TEMA 4: SISTEMAS DE ARCHIVOS

- Una de las principales tareas del SO es **crear y administrar abstracciones del disco** del sistema.

CONSIDERACIONES GENERALES

- Todos los sistemas necesitan almacenar y recuperar información a largo plazo de manera que se cumplan los siguientes requisitos:
  - ↳ Los procesos pueden almacenar cierta cantidad de información en su espacio de direcciones, pero de manera muy limitada.
  - Debe ser posible almacenar una **cantidad muy grande** de información → si usásemos el espacio de direcciones, el tamaño estaría limitado.
  - La información almacenada debe **sobrevivir a la terminación del proceso** que la utilice → si usásemos el espacio de direcciones, al terminar el proceso se perdería la información.
  - **Varios procesos** deben poder **acceder concurrentemente** a la información → si usásemos el espacio de direcciones, sólo un proceso podría acceder a la información.

- Como los espacios de direcciones no son viables para almacenar información a largo plazo, se usan **dispositivos físicos** como discos magnéticos.
  Por ahora, entenderemos el disco como una secuencia lineal de bloques de tamaño fijo que admiten únicamente dos operaciones, leer y escribir.
  - ↳ Esto plantea varios problemas: ¿cómo se busca información? ¿cómo se sabe qué bloques están libres?...
- El ARCHIVO es una **abstracción** que realiza el SO **de los dispositivos físicos** de almacenamiento. Son **unidades lógicas de información creadas por los procesos**.
  - Los **procesos** pueden **leer** los archivos existentes y **crear** otros si es necesario.
  - La información que se almacena en los archivos debe ser **persistente**, es decir, sobrevivir a la terminación de los procesos.
  - Un archivo debe **desaparecer** solo cuando un **proceso autorizado** (por ejemplo, su dueño) lo elimina.
- Los archivos son administrados por una parte del SO denominada SISTEMA DE ARCHIVOS.
  - Desde el **punto de vista del usuario**, lo más importante del sistema de archivos es su **apariencia** (cómo se estructuran los archivos, cómo se llaman, cómo se opera con ellos, etc.).
  - Desde el **punto de vista del diseñador del SO**, lo más importante del sistema de archivos son **detalles sobre su implementación** (listas enlazadas, mapas de bits, etc.).

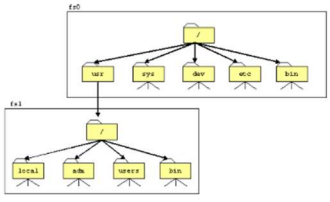
DISCO FÍSICO vs DISCO LÓGICO

- Un DISCO FÍSICO es un **dispositivo de E/S** para el **almacenamiento permanente de datos** formado por un **conjunto de bloques de tamaño fijo**.
  - Cada uno de estos bloques se identifica con un **número de bloque físico**.
- Un DISCO LÓGICO es una **abstracción del hardware** que el SO ve como una **secuencia lineal de bloques accesibles aleatoriamente**.
  - Cada uno de estos bloques se identifica con un **número de bloque lógico.**
- El manejador del disco se encarga de traducir los números de bloque lógicos a números de bloque físicos.
- Normalmente, el disco físico se divide en varias PARTICIONES contiguas independientes, de manera que cada una de ellas puede actuar como disco lógico.
  - Es responsabilidad del administrador del sistema decidir qué va a contener cada una de ellas.
  - Permiten que convivan varios SOs en un único disco físico, de manera que a cada uno de ellos se le asigna una partición.
    - La **partición activa** será aquella en la que se busca el SO en el momento del arranque de la máquina.
- Cada **sistema de archivos** se encuentra **contenido por completo en un disco lógico** y un **disco lógico** puede **contener un único sistema de archivos**.
  - ↳ Algunos discos lógicos, en vez de contener un sistema de archivos, son usados como el área de **swapping** de la memoria principal (en otros sistemas se usa una partición dedicada para esto).
- En los sistemas modernos, **varios discos físicos** o **particiones de distintos discos** pueden combinarse en un **único disco lógico** para que soporte sistemas de archivos más grandes.

