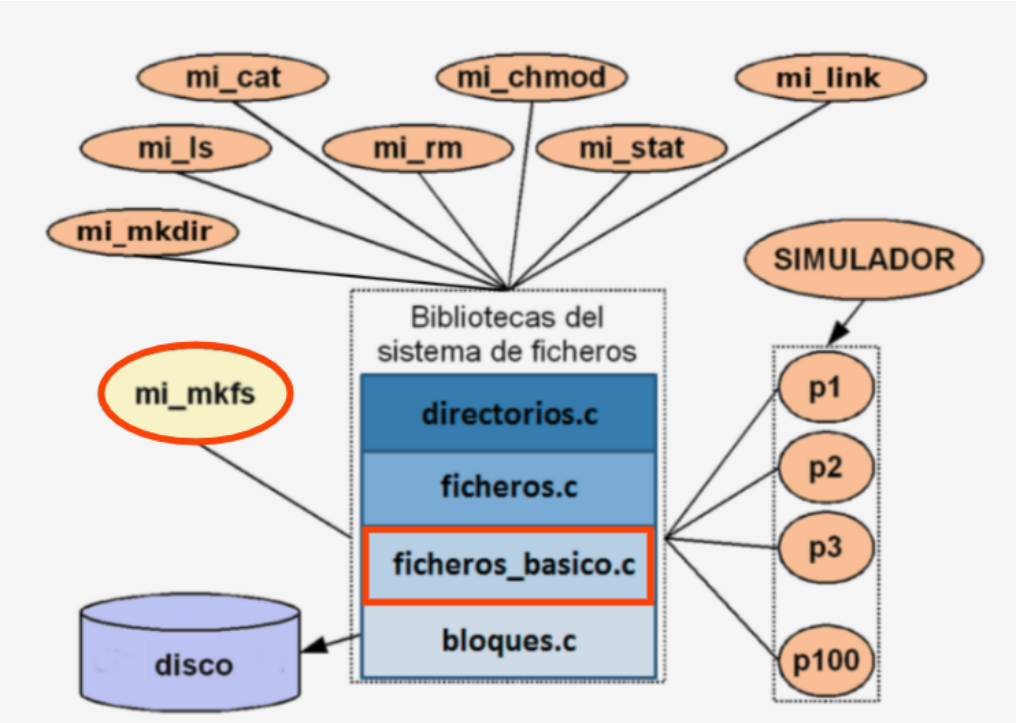
Nivel 3: ficheros_basico.c {escribir_bit(), leer_bit(), reservar_bloque(), liberar_bloque(), escribir_inodo(), leer_inodo(), reservar_inodo()} y **mi_mkfs.c**

Continuaremos con la definición de funciones básicas de gestión de ficheros (en **ficheros_basico.c**, y declaradas en su cabecera **ficheros_basico.h**), y actualizando **mi_mkfs.c** para afinar más el formateo de nuestro sistema de ficheros. En particular definiremos **funciones para operar con el mapa de bits** (*escribir_bit()*, *leer_bit()*), otras para reservar y liberar bloques (*reservar_bloque()*, *liberar_bloque()*) y otras para **gestionar inodos** (*escribir_inodo()*, *leer_inodo()*, *reservar_inodo()*)[1].



ficheros_basico.c

Hay que programar funciones básicas[2] de E/S para los bits del mapa de bits (en adelante MB):

[1] Dejaremos para más adelante la función liberar_inodo(), dada su complejidad, y porque para liberar un inodo, tenemos que liberar sus bloques y antes tenemos que aprender a asignárselos mediante la función traducir_bloque_inodo() del nivel 4.

[2] Los parámetros indicados son orientativos. Si necesitáis adaptarlos lo hacéis, siempre y cuando las funciones hagan lo que se requiere. Igualmente podéis utilizar funciones auxiliares cuando lo consideréis oportuno.

1) int escribir_bit(unsigned int nbloque, unsigned int bit);

Esta función escribe el valor indicado por el parámetro *bit*: 0 (libre) ó 1 (ocupado) en un determinado bit del MB que representa el bloque *nbloque*. La utilizaremos cada vez que necesitemos reservar o liberar un bloque.

Dado un nº de bloque **físico**, *nbloque*, del que queremos indicar si está libre o no, primeramente deberemos averiguar donde se ubica su bit correspondiente en el MB, y luego de manera absoluta en el dispositivo[3] (nº de bloque físico) para grabarlo cuando le hayamos dado el valor deseado.

