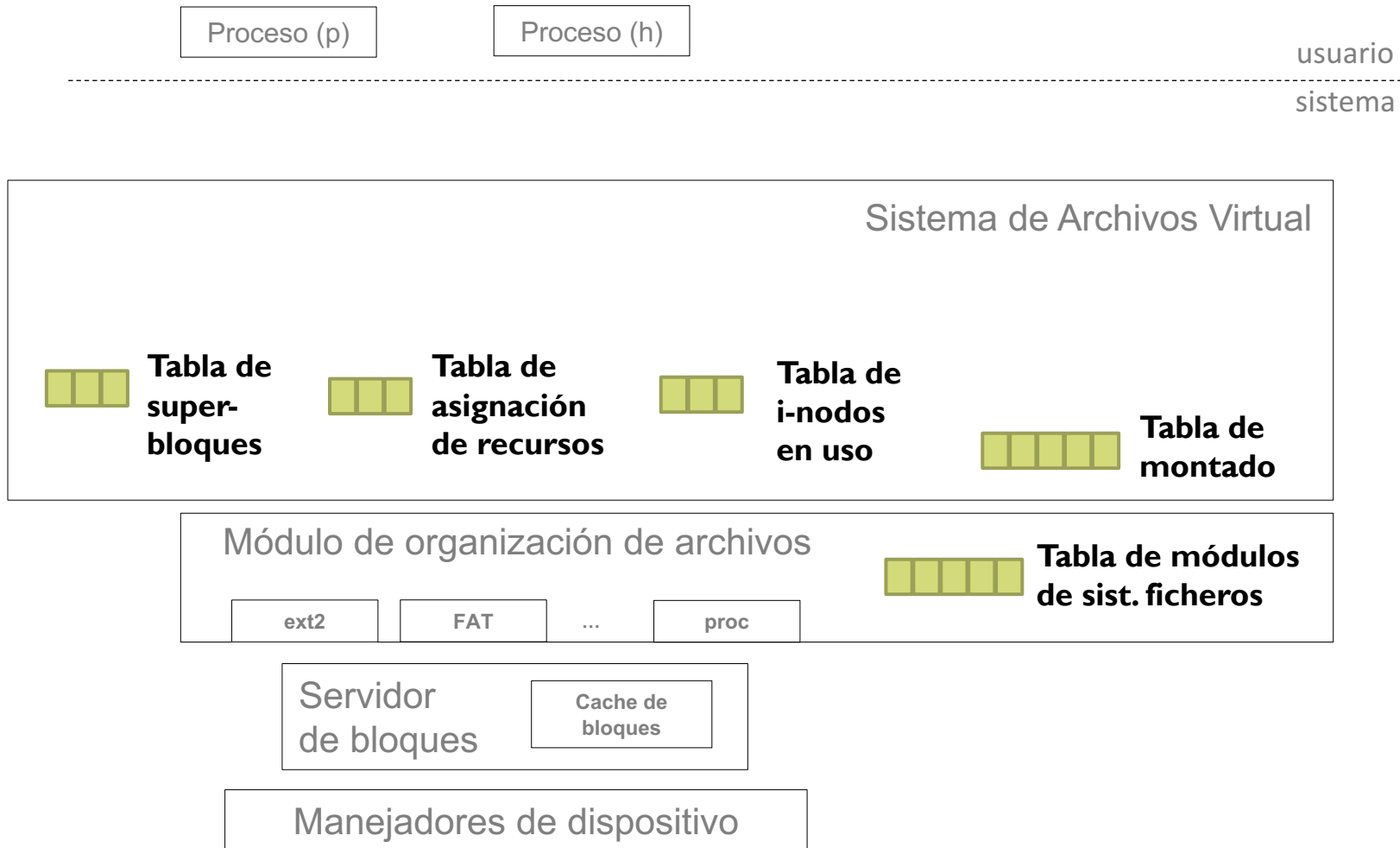
# Modelo de partida...

estructuras en memoria en relación al código



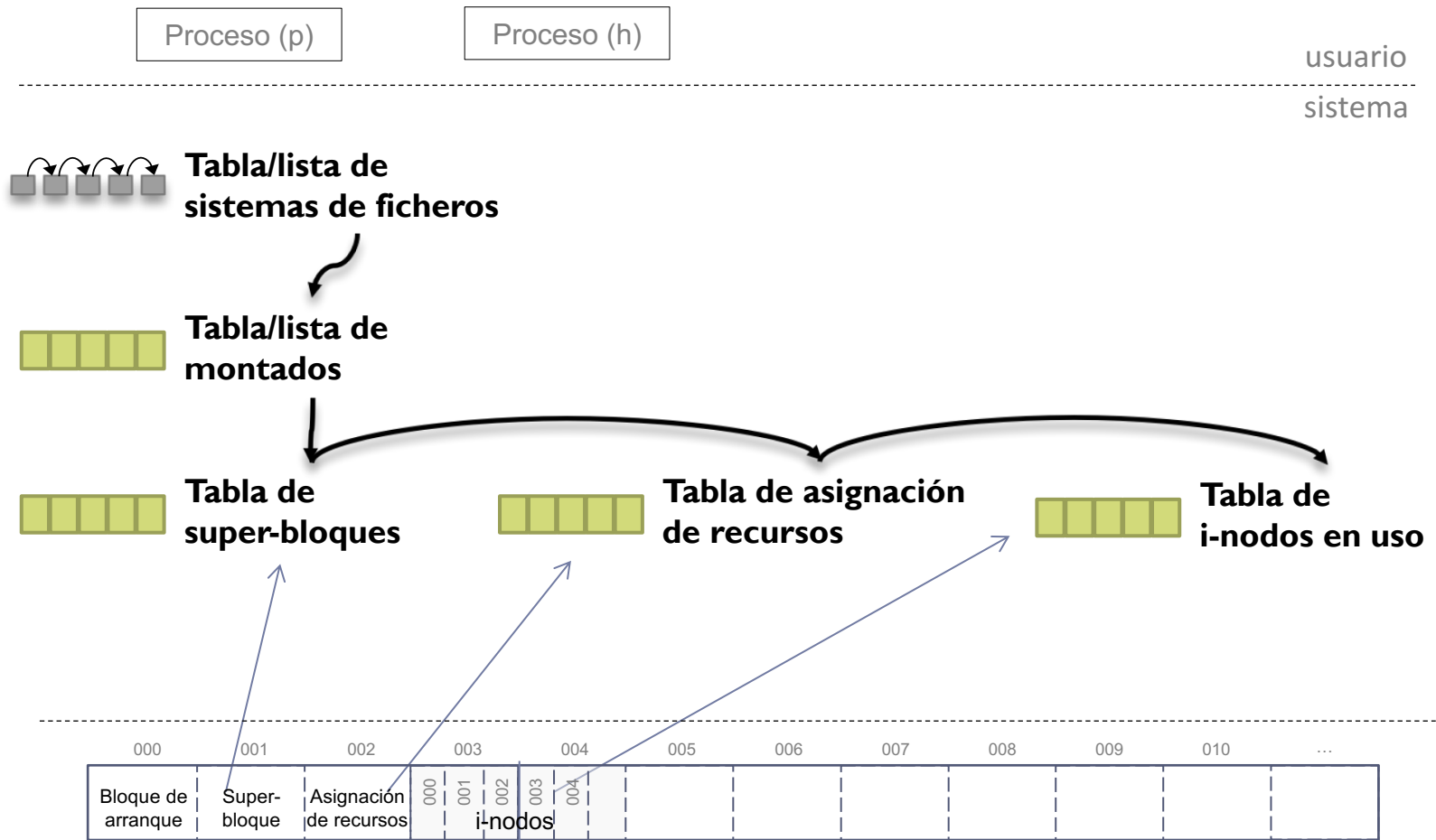
# Modelo de partida...

estructuras en memoria en relación a la arquitectura del sistema

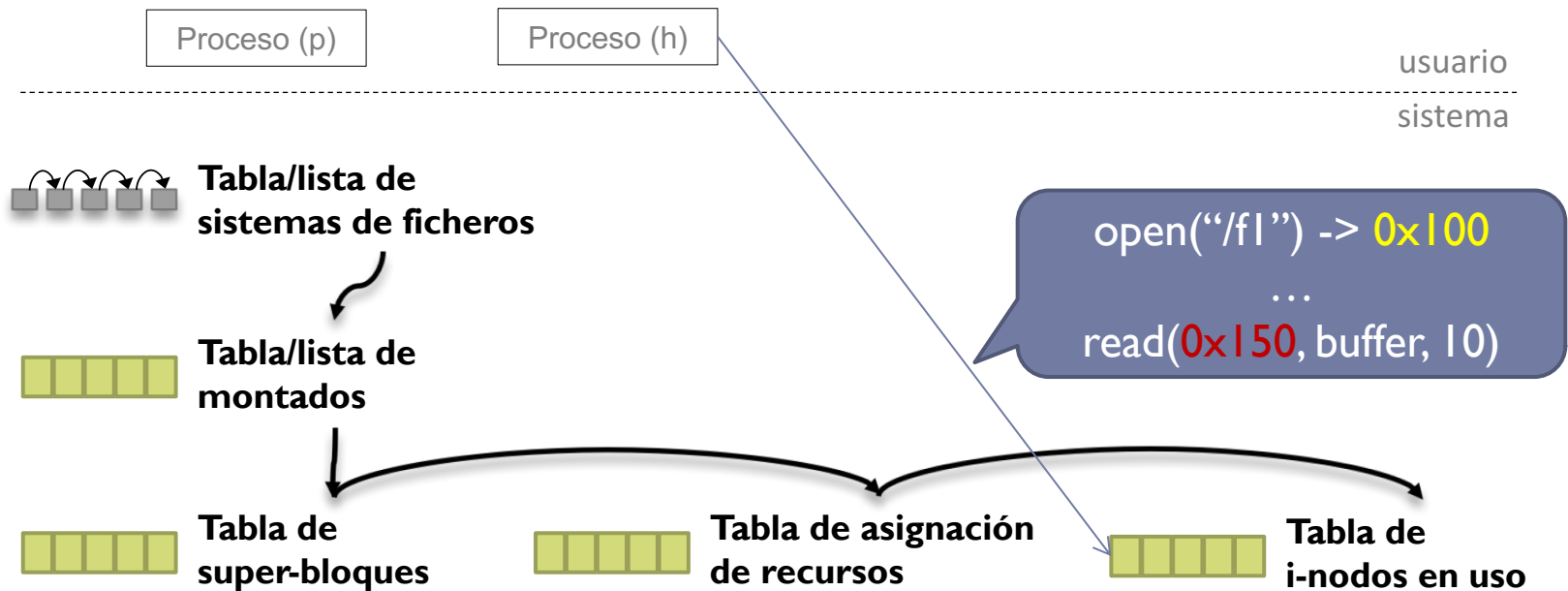


# Modelo de partida...

estructuras en memoria solo



# Modelo de partida...

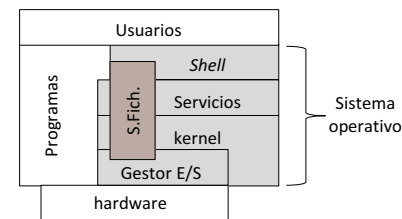


000	001	002	003	004	005	006	007	008	009	010	...
Bloque de arranque	Super-bloque	Asignación de recursos	000	001	002	003	004				
			i-nodos								



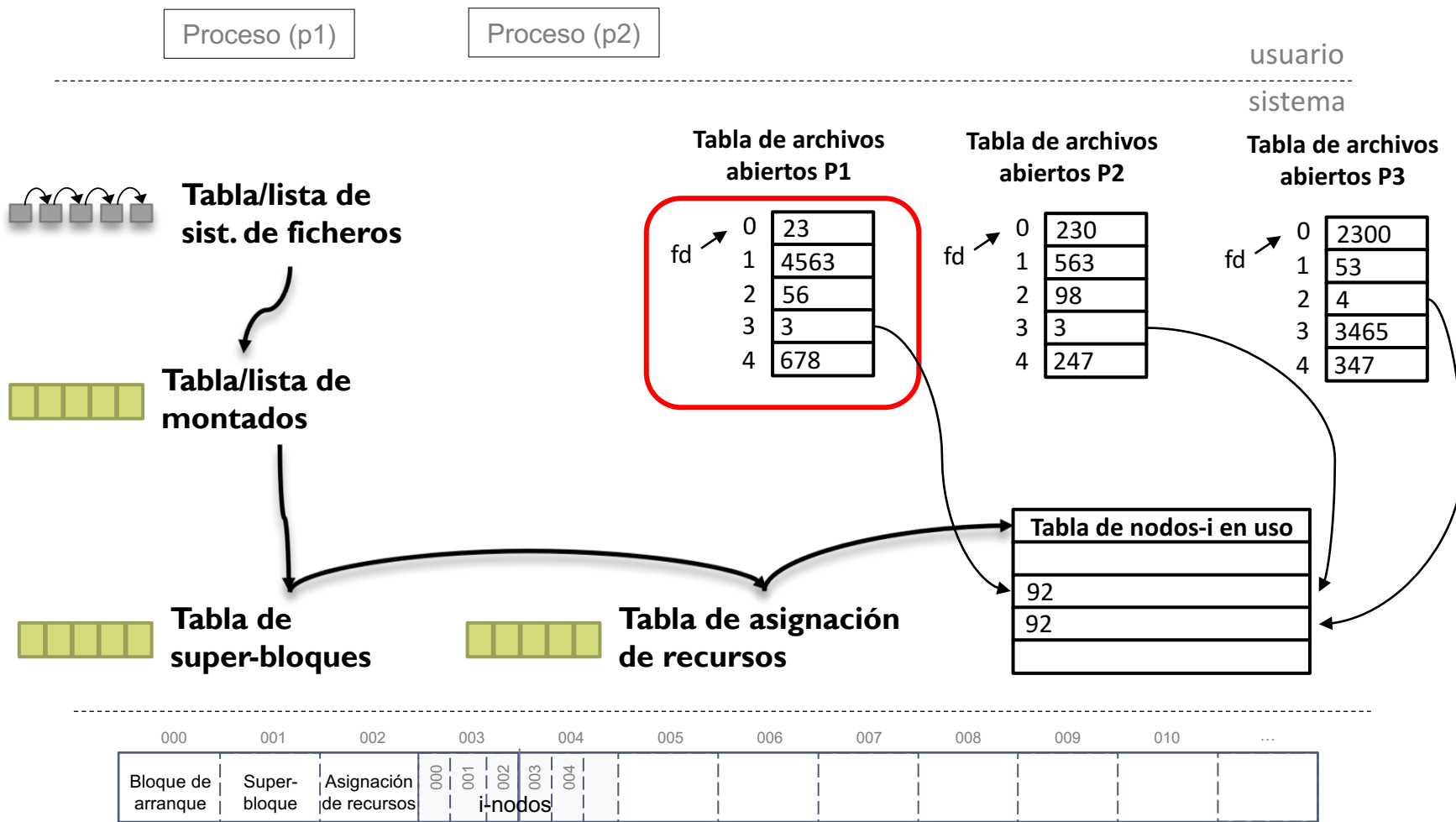
# (0) ~~Objetivos~~ requisitos principales

ej.: sistema de ficheros tipo Unix



- ▶ Lograr la **persistencia de los datos del usuario**, buscando **minimizar el impacto en el rendimiento y en el espacio para metadatos**.
- ▶ **Los procesos usarán una interfaz de trabajo segura**, sin acceso directo a la información usada en el kernel.
- ▶ **Compartir el puntero de posición de ficheros** entre procesos con relación de parentesco.
- ▶ Poder tener **una sesión de trabajo con un fichero/directorio** para actualizar la información que contiene.
- ▶ Poder tener **una sesión de trabajo con varios directorios** para poder recorrer sus entradas.
- ▶ **Llevar la pista de los sistemas de ficheros presentes en el kernel**, y llevar la pista de **los puntos donde están siendo usados**.

