Sistemas Operativos

Sistemas de Ficheros

Agenda

Agenda

Tipos de ficheros en POSIX

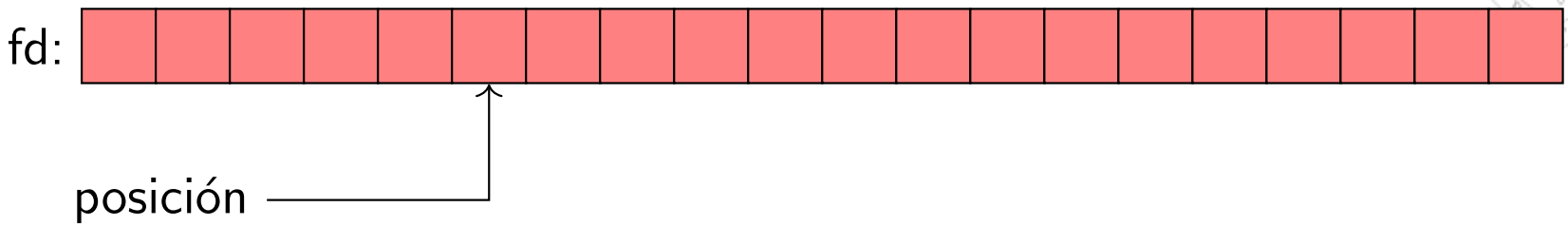
- Regular
- Directorio
- Enlace Simbólico
- Tubería con nombre (FIFO)
- Fichero de dispositivo (caracteres o bloques)
- Socket

Atributos de un fichero en POSIX

El sistema asigna una serie de atributos a un fichero:

- Nombre (incluye su extensión)
- Tipo
- Tamaño
- Fecha de creación
- Fecha de última modificación
- Propietario, grupo
- Permisos de lectura, escritura y ejecución
  - distintos para propietario, grupo y otros

Modelo



El modelo de fichero que el SO presenta al programador es:

- una secuencia ordenada de bytes
  - byte **n**: saltar **n** bytes desde el comienzo
- un marcador o puntero de posición del fichero en dicha secuencia
  - no confundir con una variable tipo puntero
  - una lectura o escritura hacen avanzar este puntero
- un descriptor de fichero (fd) para manejarlo
  - Asociado a un apertura concreta del fichero (permisos y posición)
- operaciones básicas sobre el fichero

API POSIX para ficheros

```
int open(const char *pathname, int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);
int close(int fd);
off_t lseek(int fd, off_t offset, int whence);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
int unlink(const char *pathname);
int stat(const char *pathname, struct stat *statbuf);
int fstat(int fd, struct stat *statbuf);
int lstat(const char *pathname, struct stat *statbuf);
```

- **lseek**: desplazamiento del puntero de posición
- **unlink**: borrado de archivo (nombre)
- **stat, lstat, fstat**: obtención de atributos de un fichero

Se recomienda consultar las páginas de manual de estas llamadas al sistema

POSIX: descriptor de fichero

En POSIX un fichero se maneja con un descriptor de fichero obtenido al abrirlo con la siguiente llamada al sistema:

```
int open(const char *path, int oflag, mode_t mode);
```

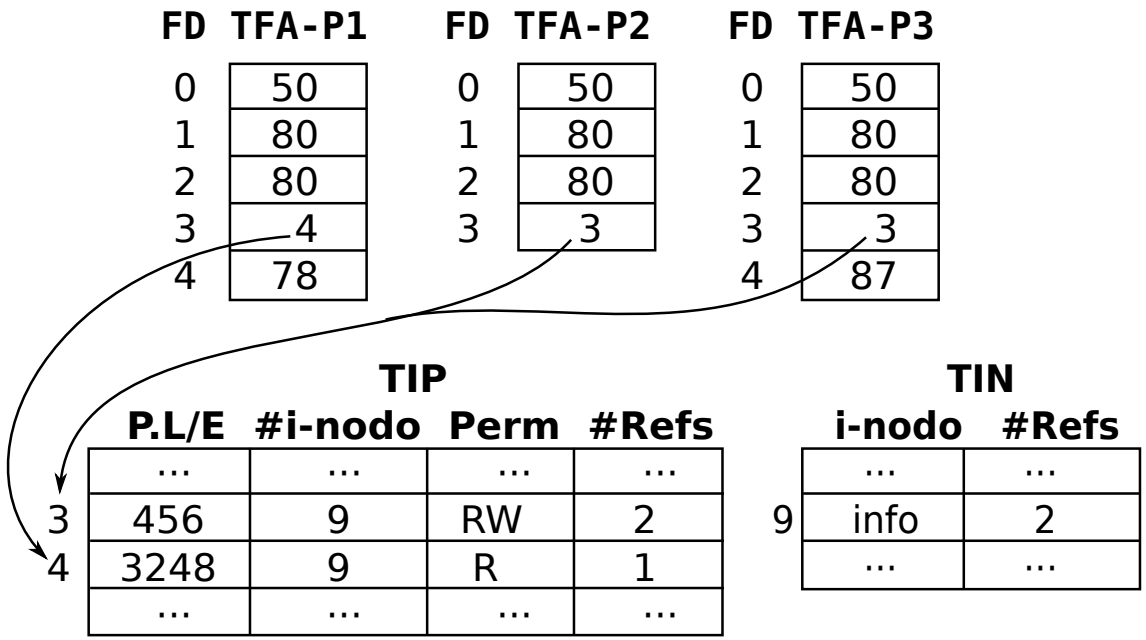
El descriptor es un entero:

- Indica la posición correspondiente en la Tabla de Descriptores de Ficheros Abiertos (TFA) del proceso
- La TFA es una estructura gestionada por el SO para el proceso
- La apertura se realiza en un modo indicado por los `flags`
  - Máscara de bits: `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_APPEND`, ...
  - Permisos de lectura y/o escritura
  - Qué hacer si existe (colocarse al final, borrar, ...) o si no existe
- El parámetro opcional `mode` indica los permisos de usuario, grupo y otros que hay que dar al fichero si se crea (flag `O_CREAT`)

Se recomienda consultar la página de manual de `open` (`# man 2 open`)

Tabla de descriptores abiertos (TFA)

- Una tabla con una entrada por descriptor de fichero abierto (TFA)
- Cada entrada contiene una referencia a la entrada de la Tabla Intermedia de Posiciones (TIP) con la información de la apertura:
  - El puntero de posición, los permisos de la apertura, nº de procesos que usan la entrada, etc.
- POSIX: Cuando un proceso crea otro (`fork()`) la TFA se copia del proceso padre al hijo, incrementando el nº de referencias en la TIP:
  - Comparten el puntero de L/E



fopen vs. open

```
int open(const char *pathname, int flags);
```

- Se usa un entero como identificador de la apertura
  - Es una entrada de una tabla de descriptores abiertos del proceso
- Sólo válida en sistemas POSIX
- El argumento flags vinculado con las opciones de apertura de los sistemas POSIX

```
FILE *fopen(const char *pathname, const char *mode);
```

- Se usa una estructura FILE como identificador de la apertura
  - `fopen` devuelve la dirección (puntero) de la estructura usada
- Los modos de apertura se identifican con distintas cadenas de caracteres: “r”, “r+”, “w”, “w+”, “a”, “a+”
- Independiente de sistema

Buffer intermedio

Las llamadas al sistema son costosas:

- Cada llamada a `open`, `close`, `read`, `write`, `...` supone una excepción software