Veámoslo paso a paso:

- Leer el superbloque para obtener la localización del MB.
- Calculamos la posición del byte **en el MB**, *posbyte*, que contiene el bit que representa el *nbloque* y luego la posición del bit dentro de ese byte, *posbit*:

```
posbyte = nbloque / 8

posbit = nbloque % 8
```

- Hemos de determinar luego en qué bloque del MB, *nbloqueMB*, se halla ese bit para leerlo:

```
nbloqueMB = posbyte / BLOCKSIZE
```

- Y finalmente hemos de obtener en qué posición absoluta del dispositivo virtual se encuentra ese bloque, *nbloqueabs*, donde leer/escribir el bit:

```
nbloqueabs = SB.posPrimerBloqueMB + nbloqueMB
```

Veamos un ejemplo:

- *nbloque* = 40.003 (es el bloque **físico** que queremos indicar si está libre u ocupado, lo recibimos como parámetro)
- *posbyte* = *nbloque* / 8 = 5.000 (dividimos entre 8 porque los bits que representan los bloques físicos se agrupan de 8 en 8 para formar bytes, se trata de una división entera). Esto significa que el byte 5.000 del MB contiene el bit que representa el *nbloque* 40.003.
- *posbit* = *nbloque* % 8 = 40.003 % 8 = 3 (sería el resto de la división). Esto significa que el bit 3 (teniendo en cuenta que se empieza a contar desde el 0) del byte 5.000 del MB es el que representa el *nbloque* 40.003.
- *nbloqueMB* = *posbyte* / *BLOCKSIZE* = 5.000 / 1.024 = 4. Esto significa que el bloque 4 del MB, contando desde el 0 **de forma relativa al MB**, contiene el byte 5.000 que a su vez contiene el bit 3 que representa al *nbloque* 40.003.
- *nbloqueabs* = *SB.posPrimerBloqueMB* + *nbloqueMB* = 1 + 4 = 5, es la **posición absoluta en el dispositivo** donde se halla *nbloqueMB*, y la que emplearemos para realizar el *bwrite()* o *bread()*.

[3] Recordemos que todas las operaciones de E/S con el dispositivo las hacemos por **bloques**.

Ahora que ya tenemos ubicado el bit en el dispositivo, leemos con *bread()* el bloque físico que lo contiene y cargamos el contenido en un *buffer* *unsigned char bufferMB[BLOCKSIZE]*, en el que tendremos que modificar el bit deseado, pero preservando el valor de los demás bits del bloque.
Veámoslo paso a paso:

- Recordemos que *posbyte* era el byte del bloque físico que contenía el bit del MB que representa el bloque determinado, y ese byte ahora lo tenemos contenido en memoria en *bufferMB*, que ocupa 1 bloque, así que necesitamos realizar la operación **módulo** con el tamaño de bloque para localizar su posición en ese array, y así quedará **dentro del rango** de ese tamaño:

```
posbyte = posbyte % BLOCKSIZE
```

> En el ejemplo anterior *posbyte* = 5.000. Para que nos sirva de índice en el *buffer* de tamaño 1024, hay que realizar el **módulo** con ese tamaño para obtener un valor dentro del rango [0, 1024], o sea que *posbyte* pasará a valer 5.000 % 1.024 = 904 y ese será el índice del array *bufferMB*.

- Ahora que ya tenemos en memoria el byte, *bufferMB[posbyte]*, podemos poner a 1 o a 0 el bit correspondiente. Para ello, primeramente, utilizaremos una máscara y realizaremos un desplazamiento de bits (tantos como indique el valor *posbit*) a la derecha:

```
unsigned char mascara = 128    // 10000000
mascara >>= posbit             // desplazamiento de bits a la derecha
```

- Para poner un bit a 1:

```
bufferMB[posbyte] | = mascara   //  operador OR para bits
```

- Para poner un bit a 0:

```
bufferMB[posbyte] &= ~mascara  // operadores AND y NOT para bits
```