# Estructuras principales de gestión tabla de descriptores (ficheros abiertos)



# Estructuras principales de gestión tabla de descriptores (ficheros abiertos)

Tabla de archivos  
abiertos P1

fd → 0	23
1	4563
2	56
3	3
4	678

Tabla de archivos  
abiertos P2

fd → 0	230
1	563
2	98
3	3
4	247

Tabla de archivos  
abiertos P3

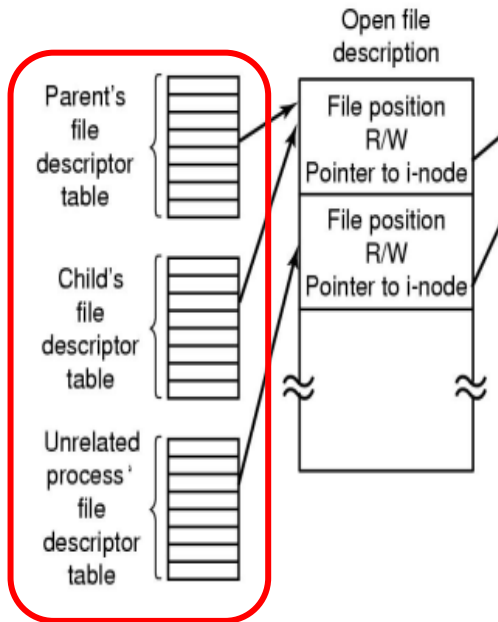
fd → 0	2300
1	53
2	4
3	3465
4	347

Tabla de nodos-i	
	92
	92



No se accede directamente al recurso (dinero)  
sino a través de un descriptor (# tarjeta)

# Estructuras principales de gestión tabla de descriptores (ficheros abiertos)



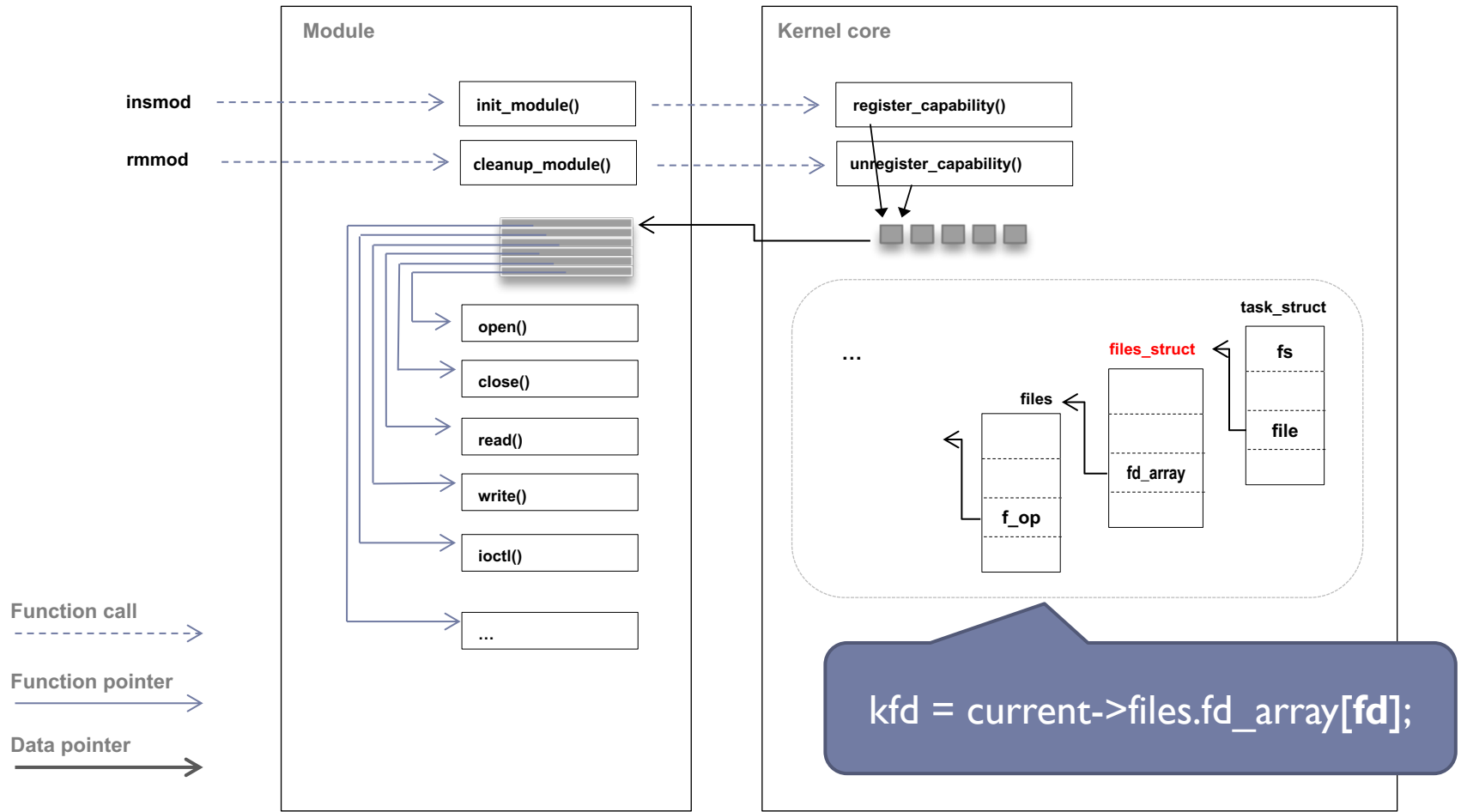
- ▶ Cada proceso tiene una tabla de descriptores donde guarda la referencia de cada uno de los ficheros que ha abierto:
  - ▶ El descriptor de archivo  $fd$  indica el lugar de tabla.
  - ▶ Se asigna a  $fd$  la primera posición libre de la  $tdaa$ .
  - ▶ El tamaño de la tabla determina el máximo número de archivos abiertos que cada proceso puede tener abierto a la vez en un instante dado.
  - ▶ En los sistemas UNIX cada proceso tiene tres descriptores de archivos abiertos por defecto: entrada estándar (0), salida estándar (1) y salida de error (2).
- ▶ Al clonar un proceso se copia todos los valores:
  - ▶ Los procesos clonados “ven” los ficheros/pipes que el padre había abierto antes de clonar.

# Estructuras principales de gestión tabla de descriptores (ficheros abiertos): Linux



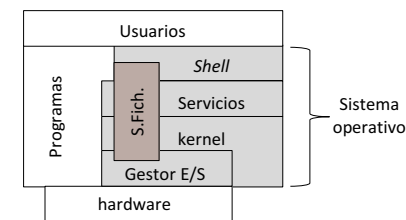
```
struct fs_struct {
    atomic_t    count;           /* structure's usage count */
    spinlock_t  file_lock;      /* lock protecting this structure */
    int         max_fds;        /* maximum number of file objects */
    int         max_fdset;      /* maximum number of file descriptors */
    int         next_fd;        /* next file descriptor number */
    struct file **fd;           /* array of all file objects */
    fd_set      *close_on_exec; /* file descriptors to close on exec() */
    fd_set      *open_fds;      /* pointer to open file descriptors */
    fd_set      close_on_exec_init; /* initial files to close on exec() */
    fd_set      open_fds_init;  /* initial set of file descriptors */
    struct file *fd_array[NR_OPEN_DEFAULT]; /* array of file objects */
};
```

# Estructuras principales de gestión tabla de descriptores (ficheros abiertos): Linux



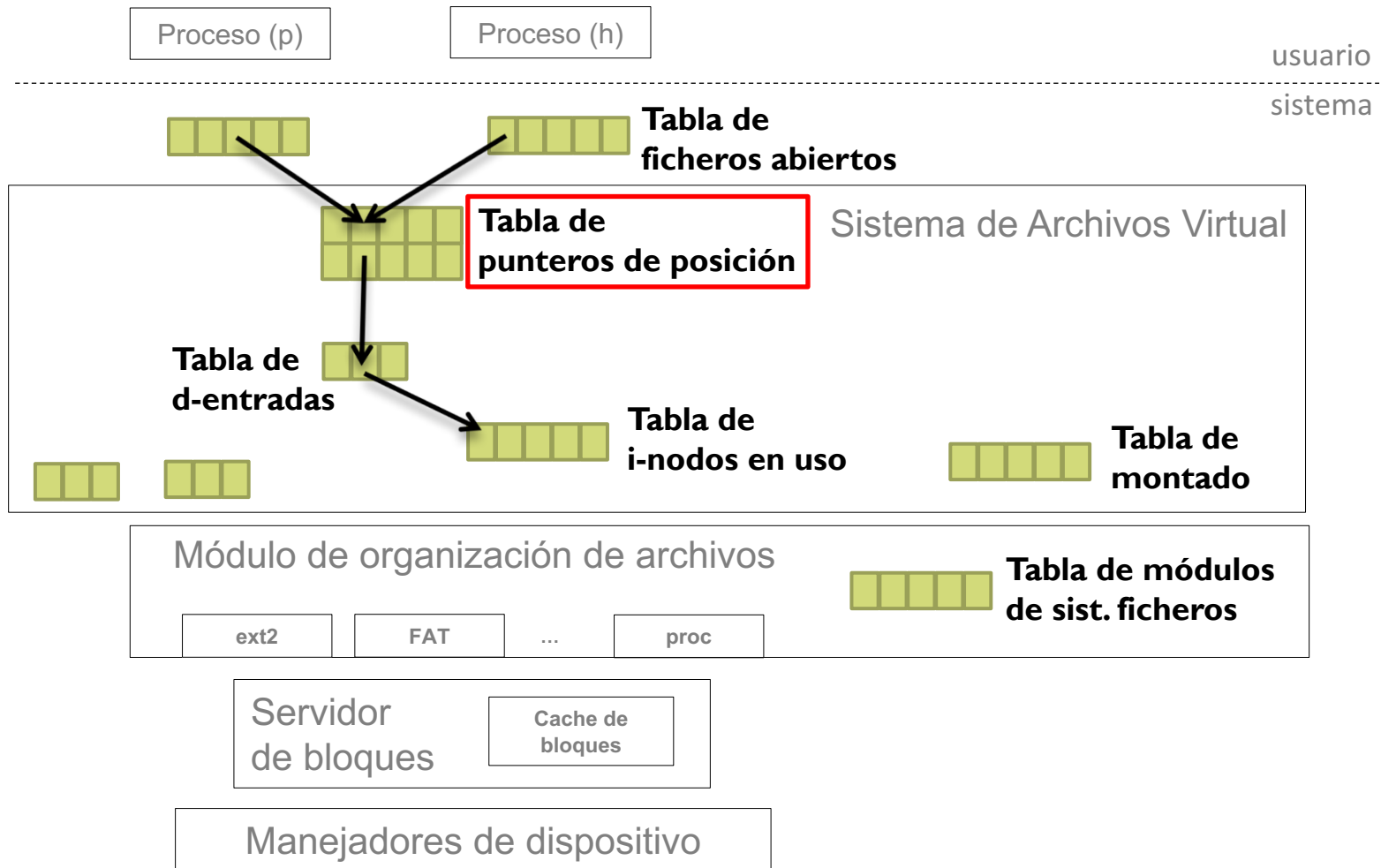
# (0) ~~Objetivos~~ requisitos principales

ej.: sistema de ficheros tipo Unix



- ▶ Lograr la **persistencia de los datos del usuario**, buscando **minimizar el impacto en el rendimiento y en el espacio para metadatos**.
- ▶ Los procesos usarán una **interfaz de trabajo segura**, sin acceso directo a la información usada en el kernel.
- ▶ **Compartir el puntero de posición de ficheros** entre procesos con relación de parentesco.
- ▶ Poder tener **una sesión de trabajo con un fichero/directorio** para actualizar la información que contiene.
- ▶ Poder tener **una sesión de trabajo con varios directorios** para poder recorrer sus entradas.
- ▶ **Llevar la pista de los sistemas de ficheros presentes** en el kernel, **y llevar la pista de los puntos donde están siendo usados**.

# Estructuras principales de gestión





# Estructuras principales de gestión

## tabla punteros de posición

Tabla de archivos abiertos P1

fd → 0	23
1	4563
2	56
3	3
4	678

Tabla de archivos abiertos P2

fd → 0	230
1	563
2	98
3	3
4	247

Tabla de archivos abiertos P3

fd → 0	2300
1	53
2	4
3	3465
4	347

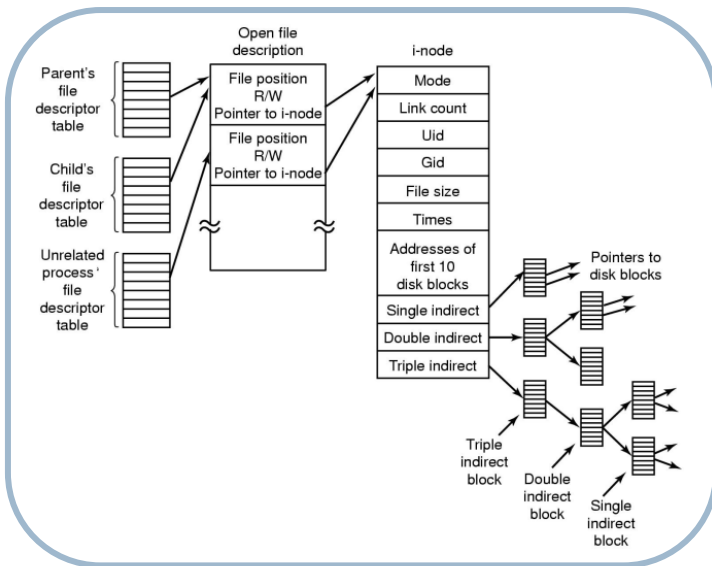


Tabla de nodos-i


Nodo-i	Posición
92	345
92	5678

Tabla intermedia de nodos-i y posiciones

# Estructuras principales de gestión

## tabla punteros de posición

- ▶ Tabla FILP (FILE Pointer)
- ▶ Entre la tabla de descriptores y (normalmente) la tabla de i-nodos.
- ▶ Guarda (principalmente) el puntero de posición del archivo.

Tabla de archivos abiertos P1

fd → 0	23
1	4563
2	56
3	3
4	678

Tabla de archivos abiertos P2

fd → 0	230
1	563
2	98
3	3
4	247

Tabla de archivos abiertos P3

fd → 0	2300
1	53
2	4
3	3465
4	347

Tabla de nodos-i


Nodo-i	Posición
92	345
92	5678

Tabla intermedia de nodos-i y posiciones

# Estructuras principales de gestión

## tabla punteros de posición

Tabla de archivos abiertos P1

fd → 0	23
1	4563
2	56
3	3
4	678

Tabla de archivos abiertos P2

fd → 0	230
1	563
2	98
3	3
4	247

Tabla de archivos abiertos P3

fd → 0	2300
1	53
2	4
3	3465
4	347

Tabla de nodos-i


Nodo-i	Posición
92	345
92	5678

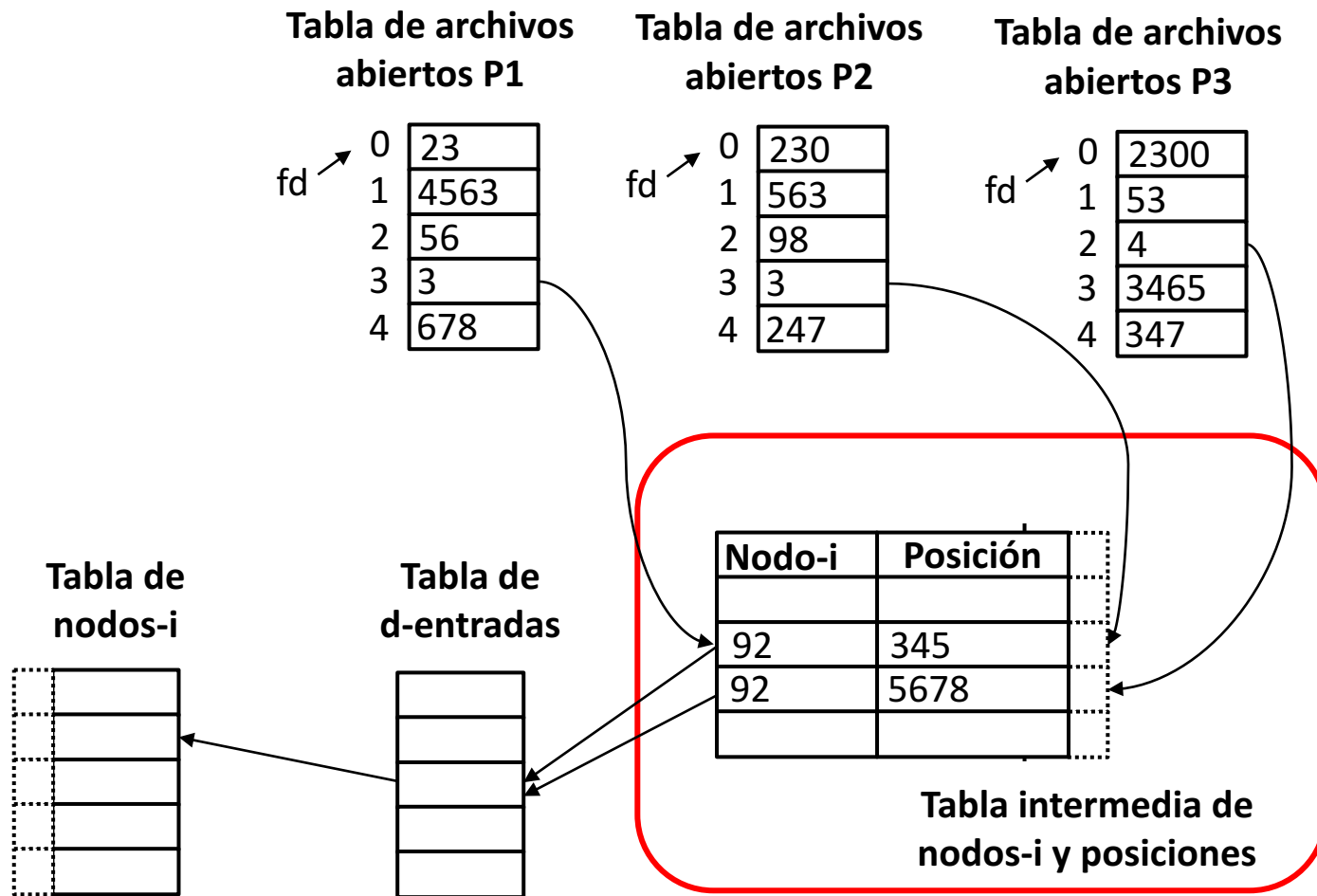
Tabla intermedia de nodos-i y posiciones

- ▶ La tabla de nodos-i (o i-nodos) suele tener todos los datos y referencias a funciones necesarias para trabajar con ficheros y directorios:

- ▶ Transferencia, metadatos, etc.