La biblioteca estándar de C usa buffers intermedios

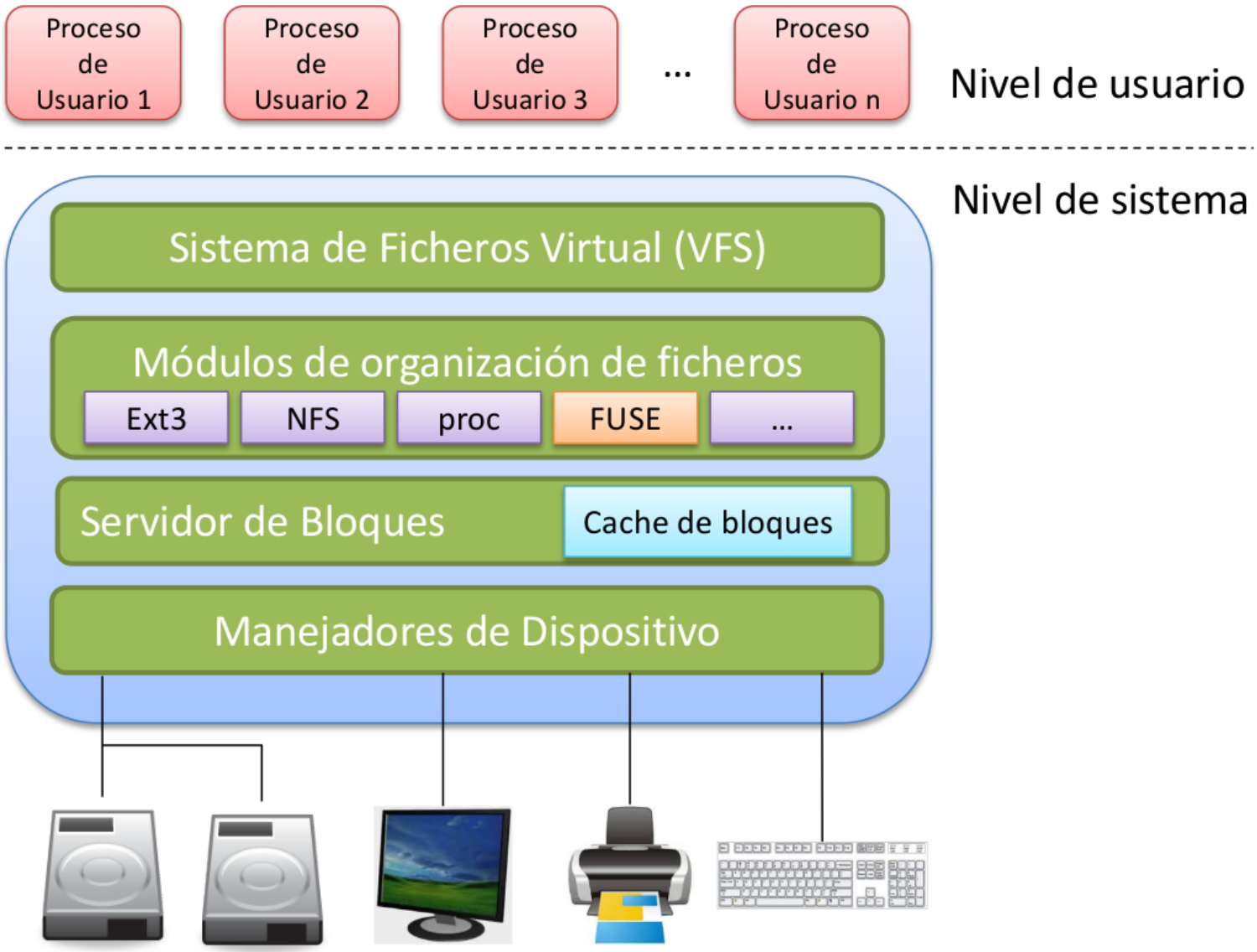
- Llamada consecutivas a `fread` o `fwrite` suponen lecturas o escrituras del buffer intermedio.
- Cuando corresponda, según la política de buffer activa, se hace la llamada al sistema para leer o escribir del fichero real (escritura/lectura en bloque):
  - Porque se llena o vacía el buffer
  - Porque leamos/escribimos un final de línea

  La política se puede cambiar con `setvbuf`

```
int setvbuf(FILE *stream, char *buf,
            int mode, size_t size);
```

Agenda

Servidor de ficheros



Sistema de Ficheros Virtual

Capa de abstracción que se encarga de:

- Abstraer los detalles de implementación de los distintos SSFF
- Hacer las comprobaciones de errores comunes a todos los SSFF
  - Ejemplo: permisos de acceso
- Establecer un API común para todos los sistemas de ficheros
  - Ejemplo: API POSIX a través de `struct file_operations`

Maneja descriptores virtuales (nodos-i virtuales)

- Contienen una referencia a la información que necesita el módulo de organización de ficheros correspondiente.
  - Ejemplo: referencia al nodo-i de un SF ext3 montado
- Permite completar información no disponible en algunos SSFF
  - Ejemplo: asigna propietario y grupo a un fichero de un SF FAT montado en un sistema POSIX

Redirige la llamada al módulo de organización de ficheros correspondiente

Módulos de organización de ficheros

- Proporcionan el modelo de fichero del sistema operativo e implementan las operaciones básicas
  - Un módulo por cada SF soportado (UNIX, AFS, Windows NT, MS-DOS, EFS, MINIX, etc.).
  - Relacionan el modelo lógico con su almacenamiento real, traduciendo desplazamientos (offsets) lógicos a números de bloques físicos
  - Gestionan el espacio, la asignación de bloques, el manejo de los descriptores internos, etc.
- También hay módulos para pseudoficheros (`dev`, `proc`).
  - Son SF virtuales que el SO utiliza para ofrecer información interna y/o dar algún servicio

Servidor de bloques

- Aisla de los detalles de los controladores de disco a los SSFF
- API simple: leer/escribir un bloque de disco
- Implementa una Cache de bloques:
  - Los bloques más recientemente accedidos se dejan copiados en memoria, mejorando el rendimiento del sistema
  - En cada acceso se comprueba si el bloque está en la cache.
  - Si no está se copia del disco a la cache.
  - Si la cache está llena, hay que quitar un bloque para hacer hueco: políticas de reemplazo.
  - Si el bloque ha sido escrito (sucio): política de escritura.

Cache: Políticas de reemplazo

- FIFO (First In First Out)
- MRU (Most Recently Used)
- LRU (Least Recently Used)
  - Política más común, se reemplaza el bloque que ha estado sin ser referenciado durante más tiempo. Los bloques más usados se encuentran en RAM.

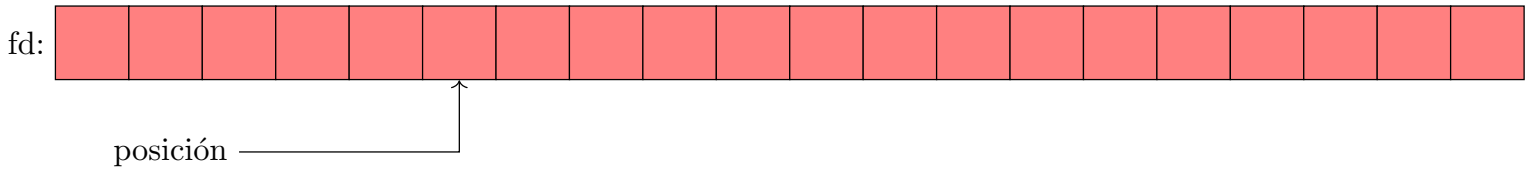
Cache: Políticas de escritura

- Escritura inmediata (write-through): cada actualización en cache implica escritura en disco. Rendimiento malo.
- Escritura diferida (write-back): un bloque se escribe en disco solo cuando se selecciona para su reemplazo en la cache.
  - Optimiza el rendimiento, pero genera problemas de fiabilidad
- Escritura retrasada (delayed-write), periódicamente se escriben a disco los bloques modificados (30s en UNIX).
  - Compromiso entre rendimiento y fiabilidad.
  - Los bloques especiales se escriben inmediatamente al disco.
  - Hay que volcar los datos de la cache antes de quitar un disco
- Escritura al cierre (write-on-close): cuando se cierra un archivo se escriben en disco los bloques modificados

Agenda

Programador vs. SO

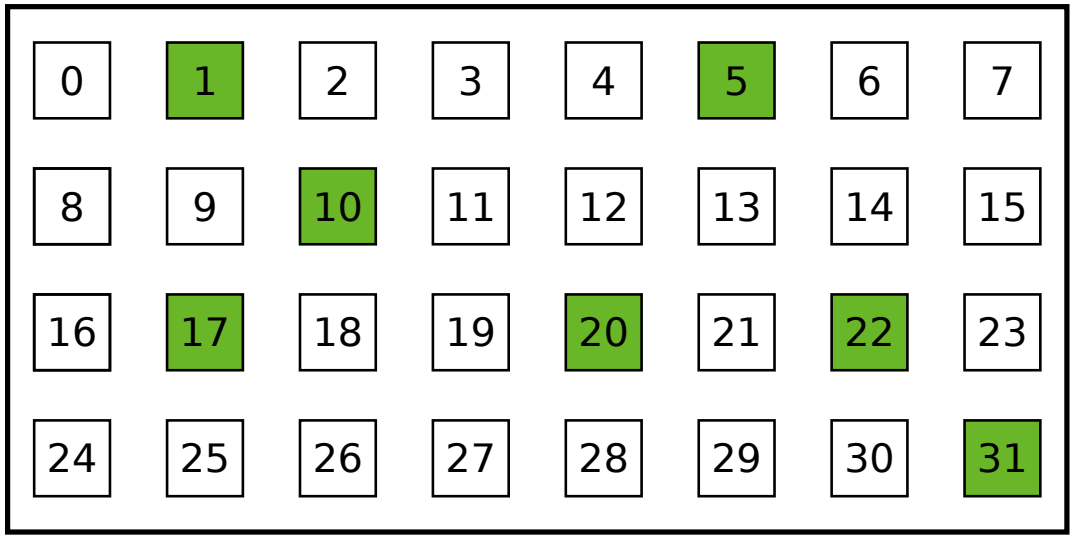
- Programador: array de bytes con un puntero de posición