> Veamos un ejemplo para *posbit*=3:
>
> - **mascara**: 10000000
> - **Desplazando** el 1er bit de la máscara a la derecha 3 posiciones ⇒ mascara: 00010000
> - Para poner el bit a 1:
>
> Si hacemos el **OR binario** de la máscara con el byte del MB, obtendremos un 1 en la posición=3 y preservaremos el valor del resto:
>
> 00010000 | xxx0xxxx = xxx1xxxx
> 00010000 | xxx1xxxx = xxx1xxxx

- Para poner el bit a 0:
  - Hacemos el **NOT binario de la máscara desplazada** ⇒ máscara: 11101111
  - Si hacemos el **AND binario de la máscara con el byte del MB**, obtendremos un 0 en la posición=3 y preservaremos el valor del resto:

    11101111 & xxx0xxxx = xxx0xxxx
    11101111 & xxx1xxxx = xxx0xxxx

Por último escribimos ese buffer del MB en el dispositivo virtual con ***bwrite()*** en la posición que habíamos calculado anteriormente, *nbloqueabs*.

2) char leer_bit(unsigned int nbloque);

Lee un determinado bit del MB y devuelve el valor del bit leído.

Se procede igual que en la función anterior para obtener el byte del dispositivo que contiene el bit deseado y el bloque físico absoluto que lo contiene, pero en vez de escribir el bit, lo lee utilizando la máscara y desplazamientos de bits a la derecha, como se indica a continuación:

```
unsigned char mascara = 128; // 10000000
mascara >>= posbit;          // desplazamiento de bits a la derecha
mascara &= bufferMB[posbyte]; // operador AND para bits
mascara >>= (7 - posbit);    // desplazamiento de bits a la derecha
```

Veamos un ejemplo para *posbit*=3:

- **mascara**: 10000000
- **Desplazando** el 1er bit de la máscara a la derecha 3 ⇒ mascara: 00010000
- Si hacemos el **AND binario de la máscara con el byte del MB**, obtenemos el bit de la posición=3 y el resto queda a 0:

  00010000 & xxx0xxxx = 000**0**0000
  00010000 & xxx1xxxx = 000**1**0000

- En el byte resultado obtenido hacemos un **desplazamiento de 7 - *posbit* posiciones a la derecha**, o sea de 4 y así nos queda:
  0000000**0** si originariamente había un 0 en el MB // 0 en decimal
  0000000**1** si originariamente había un 1 en el MB // 1 en decimal

---

En este nivel también hay que programar funciones básicas para reservar y liberar bloques:

3) int reservar_bloque();

Encuentra el primer bloque libre, consultando el MB (primer bit a 0), lo ocupa (poniendo el correspondiente bit a 1 con la ayuda de la función *escribir_bit()*) y devuelve su posición.

Veámoslo paso a paso:

- Comprobamos la variable del superbloque que nos indica si quedan bloques libres.
- Si aún quedan, hemos de localizar el 1er bloque libre del dispositivo virtual consultando cuál es el primer bit a 0 en el MB:

  - (1) Primero localizamos la posición **en el dispositivo virtual**, del primer **bloque** del MB que tenga algún bit a 0, *nbloqueabs* y lo leemos:

    recorremos los bloques del MB (iterando con *nbloqueabs*) y los iremos cargando en *bufferMB*

    ```
    bread(nbloqueabs, bufferMB)
    ```

    hasta encontrar uno que tenga algún 0.

    Para ello utilizaremos un buffer auxiliar, *bufferAux*, inicializado con sus bits a 1s (255 en decimal):

    ```
    memset(bufferAux, 255, BLOCKSIZE); // llenamos el buffer auxiliar con 1s
    ```

    y comparamos cada bloque leído del MB, *bufferMB*, con ese buffer auxiliar, utilizando la función *memcmp()*.

  - (2) Luego localizamos qué **byte** dentro de ese bloque tiene algún 0:

    Cuando salgamos de la iteración anterior, en *bufferMB* estará el bloque que contiene al menos un 0 y buscamos en ese bloque, procedente del MB, la posición del primer byte, *posbyte*, que tenga algún 0 (podemos hacerlo recorriendo los bytes de ese bloque y comparando cada byte con 255).