# Estructuras principales de gestión

tabla ficheros: Linux



# Estructuras principales de gestión

## tabla ficheros: Linux



- ▶ Hay una tabla intermedia más para llegar a la de i-nodos (nodos-i)
- ▶ Las operaciones más frecuentes se mueven de los i-nodos a la tabla intermedia

Tabla de archivos abiertos P1

fd → 0	23
1	4563
2	56
3	3
4	678

Tabla de archivos abiertos P2

fd → 0	230
1	563
2	98
3	3
4	247

Tabla de archivos abiertos P3

fd → 0	2300
1	53
2	4
3	3465
4	347

Tabla de nodos-i


Tabla de d-entradas


Nodo-i	Posición
92	345
92	5678

Tabla intermedia de nodos-i y posiciones

# Estructuras principales de gestión tabla ficheros: Linux

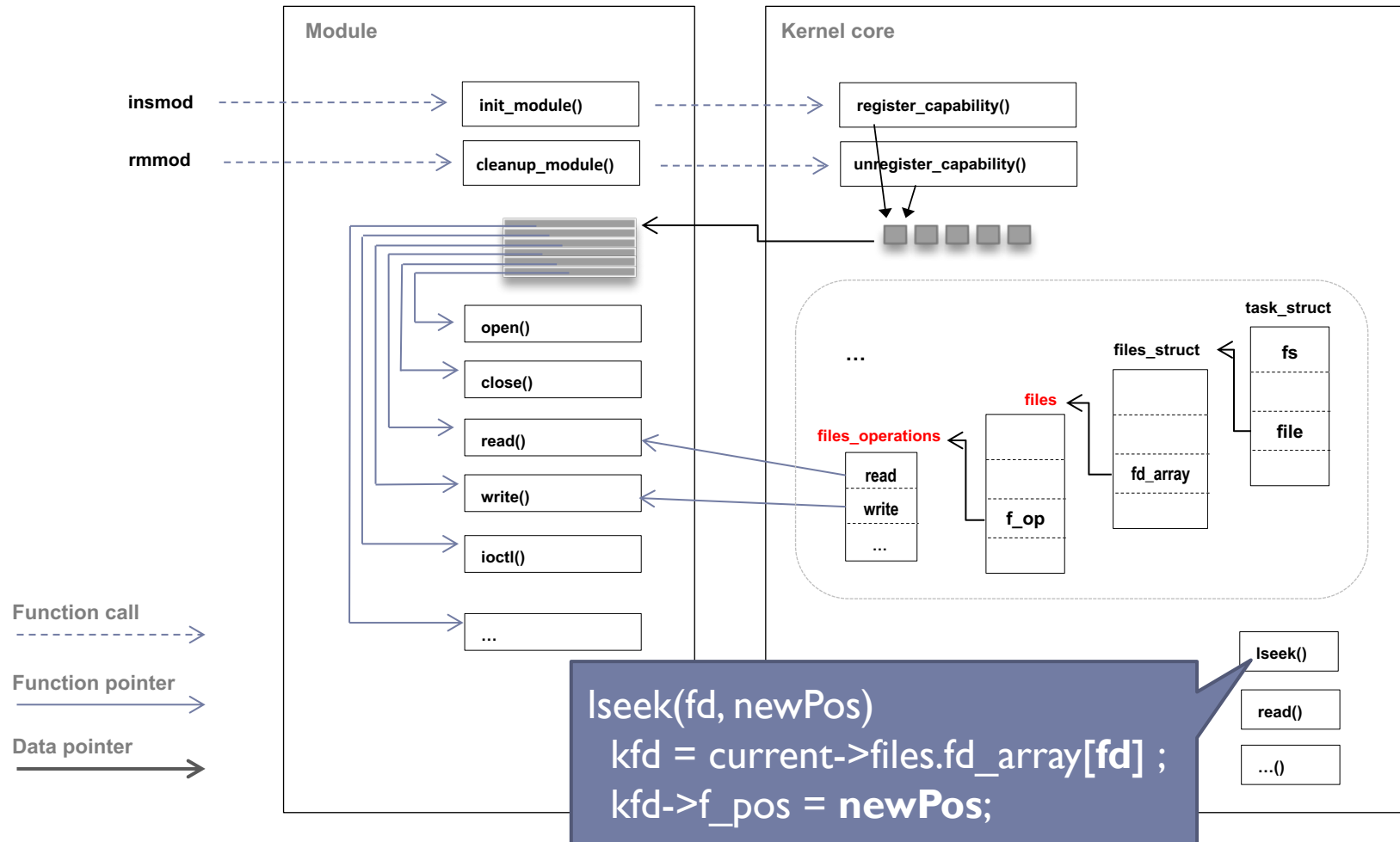


```
struct file {
    struct dentry      *f_dentry;
    struct vfsmount    *f_vfsmnt;
    struct file_operations *f_op;
    mode_t             f_mode;
    loff_t             f_pos;
    struct fown_struct f_owner;
    unsigned int       f_uid, f_gid;
    unsigned long      f_version;
    ...
};
```

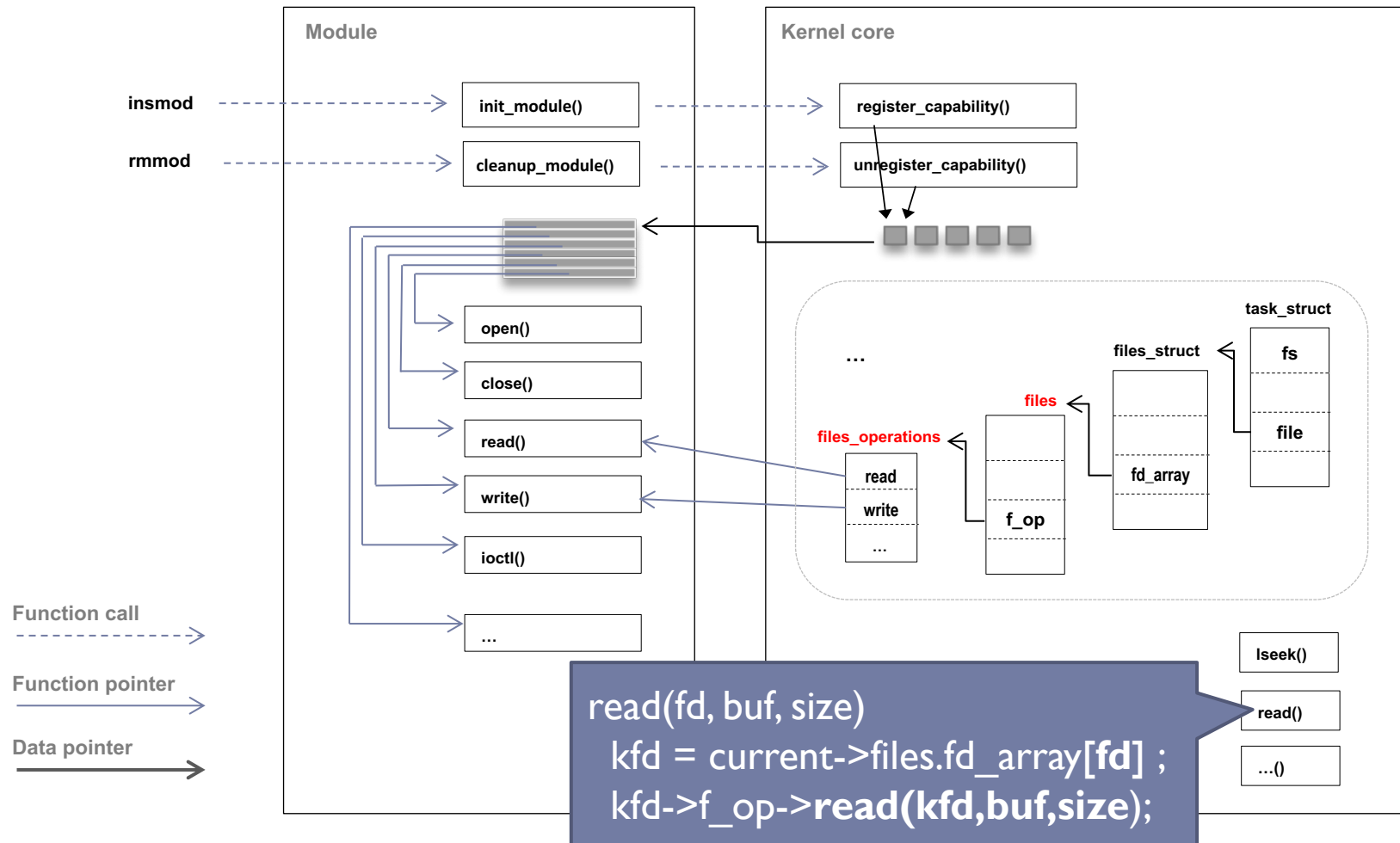
→

```
struct file_operations {
    int      (*open)   (struct inode *, struct file *);
    ssize_t (*read)   (struct file *, char *, size_t, loff_t *);
    ssize_t (*write)  (struct file *, const char *, size_t, loff_t *);
    loff_t  (*llseek) (struct file *, loff_t, int);
    int     (*ioctl)  (struct inode *, struct file *,
                     unsigned int, ulong);
    int     (*readdir) (struct file *, void *, filldir_t);
    int     (*mmap)   (struct file *, struct vm_area_struct *);
    ...
};
```

# Estructuras principales de gestión tabla ficheros: Linux



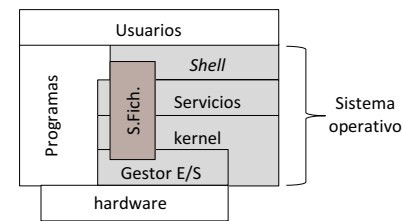
# Estructuras principales de gestión tabla ficheros: Linux





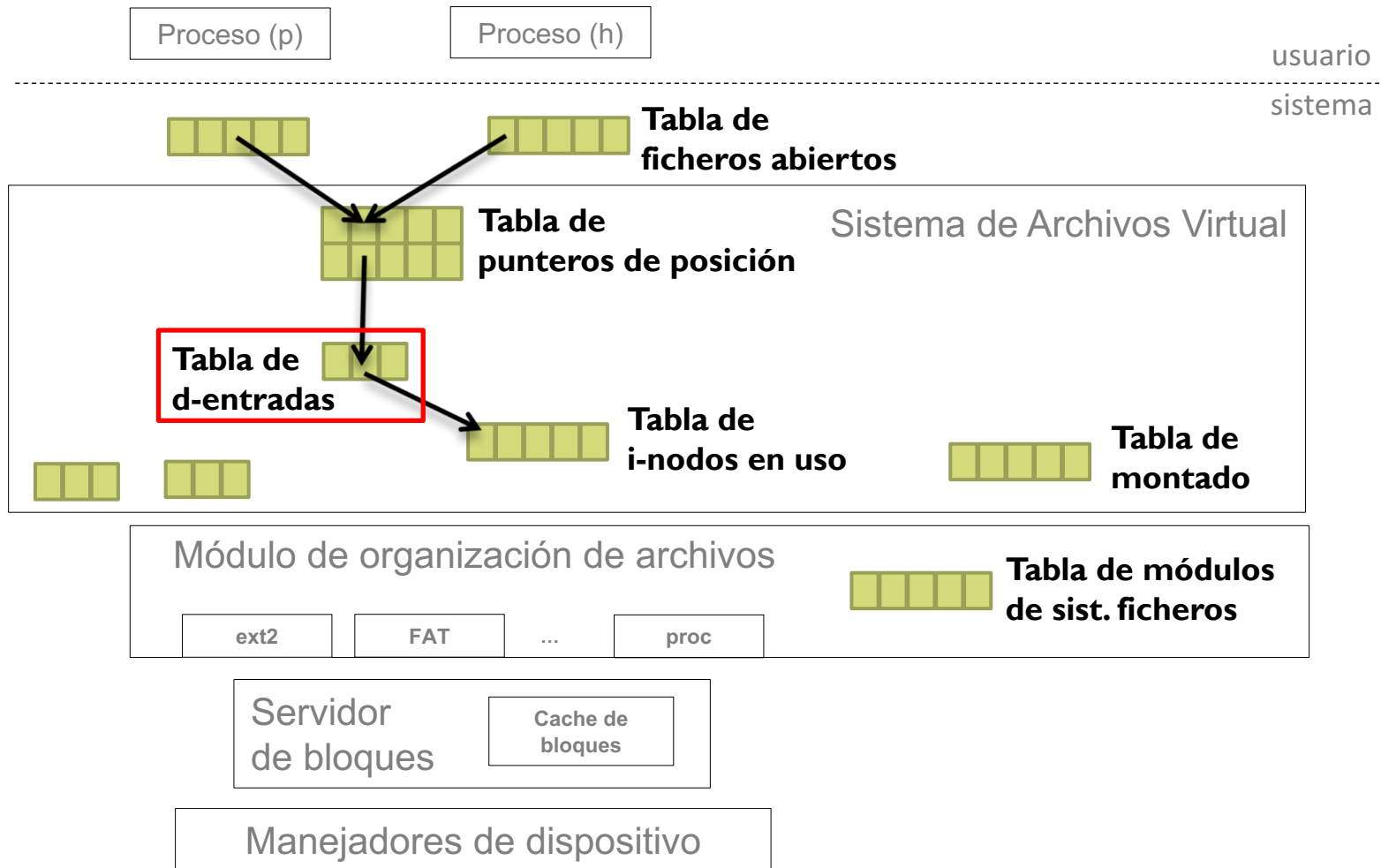
# (0) ~~Objetivos~~ requisitos principales

ej.: sistema de ficheros tipo Unix



- ▶ Lograr la **persistencia de los datos del usuario**, buscando **minimizar el impacto en el rendimiento y en el espacio para metadatos**.
- ▶ Los procesos usarán una **interfaz de trabajo segura**, sin acceso directo a la información usada en el kernel.
- ▶ **Compartir el puntero de posición de ficheros** entre procesos con relación de parentesco.
- ▶ Poder tener **una sesión de trabajo con un fichero/directorio** para actualizar la información que contiene.
- ▶ Poder tener **una sesión de trabajo con varios directorios** para poder recorrer sus entradas.
- ▶ **Llevar la pista de los sistemas de ficheros presentes en el kernel, y llevar la pista de los puntos donde están siendo usados.**

# Estructuras principales de gestión



# Estructuras principales de gestión

## tabla de d-entradas (entradas de directorio)

---

- ▶ Usada como la caché de entradas de directorios.
  - ▶ Principalmente relaciona el nombre de una entrada (fichero o directorio) con su i-nodo.
    - ▶ Se llama dentry negativa si no tiene asociado un i-nodo (solo es un nombre).
  - ▶ Pero también relaciona el nombre de una entrada (fichero o directorio) con el nombre del directorio padre, con el superbloque, funciones de gestión asociadas, etc.