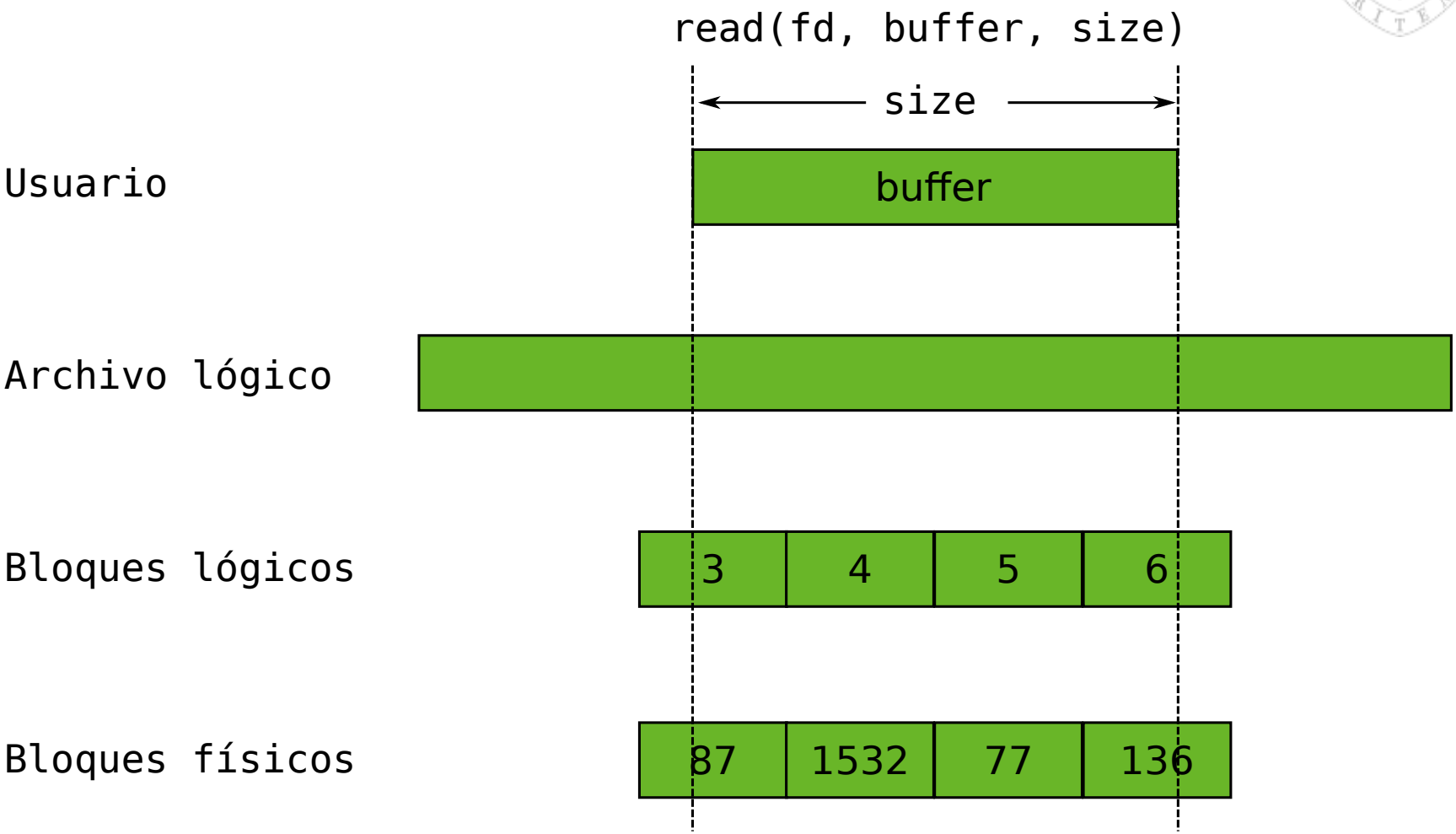
- SO: un conjunto de bloques del disco

Bloques del Archivo X

0	17
1	22
2	1
3	5
4	20
5	31
6	10



Operaciones en ficheros



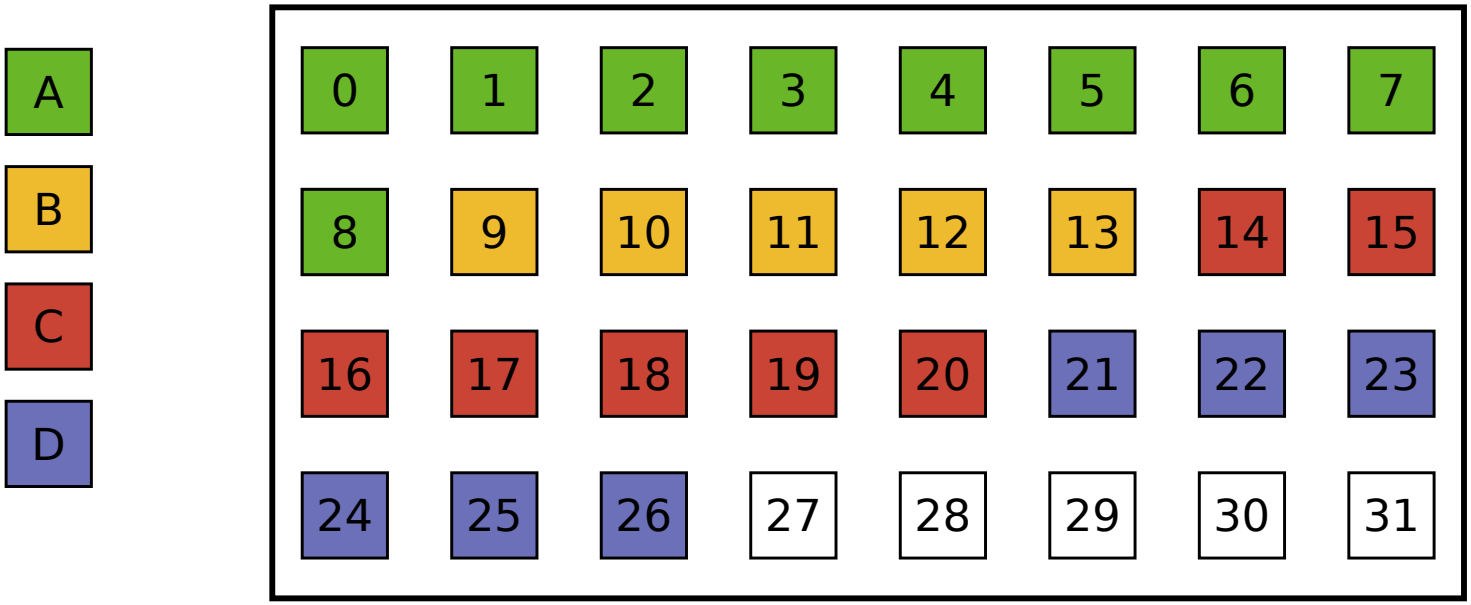
Asignación de bloques

- Bloques contiguos:
  - Usado en CD-ROM y cintas
- Bloques enlazados:
  - Usado en sistemas FAT (File Allocation Table)
- Bloques indexados:
  - UFS, FFS (Fast File System), ext2, ext3
- Árboles de extents (grupos de bloques contiguos)
  - NTFS, JFS, Reiser4, HFS, XFS, ...

Agenda

Bloques Contiguos

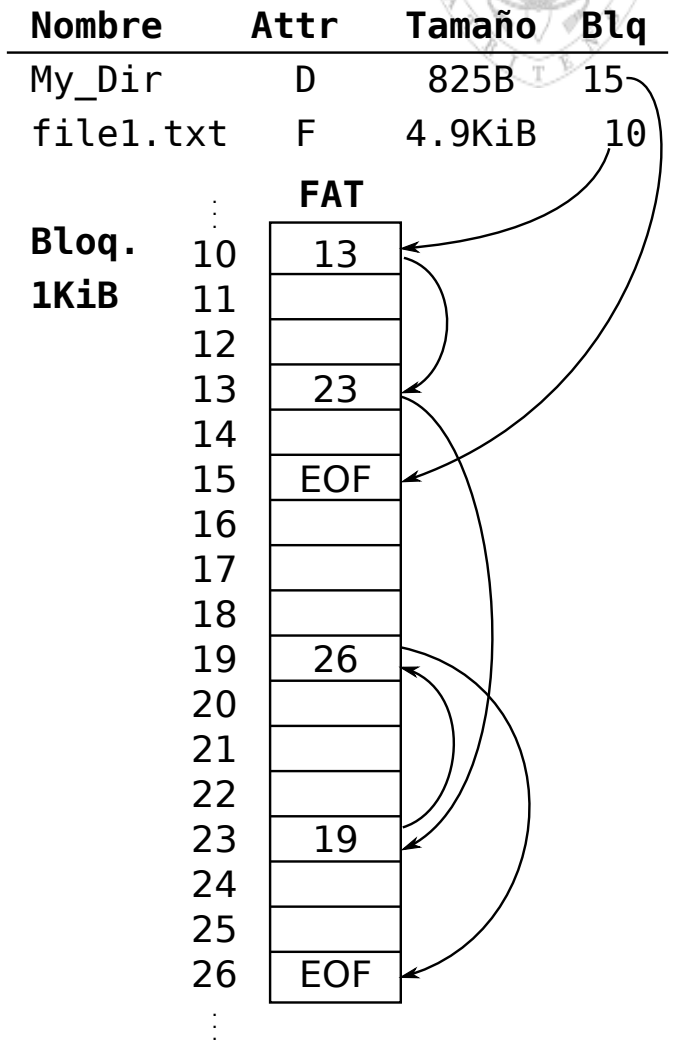
- Ventajas:
  - Acceso secuencial óptimo, permite lecturas anticipadas, fácil acceso aleatorio
- Desventajas:
  - Fragmentación externa, pre-declaración de tamaño, necesidad de compactación
- Tamaño máximo de fichero:
  - Num. Bloques dispositivo x Tam. bloque