MONTAJE DE SISTEMAS DE ARCHIVOS

- La jerarquía de archivos del sistema puede tener varios **subárboles** independientes, donde cada uno de ellos puede contener un sistema de archivos entero.
1. Se configura un sistema de archivos como **sistema de archivos raíz**, de manera que su directorio raíz sea el directorio raíz del sistema.
2. El resto de sistemas de archivos se adjuntan a la nueva estructura **montándolos** dentro de un directorio ya existente del árbol, el **directorio de montaje**.
3. Una vez se monta un sistema de archivos, su directorio raíz pasa a ocultar el contenido del directorio de montaje hasta que se desmonte el sistema de archivos.
- El montaje de sistemas de archivos permite **ocultar al usuario** los detalles de la organización del almacenamiento, pues el espacio de **nombres de archivos será homogéneo**, es decir, no habrá que especificar la unidad del disco como parte del nombre del archivo.
- Normalmente los sistemas de archivos que usa el sistema no están cambiando frecuentemente.
  Entonces, el núcleo usará una TABLA DE MONTAJE (normalmente en $/etc/mtab$) para identificar los sistemas de ficheros que debe montar al arrancar la máquina.
- Cada una de las líneas de esta tabla contiene la información sobre uno de los sistemas de archivos a montar: dispositivo que se monta, directorio de montaje, tipo de sistema de archivos montado, opciones del montaje.

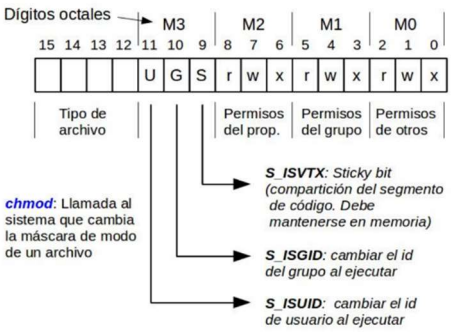
SYSCALLS PARA MONTAJE DE SISTEMAS DE ARCHIVOS



- La syscall $mount(dispositivo, directorio, flags)$ se usa para montar un sistema de archivos desde un programa.
  - Su salida es $\begin{cases} \text{en caso de éxito} \rightarrow 0. \\ \text{en caso de fracaso} \rightarrow -1. \end{cases}$
  - $dispositivo \rightarrow$ ruta de acceso del fichero del dispositivo donde se encuentra el sistema de archivos que se va a montar
  - $directorio \rightarrow$ ruta de acceso del directorio sobre el que se va a montar el sistema de archivos.
  - $flags \rightarrow$ máscara de bits para especificar las opciones de montado.
- La syscall $umount(dispositivo)$ se usa para desmontar un sistema de archivos desde un programa.
  - $dispositivo \rightarrow$ ruta de acceso del fichero del dispositivo donde se encuentra el sistema de archivos que se va a desmontar.

ARCHIVOS DESDE EL PUNTO DE VISTA DEL USUARIO

MÁSCARA DE MODO



- Cada archivo tiene asociada una MÁSCARA DE MODO de 16 bits:
  - 4 bits que indican el tipo de archivo
  - Bit $S\_ISUID \rightarrow$ si está activo, se permite cambiar el id del usuario efectivo durante la ejecución del archivo.
  - Bit $S\_ISGUID \rightarrow$ si está activo, se permite cambiar el id del grupo efectivo durante la ejecución del archivo.
  - Bit S_ISVTX (sticky bit).
  - 3 bits de permisos rwx para el propietario del archivo.
  - 3 bits de permisos rwx para el grupo al que pertenece propietario del archivo.
  - 3 bits de permisos rwx para el resto de usuarios.
- ▸ La syscall $chmod$ permite modificar la máscara de modo de un archivo.

SYSCALLS PARA OPERAR CON ARCHIVOS

- $open(ruta, flags, modo)$ → abre un archivo para lectura, escritura o ambos y devuelve su descriptor de archivo.
  - Se puede hacer que, si el archivo no existe, lo cree.
- $close(fd)$ → cierra un archivo.
- $read(fd, buffer, tamano)$ → lee datos de un archivo.
- $write(fd, buffer, tamano)$ → escribe datos en un archivo.
- $lseek(fd, desplazamiento, whence)$ → desplaza el puntero de un archivo.
- $stat(ruta, buffer)$ → obtiene información sobre un archivo.
- $chmod(ruta, modo)$ → cambia la máscara de modo de un archivo.
- $rename(ruta\_anterior, ruta\_nueva)$ → renombra un archivo.