# Estructuras principales de gestión tabla de d-entradas (entradas de directorio): Linux



```
struct dentry {  
    struct inode      *d_inode;  
    struct dentry     *d_parent;  
    struct qstr       d_name;  
    struct dentry_operations *d_op;  
    struct super_block *d_sb;  
    struct list_head  d_subdirs;  
    ...  
}  
  
struct dentry_operations {  
    int (*d_revalidate) (struct dentry *, int);  
    int (*d_hash)      (struct dentry *,  
                        struct qstr *);  
    int (*d_compare)   (struct dentry *,  
                        struct qstr *,  
                        struct qstr *);  
    int (*d_delete)    (struct dentry *);  
    void (*d_release)  (struct dentry *);  
    void (*d_iput)     (struct dentry *,  
                        struct inode *);  
}
```

# Estructuras principales de gestión tabla de d-entradas (entradas de directorio)

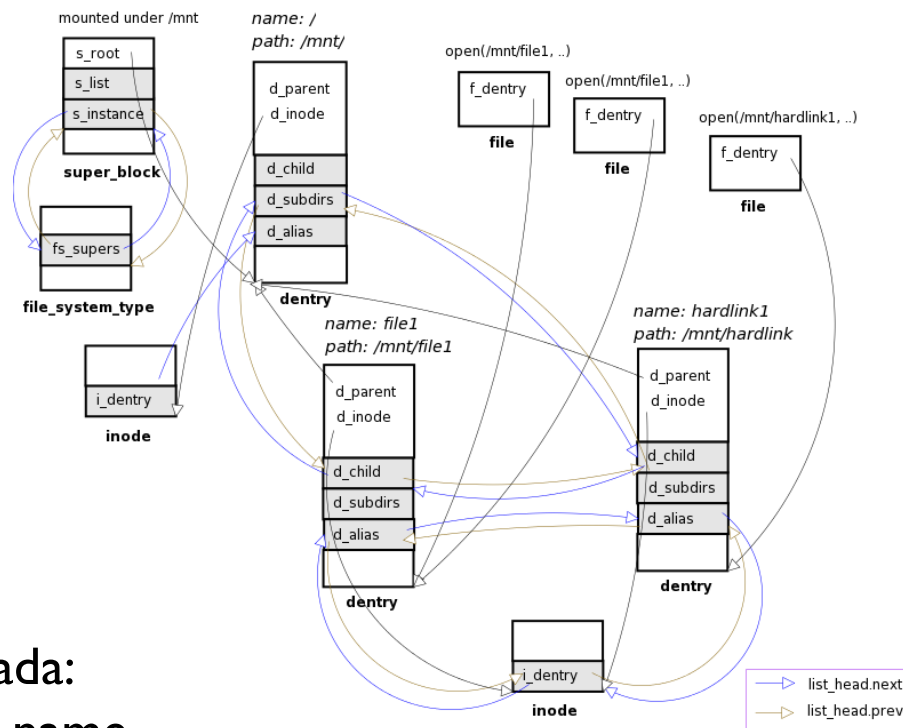
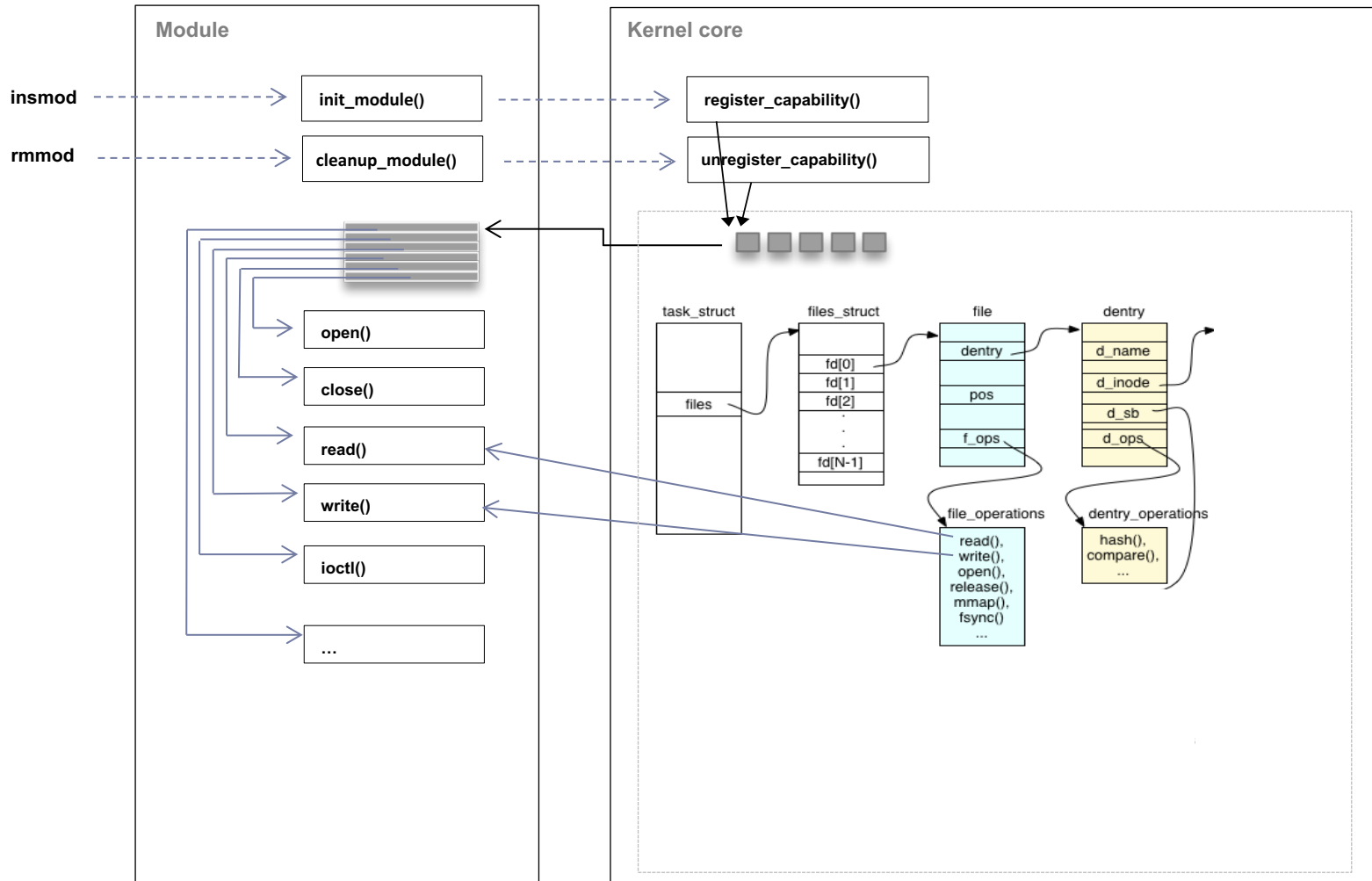


Fig: Relationships between the VFS objects

► **Posible camino asociado a una entrada:**  
`x->d_parent->d_name + '/' + x->d_name`

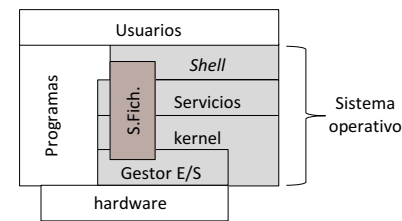
- Si es la raíz de un sistema montado (`dentry->d_covers != dentry`) entonces es el camino del punto de montaje `d_covers`.
- Si es la raíz de un sistema de ficheros (`dentry->d_parent == dentry`) entonces es "/" (`== d_name`)

# Estructuras principales de gestión tabla de d-entradas (entradas de directorio)



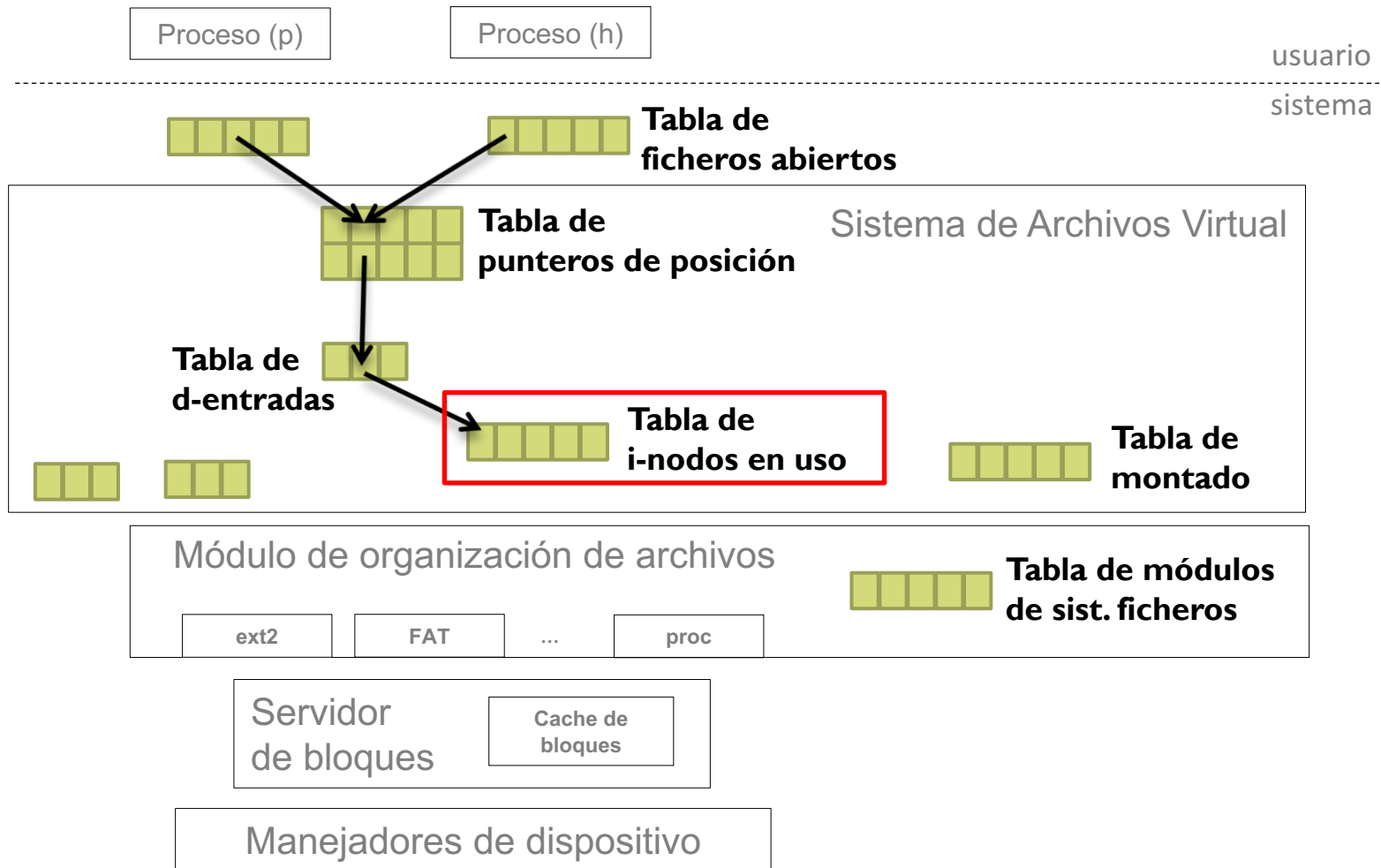
# (0) ~~Objetivos~~ requisitos principales

ej.: sistema de ficheros tipo Unix



- ▶ Lograr la **persistencia de los datos del usuario**, buscando **minimizar el impacto en el rendimiento y en el espacio para metadatos**.
- ▶ Los procesos usarán una **interfaz de trabajo segura**, sin acceso directo a la información usada en el kernel.
- ▶ **Compartir el puntero de posición de ficheros** entre procesos con relación de parentesco.
- ▶ Poder tener **una sesión de trabajo con un fichero/directorio** para actualizar la información que contiene.
- ▶ Poder tener **una sesión de trabajo con varios directorios** para poder recorrer sus entradas.
- ▶ **Llevar la pista de los sistemas de ficheros presentes** en el kernel, **y llevar la pista de los puntos donde están siendo usados**.

# Estructuras principales de gestión





# Estructuras principales de gestión

## tabla de i-nodos (ficheros en uso)

---

- ▶ Almacena en memoria la información de los i-nodo en uso . Hay dos tipos de información:
  - ▶ La existente en el disco
  - ▶ La que se usa dinámicamente y que sólo tiene sentido cuando el archivo está en uso.
- ▶ Así mismo guarda punteros a las funciones propias.
  - ▶ En Linux/Unix están asociadas con gestión de metadatos:
    - ▶ Funciones para creación y borrado de ficheros.
    - ▶ Funciones para creación de descriptores (información de los ficheros abiertos por cada proceso).
    - ▶ Función de mapeo de bloques de fichero a bloques de disco.

# Estructuras principales de gestión tabla de i-nodos: Linux



```
struct inode {  
    unsigned long    i_ino;  
    umode_t          i_mode;  
    uid_t            i_uid;  
    gid_t            i_gid;  
    kdev_t           i_rdev;  
    loff_t           i_size;  
    struct timespec  i_atime;  
    struct timespec  i_ctime;  
    struct timespec  i_mtime;  
    struct super_block *i_sb;  
    struct inode_operations *i_op;  
    struct address_space *i_mapping;  
    struct list_head i_dentry;  
    ...  
};
```

# Estructuras principales de gestión

## tabla de i-nodos: Linux



### struct inode\_operations {

```
int (*create) (struct inode *,
              struct dentry *, int);
```

```
int (*unlink) (struct inode *,
              struct dentry *);
```

```
int (*mkdir) (struct inode *,
             struct dentry *, int);
```

```
int (*rmdir) (struct inode *,
             struct dentry *);
```

```
int (*mknod) (struct inode *,
             struct dentry *,
             int, dev_t);
```

```
int (*rename) (struct inode *,
             struct dentry *,
             struct inode *,
             struct dentry *);
```

```
void (*truncate) (struct inode *);
```

```
struct dentry * (*lookup) (struct inode *,
                          struct dentry *);
```



```
int (*permission) (struct inode *, int);
```

```
int (*setattr) (struct dentry *,
              struct iattr *);
```

```
int (*getattr) (struct vfsmount *mnt,
              struct dentry *,
              struct kstat *);
```

```
int (*setxattr) (struct dentry *,
              const char *,
              const void *,
              size_t, int);
```

```
ssize_t (*getxattr) (struct dentry *,
                  const char *,
                  void *, size_t);
```

```
ssize_t (*listxattr) (struct dentry *,
                  char *, size_t);
```

```
int (*removexattr) (struct dentry *,
                  const char *);
```



```
int (*link) (struct dentry *,
            struct inode *,
            struct dentry *);
```

```
int (*symlink) (struct inode *,
              struct dentry *,
              const char *);
```

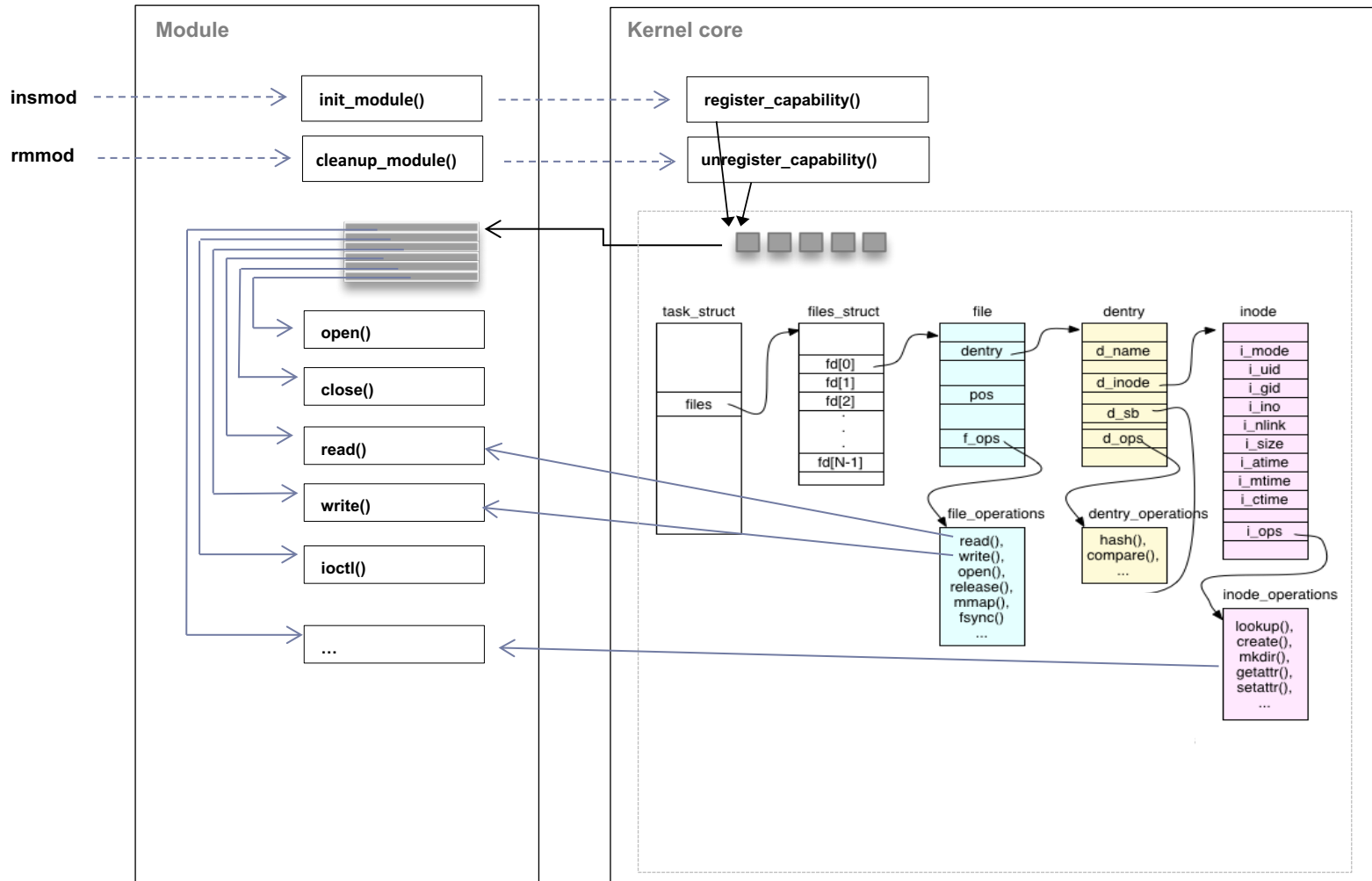
```
int (*readlink) (struct dentry *,
                char *, int);
```

```
int (*follow_link) (struct dentry *,
                  struct nameidata *);
```