Agenda

Bloques enlazados (FAT)

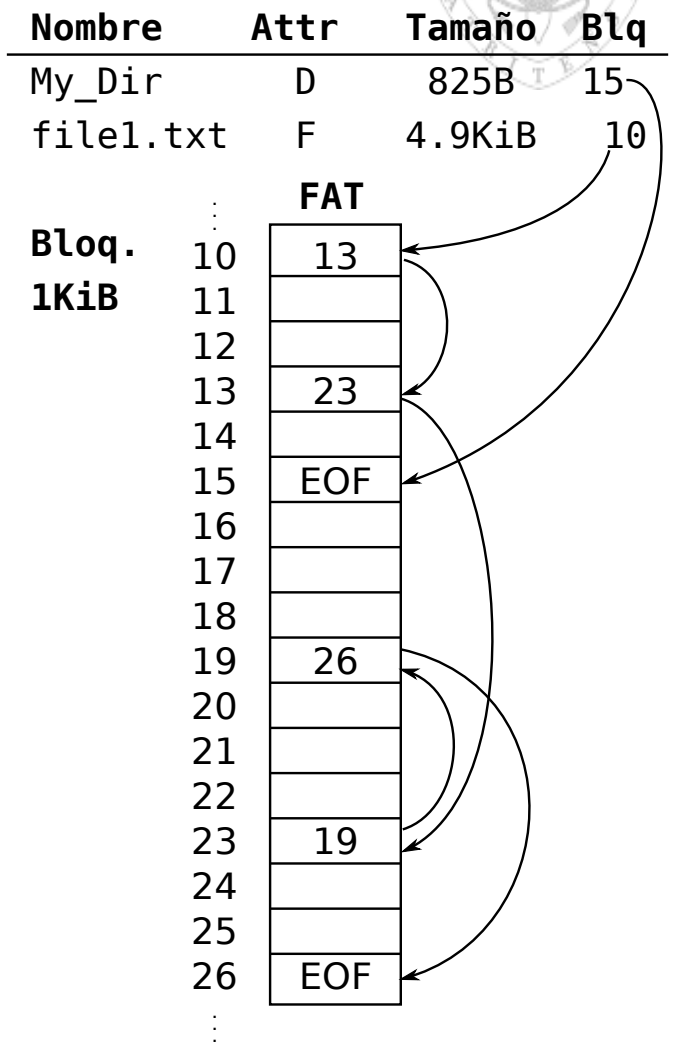
- Lista de bloques enlazados
  - En el directorio tenemos guardado el número de bloque físico que se utiliza para almancenar el primer bloque lógico del fichero
  - Se utiliza una tabla (FAT) para almacenar los enlaces a los siguientes bloques
  - Se usa una marca especial en la tabla FAT para indicar que es el último bloque del fichero (EOF)
- Distintas versiones de FAT indican el número de bits (12, 16, 32) usados para identificar un bloque



Acceso al n-ésimo byte

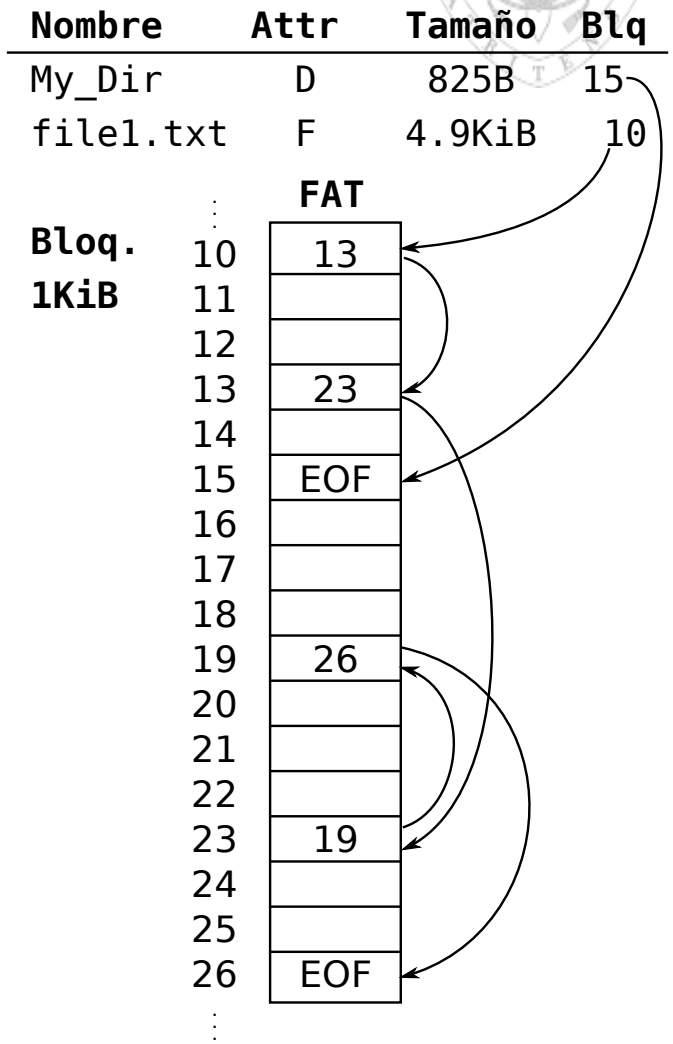
Para encontrar el n-ésimo byte del fichero tenemos que:

1. Calcular el bloque lógico en el que se encuentra el byte: $B = n/TB$
2. Acceder al directorio para obtener el id del primer bloque
3. Seguir los enlaces en la tabla FAT hasta llegar al bloque lógico B (B-1 enlaces)
4. Acceder al disco, en el bloque físico (indicado en la última entrada de la FAT consultada)



Ventajas y desventajas

- Ventajas:
  - No produce fragmentación externa
  - Asignación dinámica simple: Cualquier bloque libre puede ser añadido a la cadena
  - Acceso secuencial fácil
- Desventajas:
  - No toma en cuenta el principio de localidad, falta de contigüidad
  - Acceso aleatorio ineficiente e irregular
    - Mejora si la FAT está en memoria (sólo FAT16)
  - Es conveniente realizar compactaciones periódicas