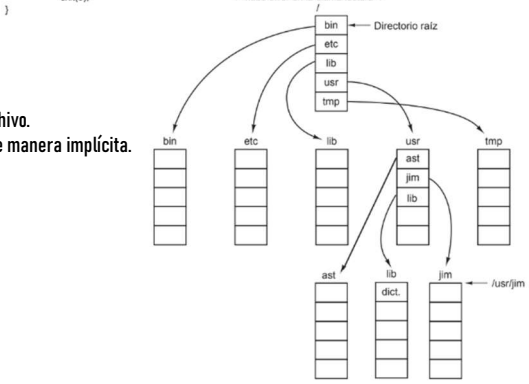
```
#include <sys/types.h>                  /* incluye los archivos de encabezado necesarios */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);       /* prototipo ANSI */

#define TAM_BUF 4096                    /* usa un tamaño de búfer de 4096 bytes */
#define MODO_SALIDA 0700                /* bits de protección para el archivo de salida */

int main(int argc, char *argv[])
{
    int ent_da, sal_da, leer_cuenta, escribir_cuenta;
    char bufer[TAM_BUF];

    if (argc != 3) exit(1);             /* error de sintaxis si argc no es 3 */

    /* Abre el archivo de entrada y crea el archivo de salida */
    ent_da = open(argv[1], O_RDONLY);   /* abre el archivo fuente */
    if (ent_da < 0) exit(2);            /* si no se puede abrir, termina */
    sal_da = creat(argv[2], MODO_SALIDA);   /* crea el archivo de destino */
    if (sal_da < 0) exit(3);            /* si no se puede crear, termina */

    /* Ciclo de copia */
    while (TRUE) {
        leer_cuenta = read(ent_da,bufer,TAM_BUF); /* lee un bloque de datos */
    if (lee_cuenta <= 0) break;         /* si llega al fin de archivo o hay un error, sale del ciclo */
        escribe_cuenta = write(sal_da,bufer,lee_cuenta); /* escribe los datos */
        if (escribe_cuenta <= 0) exit(4);   /* escribe_cuenta <= 0 es un error */
    }

    /* Cierra los archivos */
    close(ent_da);
    close(sal_da);
    if (lee_cuenta==0)                  /* no hubo error en la última lectura */
        exit(0);
    else
        exit(5);                        /* hubo error en la última lectura */
}
```

DIRECTORIOS

- Los DIRECTORIOS o carpetas son los **contenedores de los archivos**, que se usan para **agrupar archivos relacionados** de manera natural.
  - Muchas veces, los directorios serán archivos a su vez.

- El sistema de archivos se organiza en una jerarquía en forma de **árbol de directorios**.
- Como consecuencia, para acceder a un archivo se necesita especificar dónde se encuentra árbol. Hay dos formas de hacer esto:
  - Ruta absoluta → comienzan por / e indica la ruta completa desde el directorio raíz hasta el archivo.
  - Ruta relativa → no comienza por /, se coloca la ruta absoluta del directorio de trabajo actual de manera implícita.
- En cada directorio hay dos **entradas especiales**:
  - "." → se refiere al directorio de trabajo actual.
  - ".." → se refiere al directorio padre del directorio de trabajo actual.
    - En el directorio raíz, se refiere a sí mismo.

  - Syscalls para operar con directorios: $create, delete$ (sólo para directorios vacíos), $opendir, closedir, readdir, rename, link, unlink$.