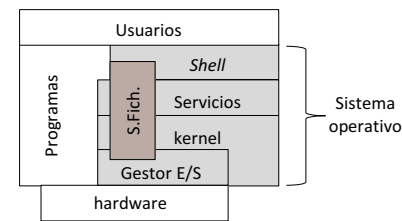
```
};
```

# Estructuras principales de gestión tabla de i-nodos: Linux



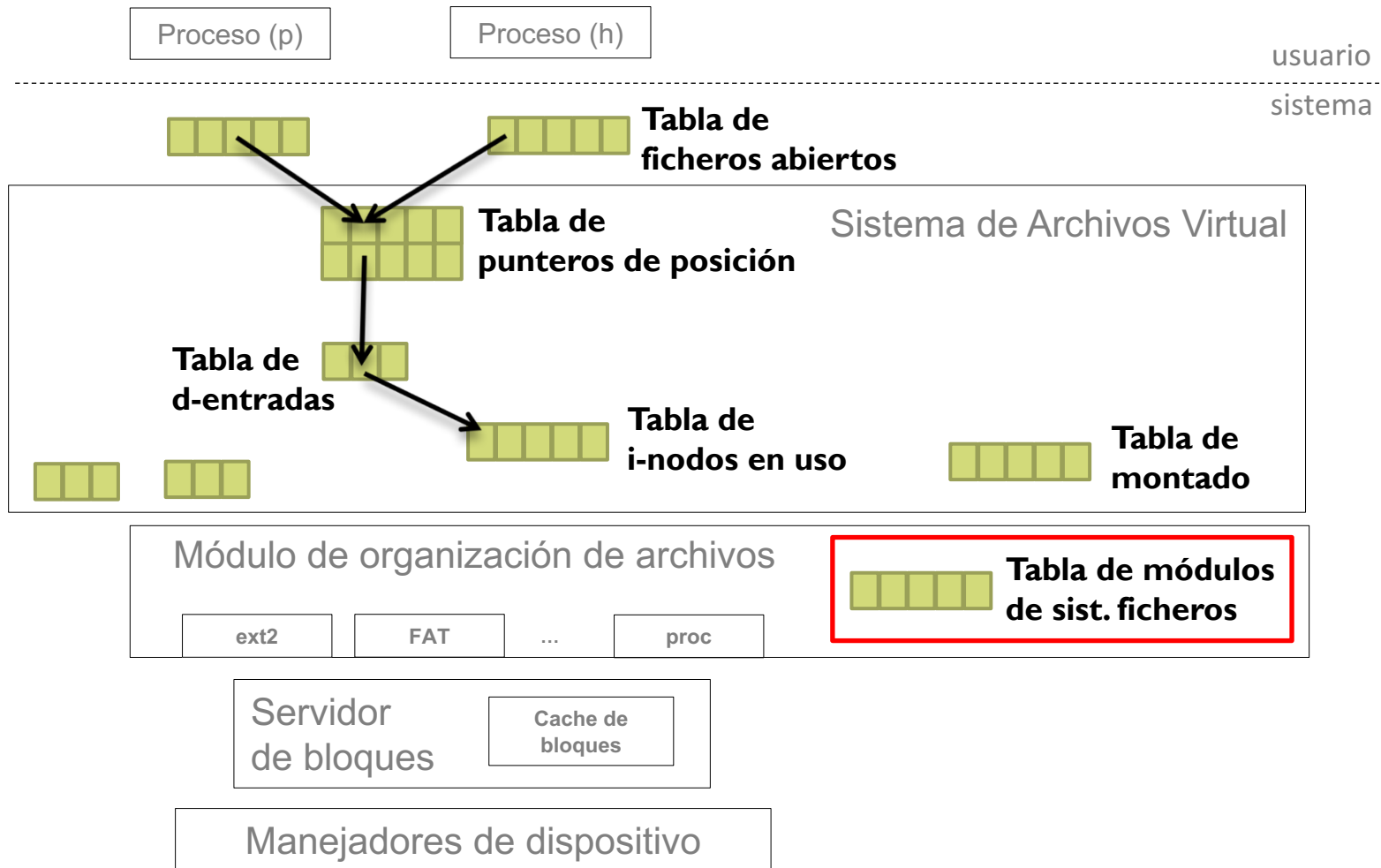
# (0) ~~Objetivos~~ requisitos principales

ej.: sistema de ficheros tipo Unix



- ▶ Lograr la **persistencia de los datos del usuario**, buscando **minimizar el impacto en el rendimiento y en el espacio para metadatos**.
- ▶ Los procesos usarán una **interfaz de trabajo segura**, sin acceso directo a la información usada en el kernel.
- ▶ **Compartir el puntero de posición de ficheros** entre procesos con relación de parentesco.
- ▶ Poder tener **una sesión de trabajo con un fichero/directorio** para actualizar la información que contiene.
- ▶ Poder tener **una sesión de trabajo con varios directorios** para poder recorrer sus entradas.
- ▶ **Llevar la pista de los sistemas de ficheros presentes en el kernel, y llevar la pista de los puntos donde están siendo usados.**

# Estructuras principales de gestión



# Estructuras principales de gestión tabla de sistemas de ficheros

---

- ▶ Almacena en memoria la información sobre los sistemas de archivos cuyo módulo está cargado en el kernel.
  - ▶ Existen funciones para registrar/borrar módulos de nuevos sistemas de ficheros.
- ▶ Así mismo guarda punteros a las funciones propias.
  - ▶ Las funciones asociadas con el propio sistema de ficheros son las de mount/umount `[[obtener/liberar superbloque]]`.

# Estructuras principales de gestión tabla de sistemas de ficheros: Linux

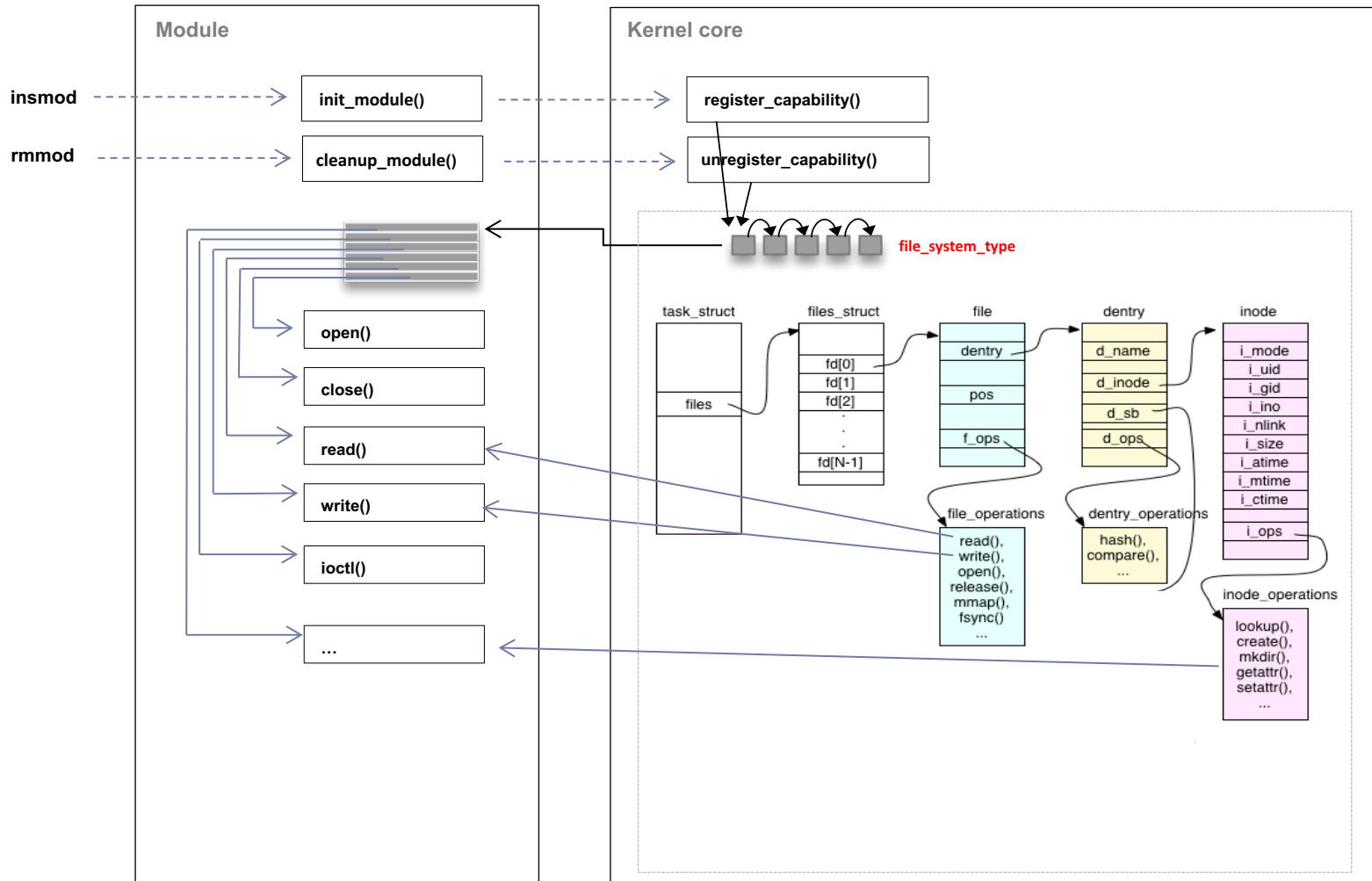


file\_systems

```
struct file_system_type {  
    const char *name;  
    int fs_flags;  
    struct dentry *(*mount) (struct file_system_type *,  
                             int, const char *, void *);  
    void (*kill_sb) (struct super_block *);  
    struct module *owner;  
    struct file_system_type *next;  
    struct list_head fs_supers;  
    struct lock_class_key s_lock_key;  
    ...  
}
```

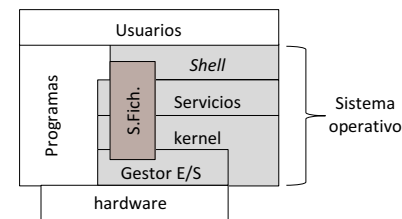


# Estructuras principales de gestión tabla de sistemas de archivos: Linux



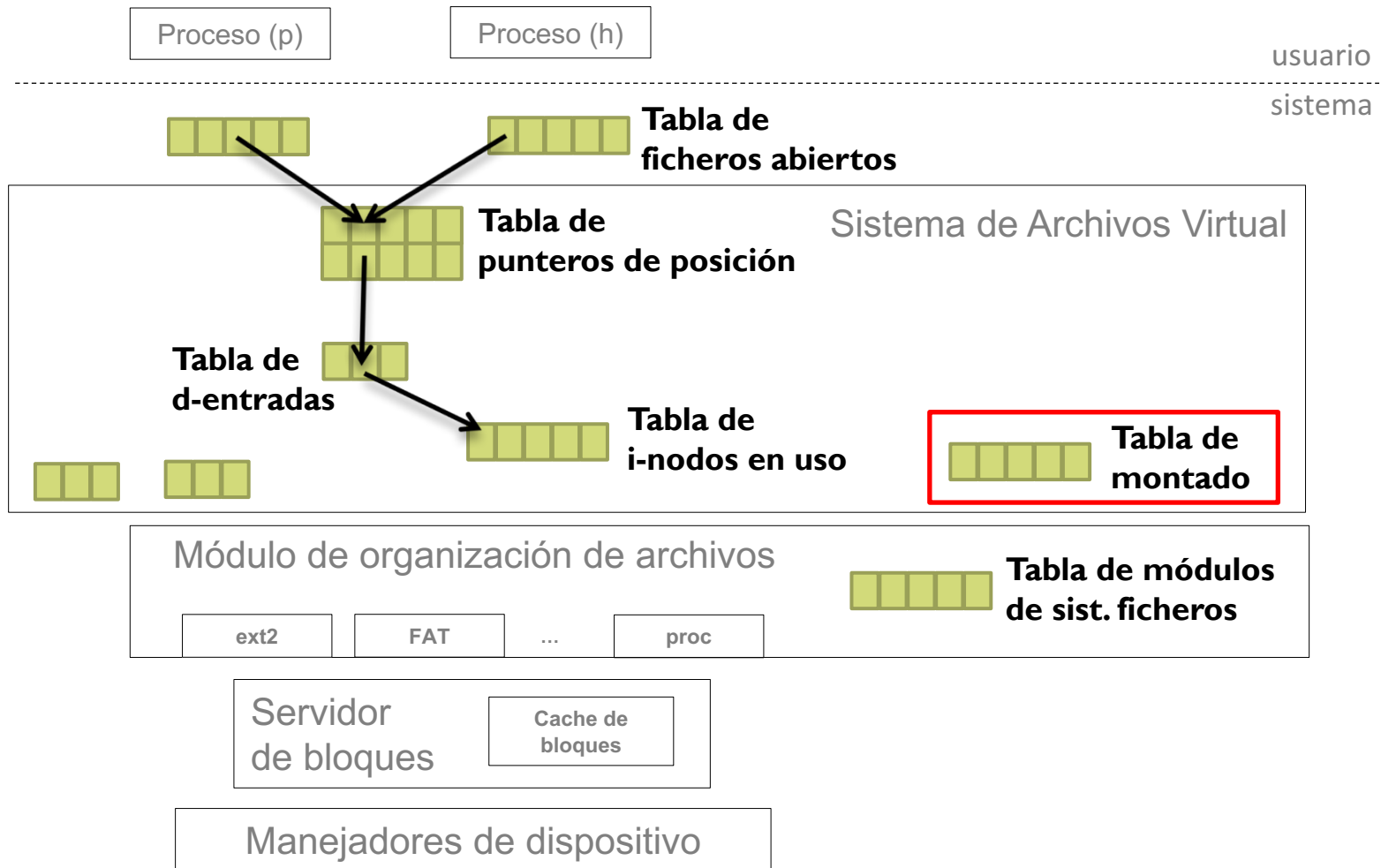
# (0) ~~Objetivos~~ requisitos principales

ej.: sistema de ficheros tipo Unix



- ▶ Lograr la **persistencia de los datos del usuario**, buscando **minimizar el impacto en el rendimiento y en el espacio para metadatos**.
- ▶ Los procesos usarán una **interfaz de trabajo segura**, sin acceso directo a la información usada en el kernel.
- ▶ **Compartir el puntero de posición de ficheros** entre procesos con relación de parentesco.
- ▶ Poder tener **una sesión de trabajo con un fichero/directorio** para actualizar la información que contiene.
- ▶ Poder tener **una sesión de trabajo con varios directorios** para poder recorrer sus entradas.
- ▶ **Llevar la pista de los sistemas de ficheros presentes en el kernel, y llevar la pista de los puntos donde están siendo usados.**

# Estructuras principales de gestión



# Estructuras principales de gestión tabla de superbloques / montaje

---

- ▶ Almacena en memoria la información sobre todos los volúmenes montados en el sistema.
  - ▶ Guarda los datos del superbloque de cada volumen.
  - ▶ Suele incluir el d-entry del directorio donde está montado.
  - ▶ Suele incluir el d-entry del directorio raíz del volumen.
- ▶ Así mismo guarda punteros a las funciones propias.
  - ▶ Los tipos de funciones asociadas con el superbloque son:
    - ▶ Funciones de trabajo con el superbloque.
    - ▶ Funciones auxiliares para el montaje.
    - ▶ Función para trabajar con los i-nodos del volumen.

# Estructuras principales de gestión tabla de montajes: Linux



current->namespace->list

```
struct vfsmount {  
    struct vfsmount *mnt_parent; /* fs we are mounted on */  
    struct dentry *mnt_mountpoint; /* dentry of mountpoint */  
    struct dentry *mnt_root; /* root of the mounted tree */  
    struct super_block *mnt_sb; /* pointer to superblock */  
    struct list_head mnt_hash;  
    struct list_head mnt_mounts; /* list of children, anchored here */  
    struct list_head mnt_child;  
    struct list_head mnt_list;  
    atomic_t mnt_count;  
    int mnt_flags;  
    char *mnt_devname; /* Device name, e.g. /dev/hda1 */  
};
```

# Estructuras principales de gestión tabla de montaje (superbloque): Linux



current->namespace->list->mnt\_sb

```
struct super_block {  
    dev_t                s_dev;  
    unsigned long        s_blocksize;  
    struct file_system_type *s_type;  
    struct super_operations *s_op;  
    struct dentry         *s_root;  
    ...  
};
```

# Estructuras principales de gestión tabla de montaje (superbloque): Linux



```
struct super_operations {
```

```
    struct inode *(*alloc_inode)(struct super_block *sb);
```

```
    void (*destroy_inode)(struct inode *);
```

```
    void (*read_inode) (struct inode *);
```

```
    void (*dirty_inode) (struct inode *);
```

```
    void (*write_inode) (struct inode *, int);
```

```
    void (*put_inode) (struct inode *);
```

```
    void (*drop_inode) (struct inode *);
```

```
    void (*delete_inode) (struct inode *);
```

```
    void (*clear_inode) (struct inode *);
```



```
void (*put_super) (struct super_block *);
```

```
void (*write_super) (struct super_block *);
```

```
int (*sync_fs)(struct super_block *sb, int wait);
```

```
void (*write_super_lockfs) (struct super_block *);
```

```
void (*unlockfs) (struct super_block *);
```

```
int (*statfs) (struct super_block *, struct statfs *);
```