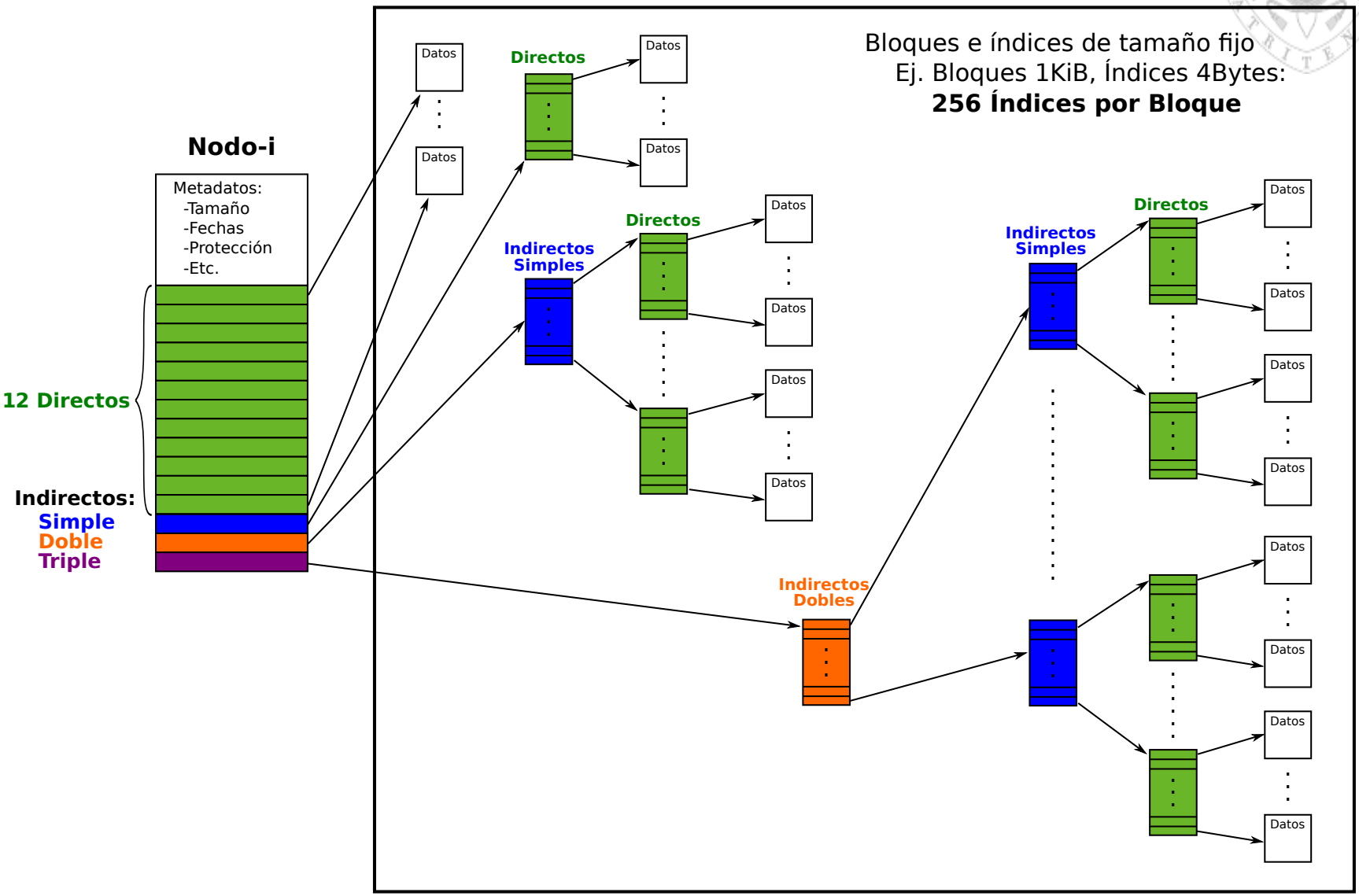


Agenda

Bloques indexados

- Cada fichero tiene una estructura llamada índice
  - Entrada $n \rightarrow$ id del bloque físico que contiene el bloque lógico n
- Puede usar varios niveles de indirección
  - Ejemplo: Unix, ext2, ext3
  - Acceso eficiente a ficheros pequeños o grandes y dispersos
  - Operaciones de borrado y truncado ineficientes en ficheros grandes
- Accesible desde una estructura asociada al fichero
  - Nodo-i en Unix
  - Representa al fichero y almacena todos los atributos del mismo
  - El Sistema de ficheros guarda una tabla con estos nodos
  - El directorio almacena el id del nodo asociado a cada fichero

Bloques indexados tipo unix



Acceso al n-ésimo byte

```
#define DP 12                                   // #Direct pointers
#define BLOCK_SIZE 1024                         // Block size
// n: byte position
block_id get_bid(int n, struct inode *p_inode) {
    int log_b = n/BLOCK_SIZE, m = BLOCK_SIZE/sizeof(block_id);
    int b_id, ind;
    block_id *p_block;
    if (log_b < DP) {                           // Direct
        b_id = p_inode->direct[log_b];
    } else if (log_b < (DP + m)) {              // Simple
        p_block = get_disk_block(p_inode->ind_simple);
        b_id = p_block[log_b - DP];
    } else if (log_b < (DP + m + m*m)) {        // Double
        log_b = log_b - (DP + m);
        p_block = get_disk_block(p_inode->ind_double);
        ind = log_b / m;
        p_block = get_disk_block(p_block[ind]);
        ind = log_b % m;
        b_id = p_block[ind];
    } ...
```

Acceso al n-ésimo byte

```
    ... else {                                          // Triple
        log_b = log_b - (DP + m + m*m);
        p_block = get_disk_block(p_inode->ind_triple);
        ind = log_b / (m*m);
        p_block = get_disk_block(p_block[ind]);
        ind = (log_b % (m*m)) / m;
        p_block = get_disk_block(p_block[ind]);
        ind = ( log_b % (m*m)) % m;
        b_id = p_block[ind];
    }

    return bid;
}
```

Tamaño máximo de fichero

- Tamaño máximo de fichero según organización:

$$T_B \times (E_d + T_B/B_{id} + (T_B/B_{id})^2 + (T_B/B_{id})^3)$$

  - T_B: tamaño de bloque
  - E_d: número de enlaces diréctos en el nodo-i
  - B_{id}: número de bytes usados para el identificador de bloque físico.
- Tamaño máximo de fichero según dirección:

$$T_B \times 2^{8B_{id}}$$

  - Es una aproximación que desprecia el espacio usado para almacenar el propio índice (usa todos los bloques para los datos del fichero)
- ¿Influye el tamaño del puntero de Lectura/Escritura?
- ¿Cómo influyen estos datos en el tamaño de la partición?

Ejemplo: ext2

Sistema de ficheros básico usado en Linux (antecesor de ext3 y ext4)

- Tamaño de bloque típico 1KiB, 4 bytes para id bloque.
- Tamaño máximo de fichero:

$$min(2^{10} \times (12 + 2^8 + 2^{16} + 2^{24}), 2^{10} \times 2^{32}) \simeq 2^{34} = 16GiB$$

- ¿Cuántos bytes se emplean para almacenar únicamente los índices de un fichero que ocupa 16GiB? (Sin contar el nodo-i)

$$2^{10} \times (1 + (1 + 2^8) + (1 + 2^8 + 2^{16})) \simeq 64MiB$$

- ¿Y para tamaño de bloque 2KiB?

Ejemplo: ext2

fichero con formato ext2

```
$ dd if=/dev/zero of=/tmp/disk.img bs=1024 count=100K
102400+0 records in
102400+0 records out
104857600 bytes (105 MB, 100 MiB) copied, 0,298336 s, 351 MB/s
$ mkfs -t ext2 -b 1024 /tmp/disk.img
mke2fs 1.42.13 (17-May-2015)
Discarding device blocks: done
Creating filesystem with 102400 1k blocks and 25688 inodes
Filesystem UUID: 1da29113-7c62-4758-8aa1-e0d2b767e3f8
Superblock backups stored on blocks:
        8193, 24577, 40961, 57345, 73729
Allocating group tables: done
Writing inode tables: done
Writing superblocks and filesystem accounting information: done
```

Ejemplo: ext2

Blocks indica el número de bloques de 512 B, independientemente del tamaño de bloque del sistema de ficheros

montaje y uso

```
$ mkdir /tmp/disk
$ sudo mount -t ext2 -o defaults,loop /tmp/disk.img /tmp/disk
$ cd /tmp/disk
$ sudo dd if=/dev/urandom of=file.bin bs=1024 count=1 seek=0
1+0 records in
1+0 records out
1024 bytes (1,0 kB, 1,0 KiB) copied, 0,0002593 s, 3,9 MB/s
$ ls -la file.bin
-rw-r--r-- 1 root root 1024 Feb 21 23:36 file.bin
$ stat file.bin
  File: 'file.bin'
  Size: 1024            Blocks: 2          IO Block: 1024   regular file
Device: 700h/1792d      Inode: 12          Links: 1
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2018-02-21 23:36:29.000000000 +0100
Modify: 2018-02-21 23:36:29.000000000 +0100
Change: 2018-02-21 23:36:29.000000000 +0100
```

Ejemplo: ext2

- ¿Y si en vez de usar dd con seek=0 ponemos seek=11?

fichero disperso (*sparse*) en ext2

```
$ rm file.bin
$ sudo dd if=/dev/urandom of=file.bin bs=1024 count=1 seek=11
1+0 records in
1+0 records out
1024 bytes (1,0 kB, 1,0 KiB) copied, 0,000197212 s, 5,2 MB/s
$ ls -la file.bin
-rw-r--r--  1 root root 12288 Feb 21 23:41 file.bin
$ stat file.bin
  File: 'file.bin'
  Size: 12288           Blocks: 2          IO Block: 1024   regular file
Device: 700h/1792d      Inode: 13          Links: 1
...
```

- ¿Qué veremos si hacemos $ hexdump -v file.bin?
- ¿Y con seek=12? ¿Y seek=268?