IDENTIFICADORES DE USUARIO Y GRUPO

- Al **iniciar un proceso**, además del PID, se le asignan dos **identificadores** (enteros positivos) de **usuario** y otros dos de **grupo**:
  - uid → identifica al usuario responsable de la ejecución del proceso.
  - gid → identifica al grupo del usuario uid.
  - $euid$ → identifica al usuario efectivo, que será aquel { propietario de los ficheros creados. / cuyos permisos se comprueban al acceder a los ficheros de otros usuarios. / cuyos permisos se comprueban al mandar señales a otros procesos.
  - $egid$ → identifica al grupo efectivo, el del usuario $euid$.
  - Usualmente, los pares uid y $euid$, gid y $egid$ van a tener el mismo valor.
- Las syscalls $getuid()$, $geteuid()$, $getgid()$ y $getegid()$ permiten determinar los valores de los identificadores.
- La syscall $setuid(id)$ permite asignar el valor id al uid y $euid$ del proceso que la invoca.
  - Su salida es { en caso de éxito → 0. / en caso de fracaso → −1.
  - id → identificador de usuario.
  - Su comportamiento depende de la situación. Se distinguen los siguientes casos:
    - El $euid$ del proceso que efectúa la llamada es el del superusuario → se establece uid y $euid$ a id.
    - El $euid$ del proceso que efectúa la llamada **no** es el del superusuario → se establece $euid$ a id si se cumple cualquiera de estas condiciones:
      - id coincide con el uid del proceso.
      - El programa tiene el bit S_ISUID activado y id es el identificador del dueño del programa.
- La syscall $setguid(id)$ permite asignar el valor id al gid y $egid$ del proceso que invoca la llamada de manera similar que $setuid$.

```
#include <fcntl.h>

main(){
int x, y, fd1,fd2;

x=getuid(); y=geteuid();
printf("\nUID=%d, EUID=%d\n", x, y);

fd1=open("fichero1.txt", O_RDONLY);
fd2=open("fichero2.txt", O_RDONLY);
printf("fd1=%d, fd2=%d\n", fd1, fd2);

setuid(x);
printf("UID=%d, EUID=%d\n",getuid(),geteuid());

fd1=open("fichero1.txt", O_RDONLY);
fd2=open("fichero2.txt", O_RDONLY);
printf("fd1=%d, fd2=%d\n", fd1, fd2);

setuid(y);
printf("UID=%d, EUID=%d\n",getuid(),geteuid());}
```

- El programa pertenece a USUARIO1
  - Máscara *rwx rwx rwx*
  - S_ISUID=1
- fichero1.txt pertenece a USUARIO1:
  Máscara *rw* – – – – – –
- fichero2.txt pertenece a USUARIO2:
  Máscara *rw* – – – – – –
- uid de USUARIO1 = 501
  uid de USUARIO2 = 503

1. USUARIO 1 ejecuta el programa

```
[1] UID=501, EUID=501       [2] fd1=3, fd2=-1
[3] UID=501, EUID=501       [4] fd1=4, fd2=-1
[5] UID=501, EUID=501
```

2. USUARIO 2 ejecuta el programa

```
[1] UID=503, EUID=501       [2] fd1=3, fd2=-1
[3] UID=503, EUID=503       [4] fd1=-1, fd2=4
[5] UID=503, EUID=501
```

3. USUARIO 2 ejecuta el programa y su bit S_ISUID=0

```
[1] UID=503, EUID=503       [2] fd1=-1, fd2=3
[3] UID=503, EUID=503       [4] fd1=-1, fd2=4
[5] UID=503, EUID=503
```

IMPLEMENTACIÓN DEL SISTEMA DE ARCHIVOS

- A continuación se presentan distintas opciones de **distribución de los bloques de un archivo** para poder mantener un registro acerca de qué bloques del disco pertenecen a qué archivo.

ASIGNACIÓN DE LISTA ENLAZADA

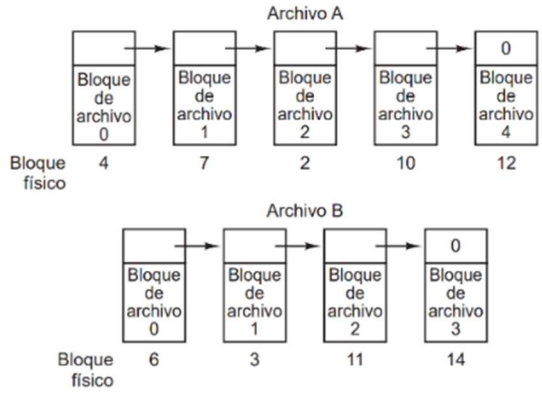
- ○ Se mantiene cada archivo como una **lista enlazada de bloques del disco**, de manera que la primera palabra de cada bloque se usa como apuntador al siguiente.
  - ↳ El último bloque del archivo tiene en el apuntador un valor no válido como 0 para indicar el fin de la cadena.
- ✓ A diferencia de lo que sucede en la asignación contigua, permite usar todos los bloques del disco, por lo que **no** se pierde espacio debido a la **fragmentación externa**.
- ✓ Para **localizar un archivo** sólo hace falta almacenar **la dirección de disco de su primer bloque**, el resto se pueden encontrar a partir de ella.
- × El **acceso aleatorio es muy lento**, pues para llegar al bloque k de un archivo, se tendrá que comenzar desde el primero y leer los $k-1$ siguientes desde el disco.
- × La cantidad de datos que se pueden almacenar en un **bloque ya no es potencia de 2**, pues el apuntador ocupa unos cuantos bytes de cada bloque.
  - ↳ En muchos sistemas, el tamaño de página es múltiplo del tamaño del bloque, por lo que para leer una sola página del disco habría que acceder a 2 bloques.