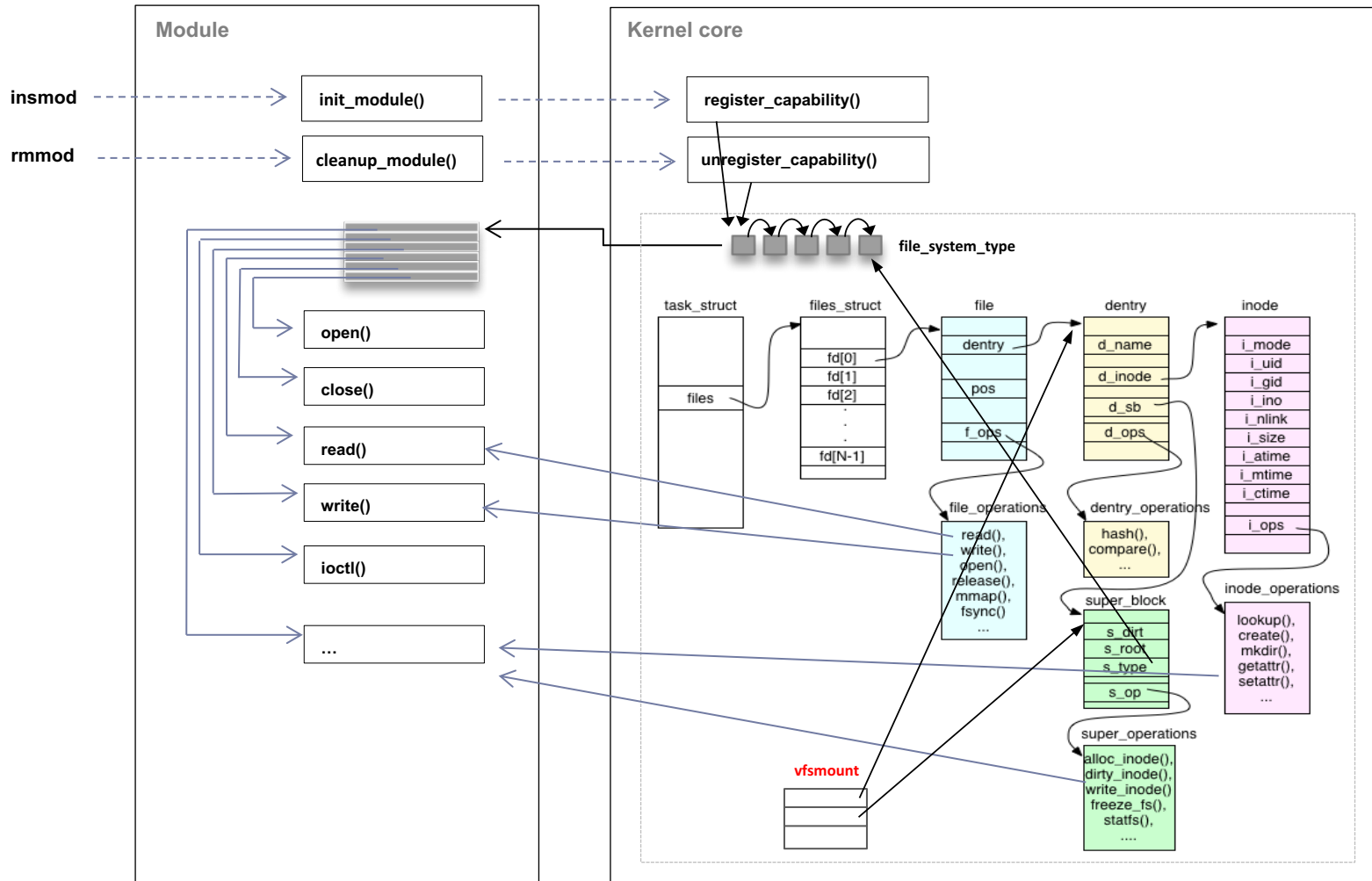
```
int (*remount_fs) (struct super_block *, int *, char *);
```

```
void (*umount_begin) (struct super_block *);
```

```
int (*show_options)(struct seq_file *, struct vfsmount *);
```

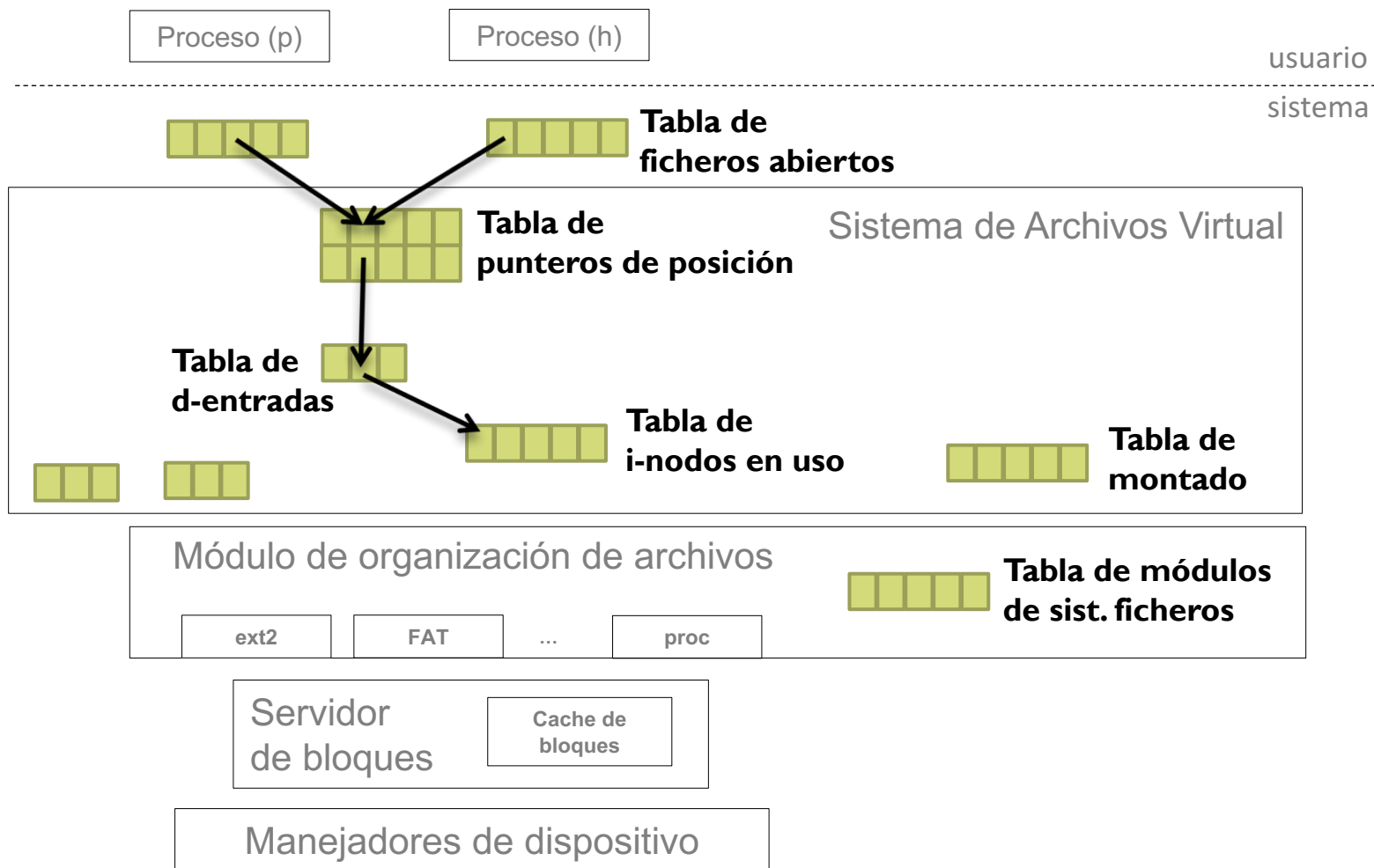
```
};
```

# Estructuras principales de gestión tabla de montaje (superbloque): Linux

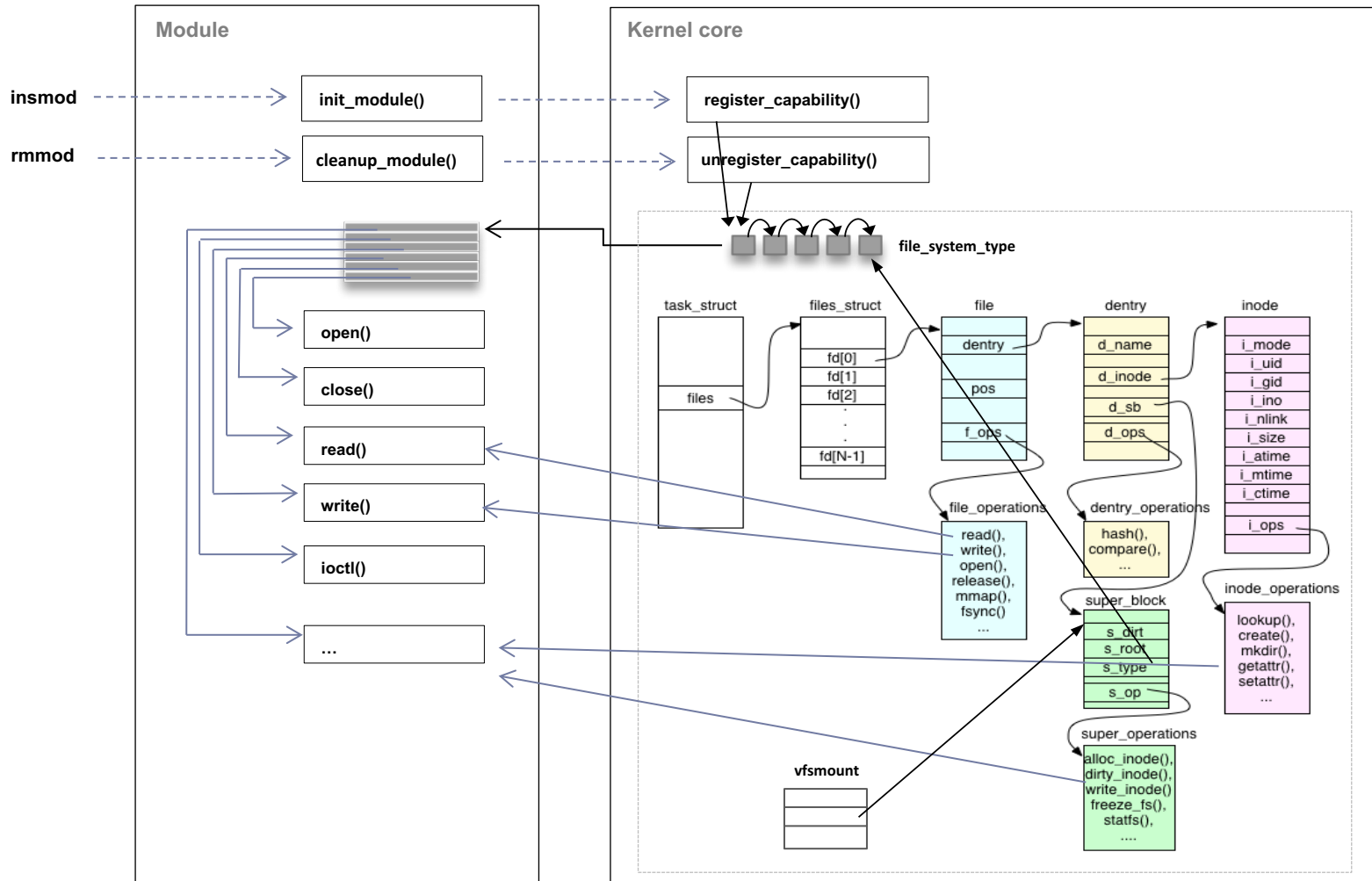




# Estructuras principales de gestión resumen (dependencias)



# Estructuras principales de gestión resumen (uso)



# Objetivos principales

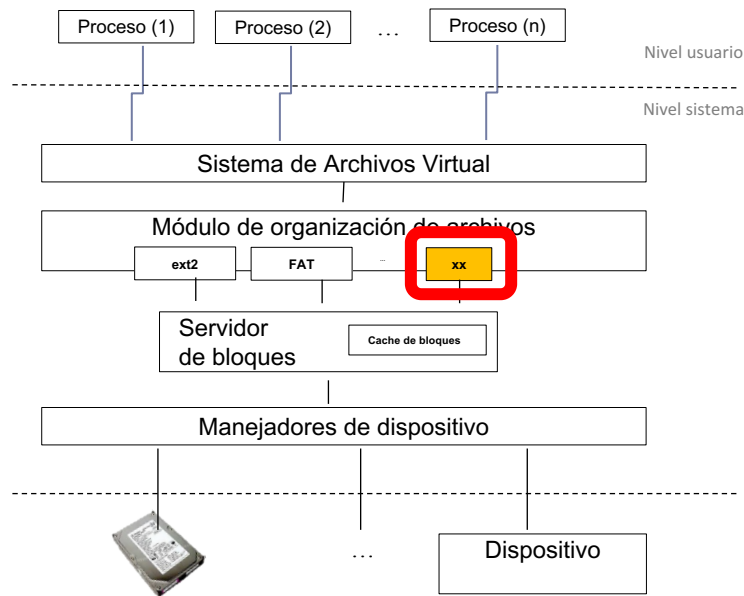
## resumen (requisitos)

---



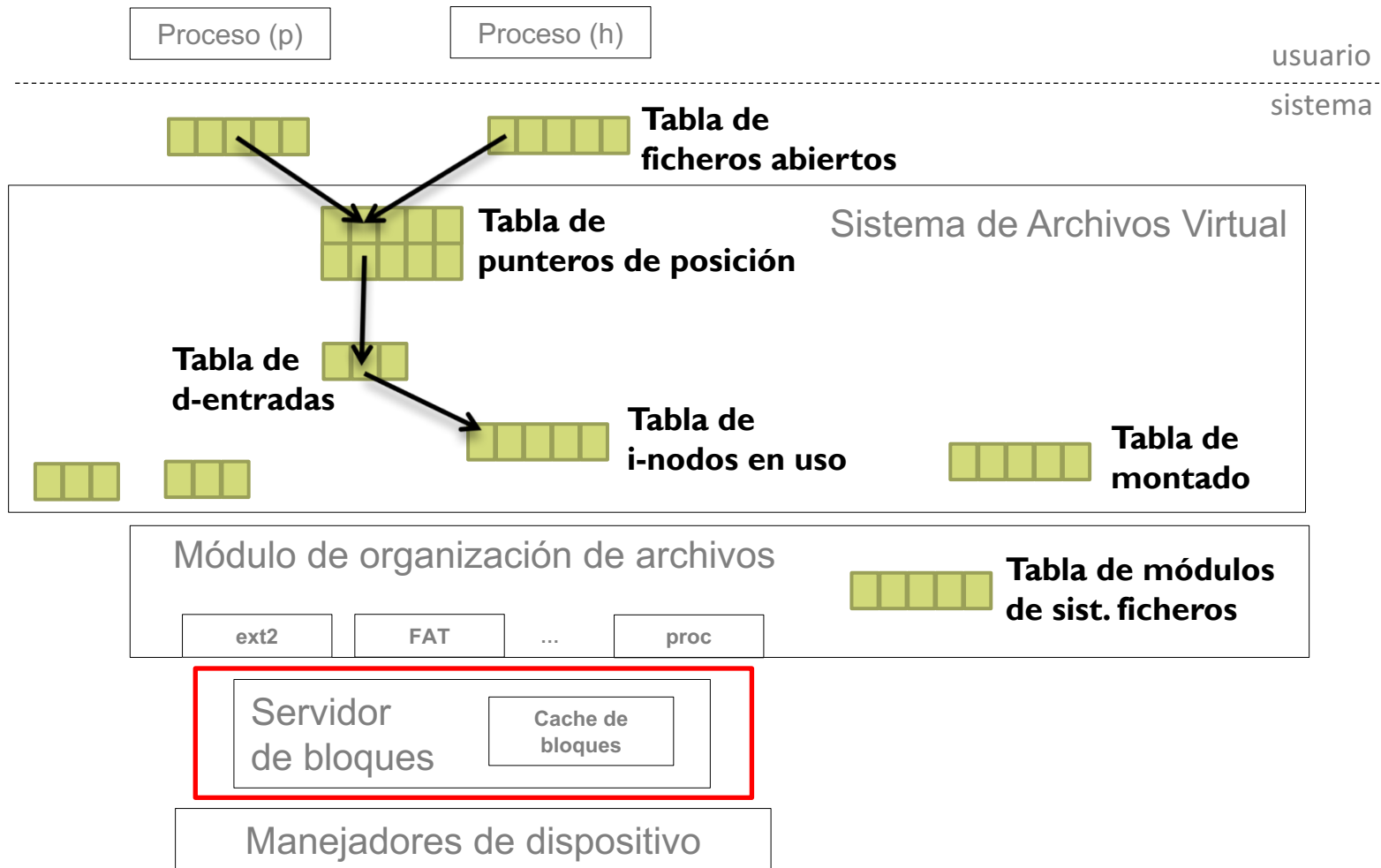
- ✓ ▶ Los procesos usarán una interfaz de trabajo segura, sin acceso directo a la información usada en el kernel.
- ✓ ▶ Compartir el puntero de posición de ficheros entre procesos con relación de parentesco.
- ✓ ▶ Poder tener una sesión de trabajo con un fichero/directorio para actualizar la información que contiene.
- ✓ ▶ Poder tener una sesión de trabajo con varios directorios para poder recorrer sus entradas.
- ✓ ▶ Lograr la persistencia de los datos del usuario, buscando minimizar el impacto en el rendimiento y en el espacio para metadatos.
- ✓ ▶ Llevar la pista de los sistemas de ficheros dados de alta en el kernel, y llevar la pista de los puntos donde están siendo usados.