Ejemplo: FAT

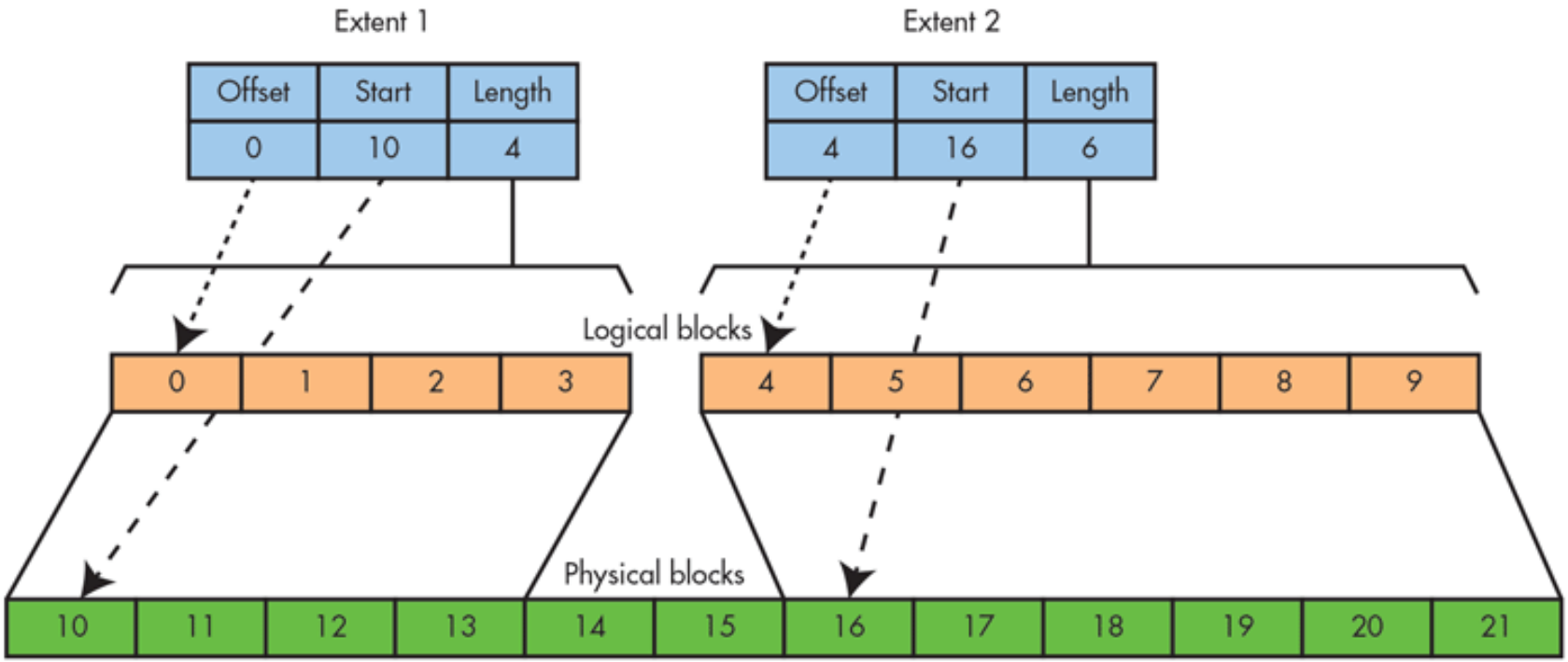
fichero disperso (*sparse*) en FAT

```
$ dd if=/dev/zero of=/tmp/disk.img bs=1024 count=10K
10240+0 records in
10240+0 records out
10485760 bytes (10 MB, 10 MiB) copied, 0,0315767 s, 332 MB/s
$ mkfs -t vfat -F32 -s 2 -S 512 /tmp/disk.img
mkfs.fat 4.2 (2021-01-31)
$ sudo mount -t vfat -o defaults,loop /tmp/disk.img /tmp/disk
$ cd disk/
$ sudo dd if=/dev/urandom of=file.bin bs=1024 count=1 seek=12
1+0 records in
1+0 records out
1024 bytes (1.0 kB, 1.0 KiB) copied, 0.000285005 s, 3.6 MB/s
$ ls -la file.bin ; du -h file.bin ; stat file.bin
-rwxr-xr-x 1 root root 13312 Dec  1 14:10 file.bin
13K        file.bin
  File: file.bin
  Size: 13312           Blocks: 26         IO Block: 1024   regular file
Device: 700h/1792d      Inode: 1050        Links: 1
```

Agenda

Extents

- Un extent es un grupo de bloques contiguos:
  - Bloque lógico del primero (offset)
  - Primer bloque físico (start)
  - Número de bloques del extent (length)



Organización del fichero

- Árbol balanceado de extents
  - Acceso homogéneo a todo el fichro
  - Ej: B Trees, B+ Trees, . . . (Video 12 min)
- El SF tiene un nodo por fichero (como nodo-i en unix)
  - Contiene atributos y la raiz del árbol
- Permite tamaños de fichero muy grandes
  - En ext4 de hasta 16 TiB con bloques de 4KiB
- Requiere algoritmos eficientes de asignación de bloques
  - La asignación a disco se retrasa lo más posible
  - Buscan grupos de bloques de disco consecutivos
  - Permiten pre-asignación de espacio (Ej: máquinas virtuales)
- Los SSFF modernos usan extents (NTFS, ext4, HFS, . . .)
  - Ejemplo: ext4 (Video 54min, ver desde 15:40 hasta 40:35)

Agenda

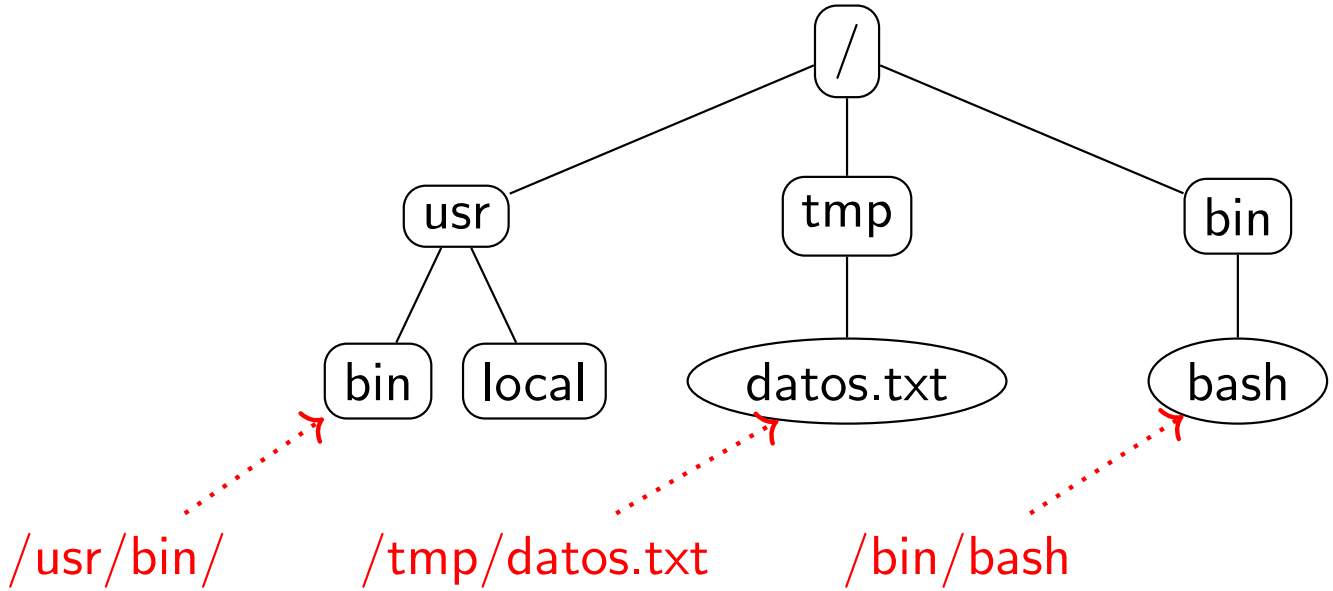
Gestión del Espacio Libre

- Mapa de bits
  - Tamaño mapa = Tamaño_de_disco / (8 * Tamaño_de_bloque)
  - Ej.: 16GiB / (8 * 1KiB) = 2MiB
- Bloques libres enlazados
- Lista de bloques libres implementada como pila o cola con parte en memoria
- Indexación de bloques libres
  - El espacio libre como fichero con indexación sobre bloques de tamaño fijo
  - Variante: Indexación con zonas de tamaño variable

Agenda

Concepto de Directorio

- Fichero especial para agrupar ficheros y directorios relacionados
- Es una relación lógica
  - El directorio contiene una *lista o relación* de nombres de ficheros, no *contiene* los ficheros en sí
  - Los ficheros tampoco se almacenan físicamente juntos en el disco
- El conjunto de directorios del SF forma una estructura de árbol
- El nombre completo o ruta de un fichero se forma con la lista de directorios que hay que atravesar para llegar al fichero desde la raiz.