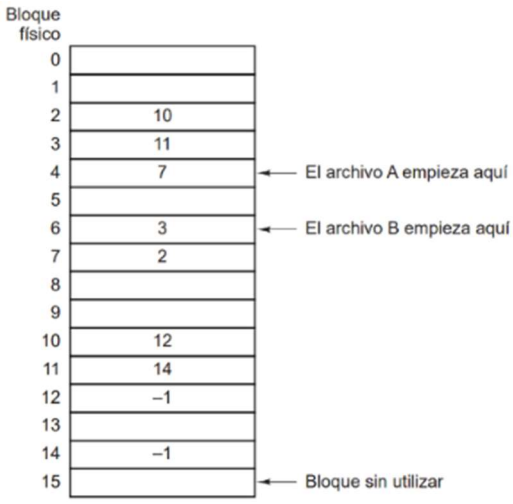
ASIGNACIÓN DE LISTA ENLAZADA USANDO UNA TABLA FAT

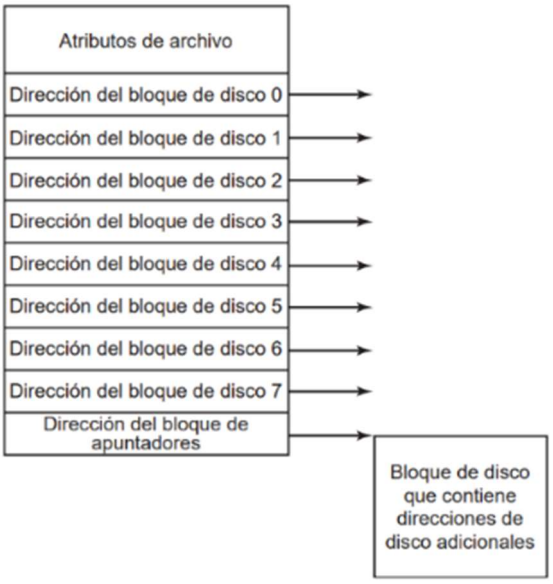
- ▸ Versión de la lista enlazada que soluciona sus problemas.
- ○ Se mantiene en memoria en todo momento una **tabla FAT** (tabla de asignación de archivos) con **tantas entradas como bloques tenga el disco** de manera que en cada una de ellas se almacena el apuntador al siguiente bloque del archivo.
  - ↳ La última entrada del archivo tiene en el apuntador un valor no válido como -1 para indicar el fin de la cadena.
- ✓ El **acceso aleatorio es más rápido** porque, aunque haya que seguir la cadena hasta llegar al bloque deseado, ahora los punteros están en memoria principal, que es más rápida.
- ✓ La cantidad de datos que se pueden almacenar en un **bloque es potencia de 2**.
- • Toda la tabla tiene que estar en memoria todo el tiempo para que funcione, y su tamaño es proporcional al del disco, por lo que la idea **escala muy mal con el tamaño del disco**.
- ▷ Se usa en Windows.

ASIGNACIÓN CON NODOS-i

- ○ A cada archivo se le asigna una estructura de datos conocida como **NODO-i**, que contiene sus **atributos** y las **direcciones de disco de los bloques** donde está contenido.
- • La cantidad de direcciones de bloques de disco que se puede almacenar en cada nodo-i está limitada, por lo que si se tiene un **archivo que ocupa más bloques que el máximo** de identificadores del nodo-i, se reserva **el último** de estos **identificadores** para apuntar a un **bloque en disco que contenga más direcciones** de bloques del archivo.
- ✓ El **nodo-i** de un archivo sólo tiene que estar **en memoria mientras está abierto el archivo** en cuestión.
- ✓ El conjunto de nodos-i presentes en memoria **ocupa mucho menos espacio que una tabla FAT** pues su tamaño es proporcional al número máximo de archivos que pueden estar abiertos a la vez, y no al tamaño del disco.
- × En los **archivos pequeños se desperdicia algo de memoria** pues se reserva espacio para su nodo-i entero, aunque no vayan a usar la mayoría de los campos para direcciones.
- ▷ Se usa en UNIX.