# Aspectos a tener en cuenta para añadir un sistema de ficheros...



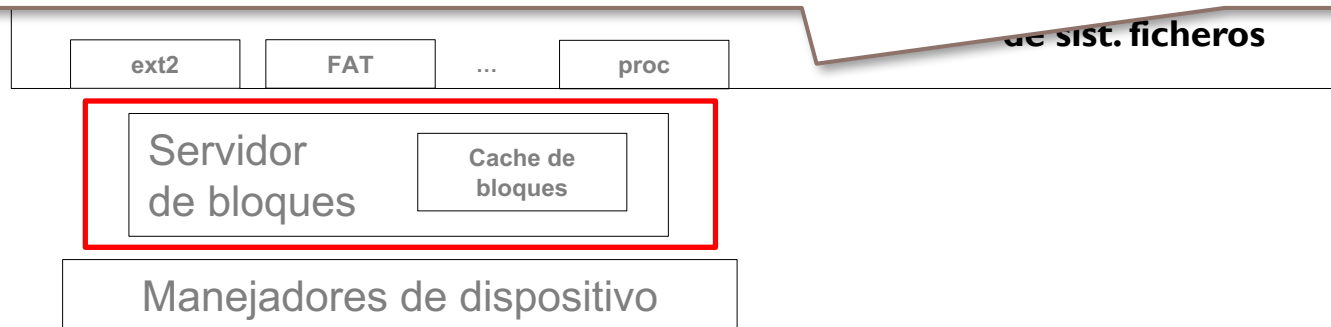
- ▶ (0) Requisitos del sistema.
- ▶ (1) Estructuras en disco.
- ▶ (2) Estructuras en memoria.
- ▶ **Caché de bloques.**
- ▶ (3a) Funciones de gestión de estructuras disco/memoria.
- ▶ (3b) Funciones de llamadas al sistema.

# Usaremos una caché de bloques...



# Usaremos una caché de bloques...

- ▶ **getblk:** busca/reserva en caché un bloque de un v-nodo, con desplazamiento y tamaño dado.
- ▶ **brelease:** libera un buffer y lo pasa a la lista de libres.
- ▶ **bwrite:** escribe un bloque de la caché a disco.
- ▶ **bread:** lee un bloque de disco a caché.
- ▶ **breada:** lee un bloque (y el siguiente) de disco a caché.



# Servidor de bloques

---

- ▶ Se encarga de:
  - ▶ Emitir los mandatos genéricos para leer y escribir bloques a los manejadores de dispositivo (usando las rutinas específicas de cada dispositivo).
  - ▶ Optimizar las peticiones de E/S.
    - ▶ Ej.: cache de bloques.
  - ▶ Ofrecer un nombrado lógico para los dispositivos.
    - ▶ Ej.: /dev/hda3 (tercera partición del primer disco)

# Servidor de bloques

---

## ▶ Funcionamiento general:

- ▶ Si el bloque está en la cache
  - ▶ Copiar el contenido (y actualizar los metadatos de uso del bloque)
- ▶ Si no está en la caché
  - ▶ Leer el bloque del dispositivo y guardarlo en la cache
  - ▶ Copiar el contenido (y actualizar los metadatos)
  - ▶ Si el bloque ha sido escrito (sucio / *dirty*)
    - Política de escritura
  - ▶ Si la cache está llena, es necesario hacer hueco
    - Política de reemplazo



# Servidor de bloques

---

## ▶ Funcionamiento general:

- **Lectura adelantada** (*read-ahead*):
  - Leer un número de bloques a continuación del requerido y se almacena en caché (mejora el rendimiento en accesos consecutivos)
- ▶ Leer el bloque del dispositivo y guardarlo en la cache
- ▶ Copiar el contenido (y actualizar los metadatos)
- ▶ Si el bloque ha sido escrito (sucio / *dirty*)
  - Política de escritura
- ▶ Si la cache está llena, es necesario hacer hueco
  - Política de reemplazo

# Servidor de bloques

---

- **Escritura inmediata** (*write-through*):
  - Se escribe cada vez que se modifica el bloque (– rendimiento, + fiabilidad)
- **Escritura diferida** (*write-back*):
  - Sólo se escriben los datos a disco cuando se eligen para su reemplazo por falta de espacio en la cache (+ rendimiento, –fiabilidad)
- **Escritura retrasada** (*delayed-write*):
  - Escribir a disco los bloques de datos modificados en la cache de forma periódica cada cierto tiempo (30 segundos en UNIX) (compromiso entre anteriores)
- **Escritura al cierre** (*write-on-close*):
  - Cuando se cierra un archivo, se vuelcan al disco los bloques del mismo.

▶ Si el bloque no está escrito (sucio / *dirty*)

□ Política de escritura

▶ Si la cache está llena, es necesario hacer hueco

□ Política de reemplazo

# Servidor de bloques

---

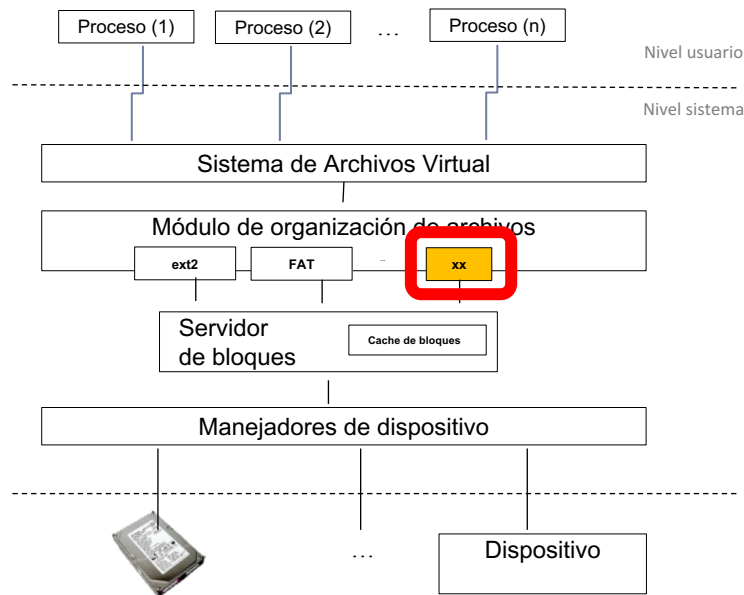
## ▶ Funcionamiento general:

- ▶ Si el bloque está en la cache
  - ▶ Copiar el contenido (y actualizar los metadatos de uso del bloque)
- ▶ Si no está en la caché
  - ▶ Leer el bloque del dispositivo y guardarlo en la cache

- **FIFO** (*First in First Out*)
- **Algoritmo del reloj** (*Segunda oportunidad*)
- **MRU** (*Most Recently Used*)
- **LRU** (*Least Recently Used*)

- Política de reemplazo

# Aspectos a tener en cuenta para añadir un sistema de ficheros...



- ▶ (0) Requisitos del sistema.
- ▶ (1) Estructuras en disco.
- ▶ (2) Estructuras en memoria.
- ▶ Caché de bloques.
- ▶ **(3a) Funciones de gestión de estructuras disco/memoria.**
- ▶ (3b) Funciones de llamadas al sistema.

# (3a) Gestión de estructuras disco/memoria...

Llamadas al sistema de archivos

Descriptor	Uso de <i>namei</i>	Asig. i-n.	Atributos	E/S.	Sist. Arch.	Vista
open pipe	open chown unlink	creat	chown	read	mount	chdir
creat close	creat chmod mknod	mknod	chmod	write	umount	chroot
dup	chdir stat mount	link	stat	lseek		
	chroot link umount	unlink				

Algoritmos de bajo nivel del sistema de archivos

namei	ialloc	alloc	bmap
iget	ifree	free	



Algoritmos de gestión de bloques/caché

getblk	brelse	bread	breada	bwrite
--------	--------	-------	--------	--------



# Ejemplo de rutinas de gestión

## i-nodos

- ▶ **namei:** convierte una ruta al i-nodo asociado.
- ▶ **iget:** devuelve un i-nodo de la tabla de i-nodos y si no está lo lee de memoria secundaria, lo añade a la tabla de i-nodos y lo devuelve.
- ▶ **iput:** libera un i-nodo de la tabla de i-nodos, y si es necesario lo actualiza en memoria secundaria.
- ▶ **ialloc:** asignar un i-nodo a un fichero.
- ▶ **ifree:** libera un i-nodo previamente asignado a un fichero.

Algoritmos de bajo nivel del sistema de archivos

namei	ialloc	alloc	bmap
iget	iput	free	

d-entradas



punteros de posición

ficheros abiertos

montajes



i-nodos en uso



Algoritmos de gestión de bloques/caché

getblk	brelse	bread	breada	bwrite
--------	--------	-------	--------	--------

módulos de s. ficheros



# Ejemplo de rutinas de gestión

## bloques

- ▶ **bmap**: calcula el bloque de disco asociado a un desplazamiento del fichero. Traduce direcciones lógicas (offset de fichero) a físicas (bloque de disco).
- ▶ **alloc**: asigna un bloque a un fichero.
- ▶ **free**: libera un bloque previamente asignado a un fichero.

Algoritmos de bajo nivel del sistema de archivos

namei	ialloc	alloc	bmap
iget	iput	free	

d-entradas



punteros de posición

ficheros abiertos

montajes



i-nodos en uso



Algoritmos de gestión de bloques/caché

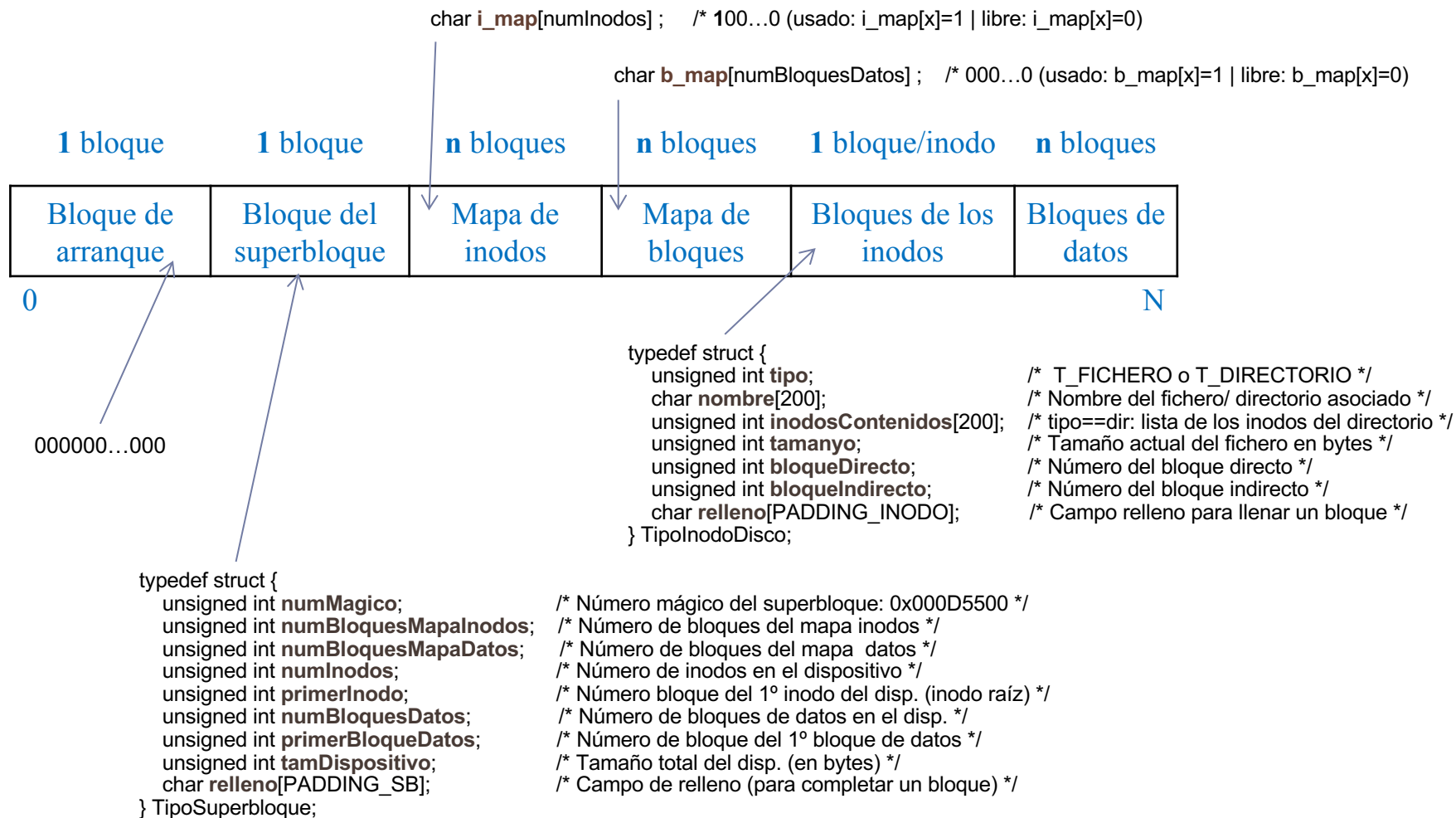
getblk	brelse	bread	breada	bwrite
--------	--------	-------	--------	--------

módulos de s. ficheros





# Ejemplo de organización en disco







# Ejemplo de organización en memoria...

---

```
// Información leída del disco
TipoSuperbloque sbloques [1] ;
char i_map [numInodo] ;
char b_map [numBloquesDatos] ;
TipoInodoDisco inodos [numInodo] ;

// Información extra de apoyo
struct {
    int posicion;
    int abierto;
} inodos_x [numInodo] ;

...
```



# Ejemplo: ialloc y alloc

```
int ialloc ( void )
{
    // buscar un i-nodo libre
    for (int=0; i<sbloques[0].numInodos; i++)
    {
        if (i_map[i] == 0) {
            // inodo ocupado ahora
            i_map[i] = 1;
            // valores por defecto en el i-nodo
            memset(&(inodos[i]), 0,
                sizeof(TipolnodoDisco));
            // devolver identificador de i-nodo
            return i;
        }
    }

    return -1;
}
```

```
int alloc ( void )
{
    char b[BLOCK_SIZE];

    for (int=0; i<sbloques[0].numBloquesDatos; i++)
    {
        if (b_map[i] == 0) {
            // bloque ocupado ahora
            b_map[i] = 1;
            // valores por defecto en el bloque
            memset(b, 0, BLOCK_SIZE);
            bwrite(DISK, sbloques[0].primerBloquesDatos + i, b);
            // devolver identificador del bloque
            return i;
        }
    }

    return -1;
}
```



# Ejemplo: ifree y free

---

```
int ifree ( int inodo_id )
{
    // comprobar validez de inodo_id
    if (inodo_id > sbloques[0].numInodos)
        return -1;

    // liberar i-nodo
    i_map[inodo_id] = 0;

    return -1;
}
```

```
int free ( int block_id )
{
    // comprobar validez de block_id
    if (block_id > sbloques[0].numBloquesDatos)
        return -1;

    // liberar bloque
    b_map[block_id] = 0;

    return -1;
}
```



## Ejemplo: namei y bmap

```
int namei ( char *fname )
{
    // buscar i-nodo con nombre <fname>
    for (int=0; i<sbloques[0].numInodos; i++)
    {
        if (! strcmp(inodos[i].nombre, fname))
            return i;
    }

    return -1;
}
```

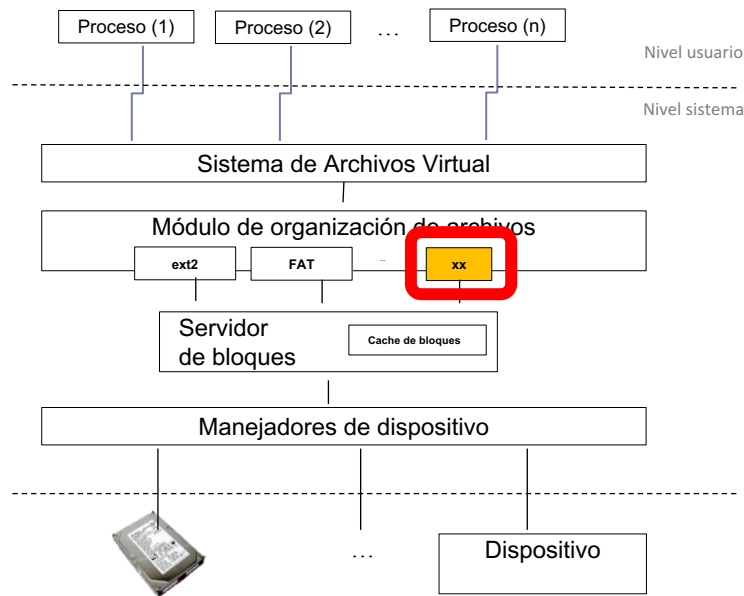
```
int bmap ( int inodo_id, int offset )
{
    int b[BLOCK_SIZE/4];

    // comprobar validez de inodo_id
    if (inodo_id > sbloques[0].numInodos)
        return -1;

    // bloque de datos asociado
    if (offset < BLOCK_SIZE)
        return inodos[inodo_id].bloqueDirecto;
    if (offset < BLOCK_SIZE*BLOCK_SIZE/4) {
        bread(DISK, sbloques[0].primerBloqueDatos +
              inodos[inodo_id].bloqueIndirecto, b);
        offset = (offset - BLOCK_SIZE) / BLOCK_SIZE;
        return b[offset] ;
    }

    return -1;
}
```

# Aspectos a tener en cuenta para añadir un sistema de ficheros...



- ▶ (0) Requisitos del sistema.
- ▶ (1) Estructuras en disco.
- ▶ (2) Estructuras en memoria.
- ▶ Caché de bloques.
- ▶ (3a) Funciones de gestión de estructuras disco/memoria.
- ▶ **(3b) Funciones de llamadas al sistema.**

# (3b) Llamadas al sistema...

## Llamadas al sistema de archivos

Descriptor	Uso de <i>namei</i>	Asig. i-n.	Atributos	E/S.	Sist. Arch.	Vista
open pipe	open chown unlink	creat	chown	read	mount	chdir
creat close	creat chmod mknod	mknod	chmod	write	umount	chroot
dup	chdir stat mount	link	stat	lseek		
	chroot link umount	unlink				

## Algoritmos de bajo nivel del sistema de archivos

namei	ialloc	alloc	bmap
iget	ifree	free	



## Algoritmos de gestión de bloques/caché

getblk	brelease	bread	breada	bwrite
--------	----------	-------	--------	--------



# Ejemplo de ll. al sistema

- ▶ **open**: localiza el i-nodo asociado al camino del fichero, ...
- ▶ **read**: localiza el bloque de datos, leer bloque de datos, ...
- ▶ **write**: localiza el bloque de datos, escribir bloque de datos, ...
- ▶ ...