Estructura de los directorios

- Los directorios se suelen implementar como ficheros con formato conocido para el SO
- Existen estructuras de directorio muy distintas. La información contenida en el directorio depende de esa estructura. Dos alternativas principales:
  - Almacenar atributos de fichero en entrada directorio.
    - FAT: Nombre, tipo, atributos, fechas, tamaño, primer bloque
  - Almacenar únicamente [nombre, identificador]. El resto de los datos del fichero en una estructura distinta
    - UNIX: i-nodo

Modelo de directorio POSIX

El modelo de directorio que el SO presenta al programador es:

- Una tabla de entradas de directorio, donde cada entrada contiene:
  - nombre de un fichero/directorio
  - un número de nodo-i
- Un descriptor de directorio
- Una serie de operaciones específicas para directorios
  - `opendir`, `readdir`, `closedir`

nombre1	num-nodo-i
nombre2	num-nodo-i
nombre3	num-nodo-i
nombre4	num-nodo-i
nombre5	num-nodo-i
nombre6	num-nodo-i
nombre7	num-nodo-i

Entradas especiales

Todos los sistemas los directorios suelen tener dos entradas especiales:

- `.` el propio directorio
- `..` el directorio padre

Estas entradas permiten:

- Nombrado de ficheros y directorios con rutas relativas
  - `/tmp/datos.txt`
  - `./datos.txt`
  - `../tmp/datos.txt`
- Recorrido del árbol de directorios

Interpretación de nombres en LINUX (1/2)

Bloques de datos del inodo 2

Nombre	i-nodo
.	2
..	2
tmp	43
home	342
info	27
⋮	⋮

Bloques de datos del inodo 342

Nombre	i-nodo
.	342
..	2
mary	430
miguel	256
elvira	78
⋮	⋮

Bloques de datos del inodo 256

Nombre	i-nodo
.	256
..	342
claves	758
texto	3265

Interpretación de nombres en LINUX (2/2)

- Interpretar `/home/miguel/claves`
  - Traer a memoria i-nodo 2 (conocido) y su[s] datos[s]
  - Buscar la cadena `home` para obtener el i-nodo 342
  - Traer a memoria i-nodo 342 (debe ser un directorio) y su[s] dato[s]
  - Buscar la cadena `miguel` para obtener el i-nodo 256
  - Traer a memoria i-nodo 256 (debe ser un directorio) y su[s] dato[s]
  - Buscar la cadena `claves` para obtener el i-nodo 758
  - Se lee el nodo-i 758 y ya se tienen los datos del fichero

- ¿Cuándo parar?
  - Se ha encontrado el i-nodo del fichero
  - No se ha encontrado y no hay más subdirectorios
  - Estamos en un directorio y no contiene la siguiente componente del nombre (por ejemplo, miguel)

API POSIX para directorios

En POSIX un directorio se maneja con el descriptor obtenido al abrirlo:

```
DIR* opendir(char *dirname);
struct dirent* readdir(DIR* dirp);
int closedir(DIR *dirp);
```

- El descriptor es una estructura `DIR`
- Su almacenamiento en memoria lo gestiona la biblioteca del sistema
- La función `readdir` devuelve la siguiente entrada de directorio como una estructura `struct dirent`
  - Almacenamiento gestionado por la biblioteca del sistema
  - Implementación dependiente del sistema
  - POSIX fija que debe tener al menos dos campos:
    - `d_name`: nombre del fichero/directorio
    - `d_ino`: número de nodo-i del fichero
  - Definida en el fichero `dirent.h`

Se recomienda consultar las páginas de manual de estas funciones.

Extensiones Linux

Linux añade un campo a la estructura `dirent`:

- `d_type`: tipo de fichero
  - DT_BLK: dispositivo orientado a bloques
  - DT_CHR: dispositivo orientado a caracteres
  - DT_DIR: directorio
  - DT_FIFO: tubería con nombre
  - DT_LNK: enlace simbólico
  - DT_REG: fichero regular
  - DT_SOCK: socket UNIX
  - DT_UNKNOWN: no se pudo identificar el tipo

  El uso de este campo ahorra llamadas adicionales a `lstat`

Ejemplo: búsqueda de fichero en ./

```
int busca(char* name)
{
  struct dirent *dp;
  DIR* dirp = opendir(".");

  if (dirp == NULL)
  return ERROR;

  while ((dp = readdir(dirp)) != NULL) {
  if (strcmp(dp->d_name, name) == 0) {
    closedir(dirp);
    return FOUND;
  }
  }
  closedir(dirp);
  return NOT_FOUND;
}
```

Otras llamadas al sistema relacionadas

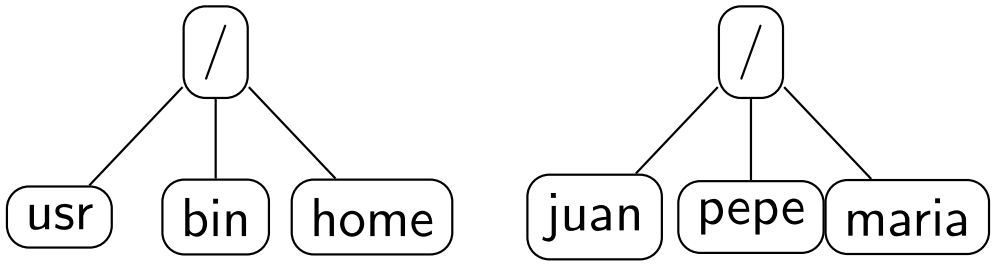
- **mkdir**: crea un directorio con un nombre y protección
- **rmdir**: borra el directorio vacío con un nombre
- **rewinddir**: sitúa el puntero de posición en la primera entrada
- **chdir**: cambia el directorio actual
- **getcwd**: obtener el directorio actual
- **rename**: cambiar el nombre de una entrada del directorio

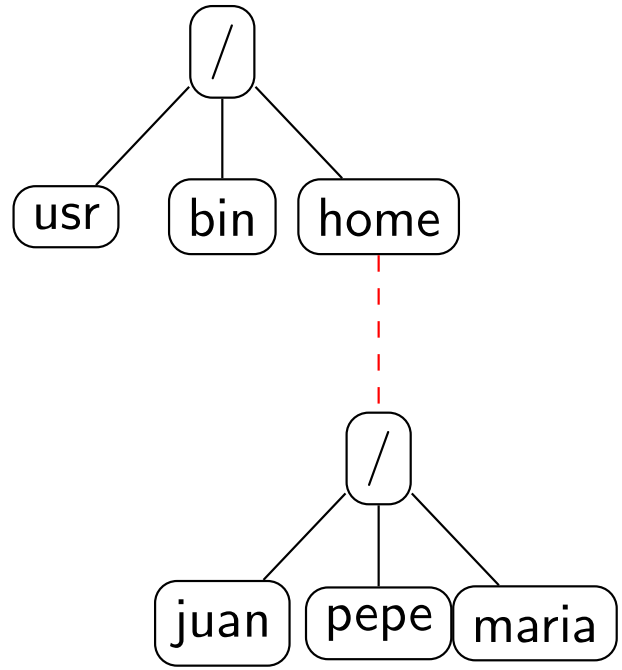
Se recomienda consultar las páginas de manual correspondientes

Árbol único vs Árbol por dispositivo

- Cada dispositivo lógico (volumen, partición,...) tiene un sistema de ficheros independiente
- VMS, Windows
  - Un árbol de directorios independiente por SF
  - La ruta completa de un fichero comienza con un identificador del volumen (Ej: C:\)
- UNIX
  - Se maneja un único árbol de directorios
  - Los directorios raiz de los SF de los volúmenes se *montan* en algún directorio de dicho árbol
  - Ventaja: imagen única del sistema, oculta el tipo de dispositivo
  - Desventaja; dificulta un poco el recorrido del árbol, en cada directorio hay que comprobar si es un punto de montaje o no

Ejemplo montado de SF





Agenda

POSIX: Enlace Duro

En POSIX existe el concepto de enlace duro, que no es más que un nombre de un fichero