ADMINISTRACIÓN Y OPTIMIZACIÓN DEL SISTEMA DE ARCHIVOS

TAMAÑO DEL BLOQUE

- Existen dos maneras de almacenar un archivo de n bytes en el disco:
  - Asignarle n bytes consecutivos → provoca fragmentación externa y, si crece, habría que moverlo a otro lugar del disco, lo cual es muy costoso.
  - Dividir el archivo en varios bloques de un determinado tamaño no necesariamente consecutivos.
- Por tanto, casi todos los sistemas escogen la segunda alternativa. Para poder implementarla, hay que decidir cuál será el tamaño de estos bloques.
  - Tendría sentido escoger un tamaño múltiplo del de sector, pista o cilindro del disco, pero estos valores dependen del dispositivo, lo cual es inconveniente.
  - Si el tamaño de bloque es **demasiado grande** → el último bloque de cada archivo quedará mayormente vacío, aparece **fragmentación interna**.
  - Si el tamaño de bloque es **demasiado pequeño** → la mayoría de archivos ocuparán muchos bloques, así que para leerlos se necesitarán **muchas búsquedas y retrasos rotacionales**.
- Entonces, cuanto más **grande** sea se **desperdicia más espacio** y cuanto más **pequeño** sea se **desperdicia más tiempo**.
  - Hay que llegar a una **solución de compromiso** entre ambos factores.

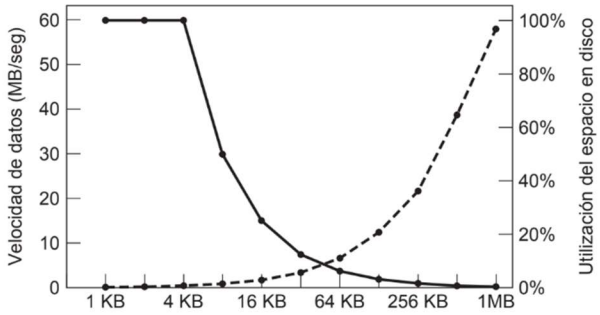


Figura 4-21. La curva sólida (escala del lado izquierdo) da la velocidad de datos del disco. La curva punteada (escala del lado derecho) da la eficiencia del espacio de disco. Todos los archivos son de 4 KB.

REGISTRO DE BLOQUES LIBRES

Existen dos métodos utilizados ampliamente para llevar registro de los bloques libres del disco:

- **Lista enlazada** de bloques libres del disco, en la que cada entrada es el identificador de un bloque de disco que no está ocupado.
  - Las entradas **no** están **ordenadas**.
  - Se divide en **secciones** del **tamaño de un bloque**, de manera que la última entrada de cada una de ellas es un apuntador a la siguiente.
    - Sólo la primera sección de la lista estará en memoria principal, para más rápido acceso. El resto se almacenan en el disco.
    - Al crear un archivo, se tomarán los bloques libres necesarios de la sección que está en memoria. Si no hay suficientes, se lee una nueva desde el disco.
    - Al borrar un archivo, se agregarán los bloques que quedaron libres a la sección que está en memoria. Si no tiene espacio suficiente, se escribirán en una del disco.
  - Su **tamaño cambia** con el tiempo en función de la cantidad de bloques libres que haya en el disco en un momento dado.
- Mapa de bits, vector de tantos bits como bloques hay en el disco de manera que en cada bit se almacena un 0 o un 1 en función de si el bloque correspondiente está ocupado o no.
  - Se almacena en su totalidad en memoria principal.
  - Su **tamaño es constante**, ocupa tantos bits como bloques tenga el disco.
- Por lo general, el **mapa de bits ocupa menos espacio** que la lista. Sólo si el disco está casi lleno (es decir, hay pocos bloques libes) la lista ocupará menos espacio.
- Para que la lista sea más corta, las entradas pueden representar **series de bloques libres** consecutivos en lugar de bloques individuales.
  - En cada entrada se almacenaría el identificador del primer bloque de la serie y el número de bloques libres consecutivos que la forman.
  - Si el disco está muy fragmentado, esta alternativa sería menos eficiente en términos de espacio, pues las entradas serían más anchas y no se reduciría mucho su cantidad.