## Llamadas al sistema de archivos

Descriptor	Uso de <i>namei</i>	Asig. i-n.	Atributos	E/S.	Sist. Arch.	Vista
open pipe	open chown unlink	creat	chown	read	mount	chdir
creat close	creat chmod mknod	mknod	chmod	write	umount	chroot
dup	chdir stat mount	link	stat	lseek		
	chroot link umount	unlink				

## Algoritmos de bajo nivel del sistema de archivos

namei	ialloc	alloc	bmap
iget	ifree	free	



## Algoritmos de gestión de bloques/caché

getblk	brelse	bread	breada	bwrite
--------	--------	-------	--------	--------





# Ejemplo: mount

---

```
int mount ( void )
{
    // leer bloque 0 de disco en sbloques[0]
    bread(DISK, 1, &(sbloques[0]) );

    // leer los bloques para el mapa de i-nodos
    for (int=0; i<sbloques[0].numBloquesMapaInodos; i++)
        bread(DISK, 2+i, ((char *)i_map + i*BLOCK_SIZE) );

    // leer los bloques para el mapa de bloques de datos
    for (int=0; i<sbloques[0].numBloquesMapaDatos; i++)
        bread(DISK, 2+i+sbloques[0].numBloquesMapaInodos, ((char *)b_map + i*BLOCK_SIZE);

    // leer los i-nodos a memoria
    for (int=0; i<(sbloques[0].numInodos*sizeof(TipoInodoDisco)/BLOCK_SIZE); i++)
        bread(DISK, i+sbloques[0].primerInodo, ((char *)inodos + i*BLOCK_SIZE);

    return 1;
}
```





# Ejemplo: umount

---

```
int umount ( void )
{
    // escribir bloque 0 de sbloques[0] a disco
    bwrite(DISK, 1, &(sbloques[0]) );

    // escribir los bloques para el mapa de i-nodos
    for (int=0; i<sbloques[0].numBloquesMapaInodos; i++)
        bwrite(DISK, 2+i, ((char *)i_map + i*BLOCK_SIZE) );

    // escribir los bloques para el mapa de bloques de datos
    for (int=0; i<sbloques[0].numBloquesMapaDatos; i++)
        bwrite(DISK, 2+i+sbloques[0].numBloquesMapaInodos, ((char *)b_map + i*BLOCK_SIZE);

    // escribir los i-nodos a disco
    for (int=0; i<(sbloques[0].numInodos*sizeof(TipoInodoDisco)/BLOCK_SIZE); i++)
        bwrite(DISK, i+sbloques[0].primerInodo, ((char *)inodos + i*BLOCK_SIZE);

    return 1;
}
```



# Ejemplo: open y close

---

```
int open ( char *nombre )
{
    int inodo_id ;

    inodo_id = namei(nombre) ;
    if (inodo_id < 0)
        return inodo_id ;

    inodos_x[inodo_id].posicion = 0;
    inodos_x[inodo_id].abierto = 1;

    return inodo_id;
}
```

```
int close ( int fd )
{
    if (fd < 0)
        return fd ;

    inodos_x[fd].posicion = 0;
    inodos_x[fd].abierto = 0;

    return 1;
}
```



# Ejemplo: creat y unlink

```
int creat ( char *nombre )
{
    int b_id, inodo_id ;

    inodo_id = ialloc() ;
    if (inodo_id < 0) { return inodo_id ; }
    b_id = alloc();
    if (b_id < 0) { ifree(inodo_id); return b_id ; }

    inodos[inodo_id].tipo = 1 ; // FICHERO
    strcpy(inodos[inodo_id].nombre, nombre);
    inodos[inodo_id].bloqueDirecto = b_id ;
    inodos_x[inodo_id].posicion = 0;
    inodos_x[inodo_id].abierto = 1;

    return 1;
}
```

```
int unlink ( char * nombre )
{
    int inodo_id ;

    inodo_id = namei(nombre) ;
    if (inodo_id < 0)
        return inodo_id ;

    free(inodos[inodo_id].bloqueDirecto);
    memset(&(inodos[inodo_id]),
           0,
           sizeof(TipoInodoDisco));
    ifree(inodo_id) ;

    return 1;
}
```



# Ejemplo: read y write

```
int read ( int fd, char *buffer, int size )
{
    char b[BLOCK_SIZE] ;
    int b_id ;

    if (inodos_x[fd].posicion+size > inodos[fd].size)
        size = inodos[fd].size - inodos_x[fd].posicion;
    if (size =< 0)
        return 0;

    b_id = bmap(fd, inodos_x[fd].posicion);
    bread(DISK,
        sbloques[0].primerBloqueDatos+b_id, b);
    memmove(buffer,
        b+inodos_x[fd].posicion, size);
    inodos_x[fd].posicion += size;

    return size;
}
```

```
int write ( int fd, char *buffer, int size )
{
    char b[BLOCK_SIZE] ;
    int b_id ;

    if (inodos_x[fd].posicion+size > BLOCK_SIZE)
        size = BLOCK_SIZE - inodos_x[fd].posicion;
    if (size =< 0)
        return 0;

    b_id = bmap(fd, inodos_x[fd].posicion);
    bread(DISK, sbloques[0].primerBloqueDatos+b_id, b);
    memmove(b+inodos_x[fd].posicion,
        buffer, size);
    bwrite(DISK, sbloques[0].primerBloqueDatos+b_id, b);
    inodos_x[fd].posicion += size;

    return size;
}
```

# (4) Utilidad *mkfs*...

Utilidad de creación del sistema de ficheros

Llamadas al sistema de archivos

Descriptor	Uso de <i>namei</i>	Asig. i-n.	Atributos	E/S.	Sist. Arch.	Vista
open pipe	open chown unlink	creat	chown	read	mount	chdir
creat close	creat chmod mknod	mknod	chmod	write	umount	chroot
dup	chdir stat mount	link	stat	lseek		
	chroot link umount	unlink				

Algoritmos de bajo nivel del sistema de archivos

namei	ialloc	alloc	bmap
iget	ifree	free	

d-entradas



punteros de posición

ficheros abiertos

montajes



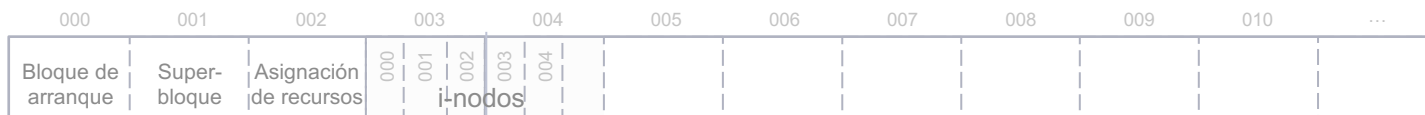
i-nodos en uso



Algoritmos de gestión de bloques/caché

getblk	brelse	bread	breada	bwrite
--------	--------	-------	--------	--------

módulos de s. ficheros





# Ejemplo: mkfs

---

```
int mkfs ( void )
{
    // inicializar a los valores por defecto del superbloque, mapas e i-nodos
    sbloques[0].numMagico = 1234; // ayuda a comprobar que se haya creado por nuestro mkfs
    sbloques[0].numInodos = numInodo;
    ...
    for (int=0; i<sbloques[0].numInodos; i++)
        i_map[i] = 0; // free
    for (int=0; i<sbloques[0].numBloquesDatos; i++)
        b_map[i] = 0; // free
    for (int=0; i<sbloques[0].numInodos; i++)
        memset(&(inodos[i]), 0, sizeof(TipoInodoDisco) );

    // to write the default file system into disk
    umount();

    return 1;
}
```

# Ejemplo de rutinas de gestión

## resumen

### Llamadas al sistema de archivos

Descriptor	Uso de <i>namei</i>	Asig. i-n.	Atributos	E/S.	Sist. Arch.	Vista
open pipe	open chown unlink	creat	chown	read	mount	chdir
creat close	creat chmod mknod	mknod	chmod	write	umount	chroot
dup	chdir stat mount	link	stat	lseek		
	chroot link umount	unlink				

### Algoritmos de bajo nivel del sistema de archivos

namei	ialloc	alloc	bmap
iget	ifree	free	



### Algoritmos de gestión de bloques/caché

getblk	brelse	bread	breada	bwrite
--------	--------	-------	--------	--------



# Contenidos

---

1. Introducción
2. Marco de trabajo
3. Diseño y desarrollo de un sistema de ficheros
4. **Aspectos complementarios**



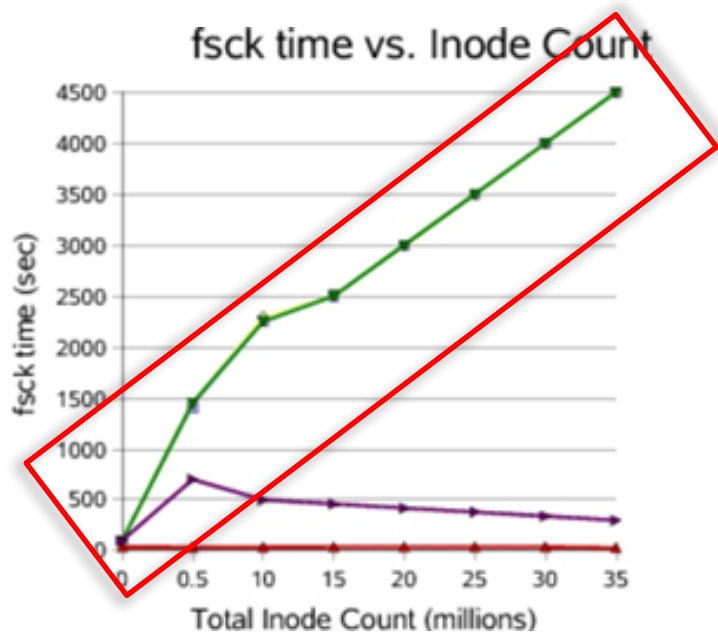
# Características avanzadas

---



- ▶ **Journaling**
- ▶ **Snapshots**
- ▶ **Dynamic file system expansion**

# sin Journaling



- ▶ El sistema de ficheros tradicional realiza las modificaciones directamente.
- ▶ Si se produce una parada no prevista (*system crash*), la recuperación consiste en repasar todos los metadatos buscando fallos e inconsistencias:
  - ▶ El tiempo de reparación es proporcional al tamaño del sistema de ficheros (hay que repasar todo)

# sin *Journaling*

ejemplo de comprobaciones a realizar (hay más)

---

## ▶ Sistema de ficheros en disco:

- ▶ Se comprueba que el contenido del superbloque responde a las características del sistema de archivos.
- ▶ Se comprueba que los mapas de bits de nodos-i se corresponden con los nodos-i ocupados en el sistema de archivos.
- ▶ Se comprueba que los mapas de bits de bloques se corresponden con los bloques asignados a archivos.
- ▶ Se comprueba que ningún bloque esté asignado a más de un archivo.

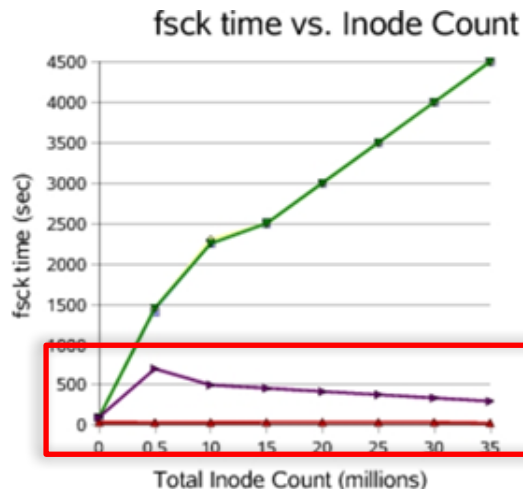
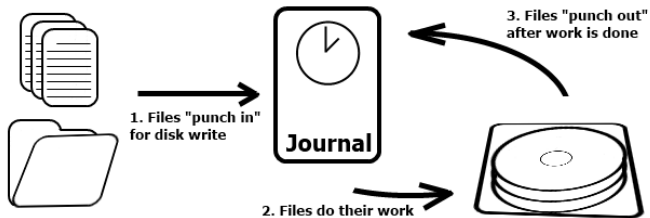
## ▶ Directorios:

- ▶ Se comprueba el sistema de directorios del sistema de archivos, para ver que un mismo nodo-i no está asignado a más de un directorio.

## ▶ Archivos:

- ▶ Se comprueba que los bits de protección y privilegios.
- ▶ Se comprueba el contador de enlaces.

# con Journaling



- ▶ El sistema de ficheros escribe cada cambio en un registro (*log*) de forma transparente a las aplicaciones antes de hacer los cambios definitivos.
- ▶ Si se produce una parada no prevista, la recuperación consiste en repasar el registro y hacer las modificaciones pendientes (*commit*):
  - ▶ El tiempo de reparación es proporcional a los cambios pendientes en el *log*, no al tamaño del sistema de almacenamiento: se puede pasar de horas a segundos

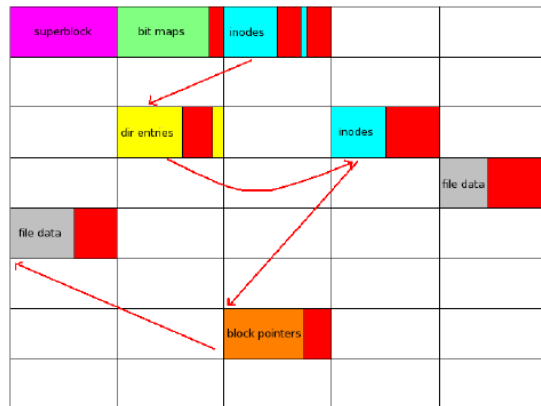
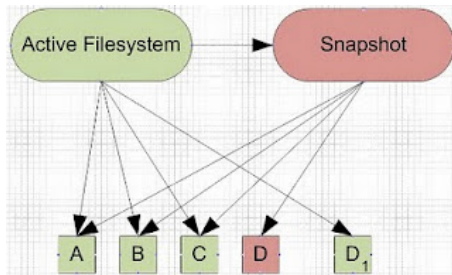
# Características avanzadas

---



- ▶ Journaling
- ▶ Snapshots
- ▶ Dynamic file system expansion

# Snapshot



- ▶ Snapshot congela el estado de un sistema de ficheros en un instante dado:
  - ▶ Se realiza en pocos segundos.
  - ▶ Es posible acceder a esta copia congelada a través de un directorio especial.
- ▶ Ej.: actualizaciones del sistema, copias de seguridad continuas, etc.

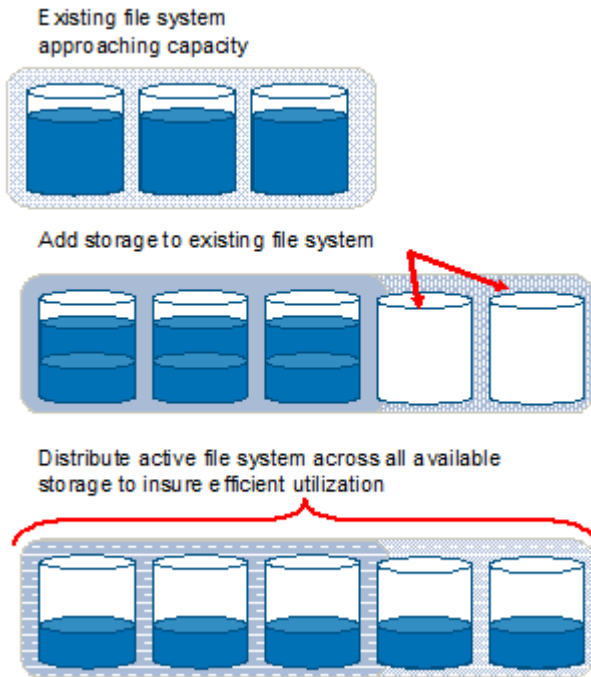
# Características avanzadas

---



- ▶ Journaling
- ▶ Snapshots
- ▶ Dynamic file system expansion

# Dynamic file system expansion



- ▶ Es importante diseñar el sistema de ficheros para que si se precisa cambiar el tamaño del mismo, se pueda hacer sin perder información y de forma eficiente:
  - ▶ Metadatos repartidos.
  - ▶ Estructuras dinámicas y flexibles.



# Lección 4

## Sistemas de ficheros

Diseño de Sistemas Operativos  
Grado en Ingeniería Informática