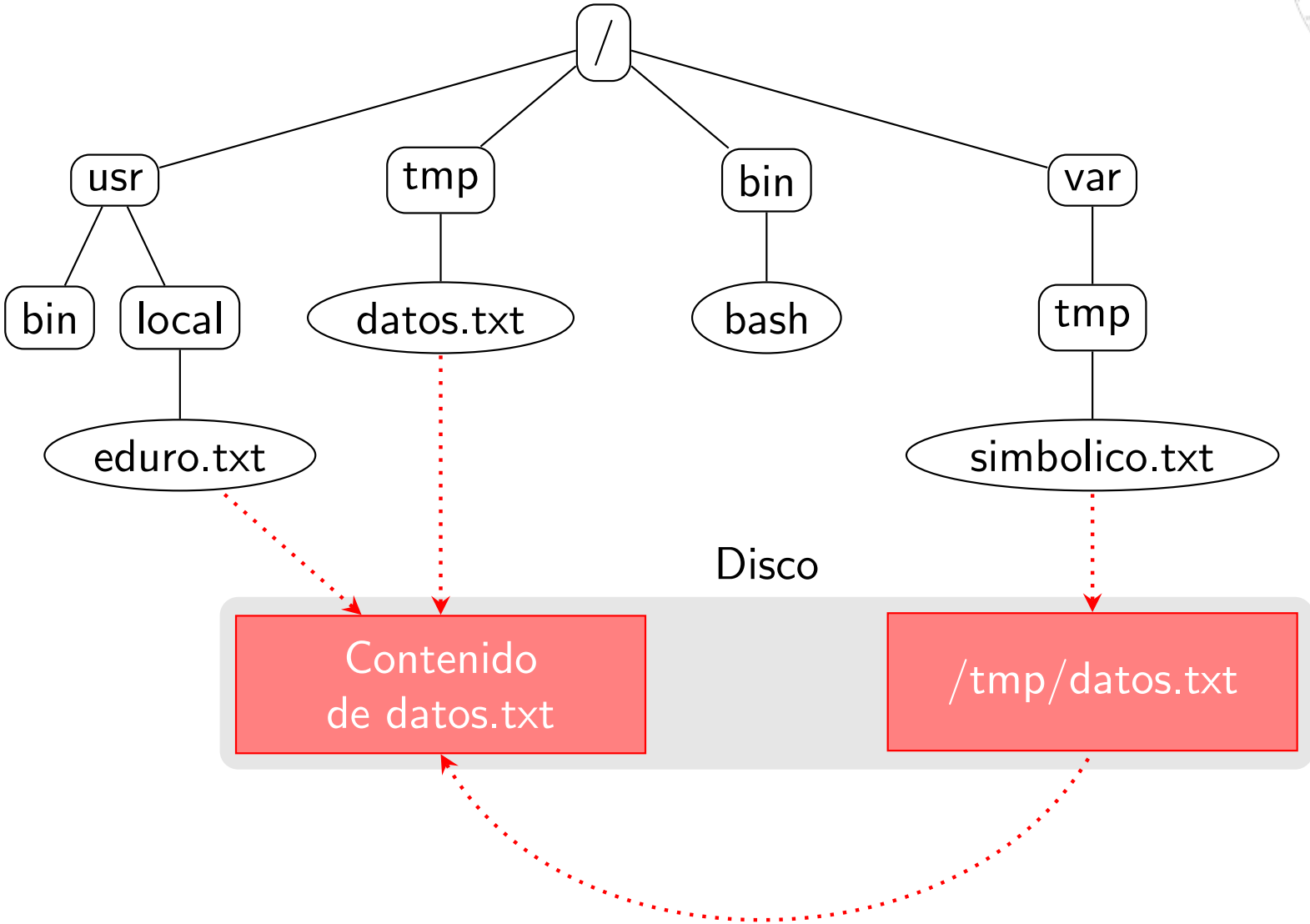
- Un fichero puede tener más de un nombre/ruta/enlace
- Siempre dentro del mismo SF
  - sin *atravesar* puntos de montaje
- Los enlaces son indistinguibles entre sí
- Para borrar el fichero hay que eliminar todos sus enlaces (`unlink`)
  - ejecutar `rm` sólo sobre uno de los nombres no borra el fichero
- No se pueden crear enlaces duros a directorios

POSIX: Enlace simbólico

En POSIX existe el concepto de enlace simbólico, que es un tipo especial de fichero cuyo contenido es el nombre del fichero al que apunta

- Un `open` sobre el enlace simbólico hace que el SO abra el fichero apuntado, no el enlace
- El nombre escrito en el enlace puede ser absoluto o relativo
- Diferencias con enlace duro:
  - Hay un fichero intermedio
  - Si se borra el fichero apuntado, el enlace permanece en el sistema, apuntando a un fichero que ya no existe
    - un `open` sobre el enlace simbólico daría error
  - Si se borra el fichero apuntado y se crea otro fichero con el mismo nombre, el enlace apuntaría al nuevo fichero
  - Borrar el enlace simbólico no afecta al fichero apuntado
  - Podemos atravesar puntos de montaje
  - Es posible crear enlaces simbólicos a directorios

Ejemplo: enlace duro vs simbólico



Ejemplo

Enlaces [vídeo]

```
$ echo "Hola" > archivo
$ ln archivo archivo_fis
$ stat archivo
$ stat archivo_fis
$ ln -s archivo archivo_sim
$ stat archivo
$ stat archivo_sim
```

- ¿Qué pasa con los nodos-i de los tres ficheros?
- ¿Cómo es el tamaño de los tres ficheros? ¿Igual? ¿Diferente?
- ¿Qué puedes decir del número de enlaces?
- ¿Qué les pasa a los otros ficheros si borras `archivo`?

Agenda

Formateado del disco

El sistema de ficheros debe almacenar una serie de estructuras en disco para poder identificar los ficheros y su contenido en el siguiente arranque:

- Se crean sobre particiones o volúmenes:
  - Una partición es una porción de un disco a la que se la dota de una identidad propia y que puede ser manipulada por el SO como una entidad lógica independiente.
  - Un volumen es un conjunto particiones de discos que pueden ser tratado por el SO como una única entidad lógica
- Una vez creadas las particiones, el SO crea en ellas las estructuras necesarias para el SSFF a utilizar
  - Se proporcionan comandos como `format` o `mkfs` al usuario.

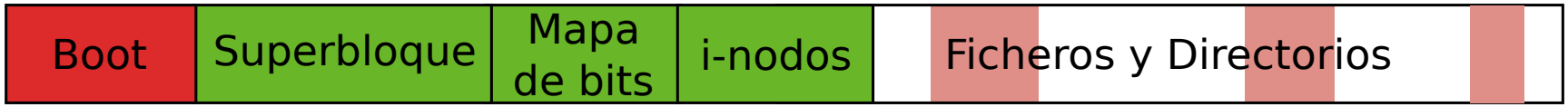
Sector y Cluster

- Sector:
  - Unidad mínima de transferencia que puede manejar el controlador de disco, 2^m Bytes (normalmente 512 Bytes)
- Cluster (o bloque del SF):
  - Agrupación lógica de sectores de disco que representa la unidad de transferencia mínima que usa el sistema de ficheros. Por lo tanto, un fichero ocupará, como mínimo, un cluster.
  - Los SSFF tienen un tamaño de bloque por defecto, pero se puede especificar otro al usar `mkfs`
  - Ayuda a optimizar la eficiencia de la entrada/salida de los dispositivos secundarios de almacenamiento.
  - El problema que introducen las agrupaciones grandes es la posibilidad de fragmentación interna.
    - El tamaño que ocupa un fichero en disco es múltiplo del tamaño del cluster.

Ejemplos: FAT y UNIX

FAT

UNIX



Agenda

SSFF en video consolas

Año	Consola	SO	Sistema Ficheros
2012	Play Station 4	Orbis	PFS
2007	Play Station 3	CellOS	PFS
2002	Play Station 2	Linux-based	PFS
2017	Xbox One X		NTFS
2013	Xbox One	Windows kernel	NTFS
2005	Xbox 360	Windows kernel	FATX
2001	Xbox	Windows kernel	FATX
2017	Nintendo Switch	µITRON IOS real-time	exFAT?
2012	Nintendo Wii U	eCOS real-time OS	?
2006	Wii Internal OS		?

PFS

- Playstation File System (PSF, psdevwiki)
- Bloques indexados
  - Bloques de 64KB normalmente
  - Nodos-i sólo con un índice directo y uno de indirección simple
- Estructura del SF:
  - Header (superbloque)
  - Tabla de nodos-i
  - Bloques de directorios
  - Bloques de datos (ficheros)

FATX o XTAF

- Basados en FAT (gamedevwiki)
  - Header (Boot). 512Bytes
    - Comienza por el magic number XTAF
  - Tabla FAT: determina el siguiente "cluster" del fichero
  - Directorio Raiz (1 cluster)
  - Bloques de datos (ficheros y directorios)
- La entrada de directorio contiene:
  - nombre, tamaño, cluster inicio, tiempos de acceso/modificación/creación
- Seguridad
  - Configuración de sectores con firma SHA1 para verificar su autenticidad
  - Dos primeros sectores del fichero *Data0000* contienen información de firma y dispositivo

NTFS

- Basado en extents
  - Usa B+-Trees
- Estructura:
  - Boot sector: información de la estructura del sistema de ficheros + el código de arranque
  - Master File Table (MFT): equivalente a nodos-i
    - Atributos y la raíz del árbol de extents
  - File System Data: datos del fichero extra no contenido en MFT
  - Mater Table Copy: datos de los ficheros