  - (3) Finalmente localizamos el primer **bit** dentro de ese byte que vale 0:

    Para ello buscamos en ese byte, *bufferMB[posbyte]*, en qué posición, *posbit*, está el 0, empezando por la izquierda:

    ```
    unsigned char mascara = 128; // 10000000
    posbit = 0;
    // encontrar el primer bit a 0 en ese byte
    while (bufferMB[posbyte] & mascara) { // operador AND para bits
       bufferMB[posbyte] <<= 1;       // desplazamiento de bits a la izquierda
       posbit++;
    }
    ```

Veamos un ejemplo para un byte con valor 235 (11101011 en binario):

- **mascara**: 10000000, *posbit* = 0
- Iteramos un **AND binario del byte del MB con la máscara**, incrementamos el **contador** y **desplazamos un bit a la izquierd**a:
  11101011 & 10000000 = 10000000, *posbit* = 1
  11010110 & 10000000 = 10000000, *posbit* = 2
  10101100 & 10000000 = 10000000, *posbit* = 3
  01011000 & 10000000 = 00000000 //fin

- Para determinar cuál es finalmente el nº de bloque físico (nbloque) que podemos reservar (posición absoluta del dispositivo), necesitaremos efectuar el siguiente cálculo:

```
nbloque = ((nbloqueabs - SB.posPrimerBloqueMB) * BLOCKSIZE + posbyte) * 8 + posbit;
```

Veamos el ejemplo donde *nbloqueabs*=5, *posbyte*=904 (relativizado al tamaño de bloque), *posbit*=3:

nbloque = ((5 -1)*1024+904)*8+3=40.003

- Utilizamos la función *escribit_bit()* pasándole como parámetro ese nº de bloque y un 1 para indicar que el bloque está reservado.
- Decrementamos la **cantidad de bloques libres** en el campo correspondiente del superbloque, y salvamos el superbloque.
- Limpiamos ese bloque en la zona de datos, grabando un buffer de 0s en la posición del nbloque del dispositivo, por si había basura (podría tratarse de un bloque reutilizado por el sistema de ficheros).
- Devolvemos el nº de bloque que hemos reservado, *nbloque*.

4) int liberar_bloque(unsigned int nbloque);

Libera un bloque determinado (con la ayuda de la función *escribir_bit()*).

Veámoslo paso a paso:

- Ponemos a 0 el bit del MB correspondiente al bloque *nbloque* (lo recibimos como parámetro) mediante la función *escribir_bit()*.
- Incrementamos la **cantidad de bloques libres** en el superbloque. No es necesario limpiar el bloque en la zona de datos; se queda basura pero se interpreta como espacio libre. Salvamos el superbloque.
- Devolvemos el nº de bloque liberado, *nbloque*.

Hay que programar también funciones básicas para escribir y leer inodos:

5) int escribir_inodo(unsigned int ninodo, struct inodo *inodo);

Escribe el contenido de una variable de tipo *struct inodo*, pasada por referencia, en un determinado inodo del array de inodos, *inodos*.

Observación: como la escritura se hace por bloques, hay que preservar el valor de los demás inodos del bloque.

Veámoslo paso a paso:

- Leemos el superbloque para obtener la localización del array de inodos.
- Obtenemos el nº de bloque del array de inodos que tiene el inodo solicitado.
- Lo leemos de su posición absoluta del dispositivo empleando como buffer de lectura un array de inodos, del tamaño de la cantidad de inodos que caben en un bloque: *struct inodo inodos[BLOCKSIZE/INODOSIZE]*.
- Una vez que tenemos el bloque en memoria escribimos el contenido del inodo, pasado por referencia, en el lugar correspondiente del array: *ninodo*%(*BLOCKSIZE*/*INODOSIZE*).
- El bloque modificado lo escribimos en el dispositivo virtual utilizando la función *bwrite()*.
- Si ha ido todo bien devolvemos 0.

6) int leer_inodo(unsigned int ninodo, struct inodo *inodo);

Lee un determinado inodo del array de inodos para volcarlo en una variable de tipo *struct inodo* pasada por referencia.

Veámoslo paso a paso:

- Leemos el superbloque para obtener la localización del array de inodos.
- Obtenemos el nº de bloque del array de inodos que tiene el inodo solicitado.
- Lo leemos de su posición absoluta del dispositivo empleando como buffer de lectura un array de inodos, del tamaño de la cantidad de inodos que caben en un bloque: *struct inodo inodos[BLOCKSIZE/INODOSIZE]*.
- El inodo solicitado está en la posición *ninodo*%(*BLOCKSIZE*/*INODOSIZE*) del *buffer*
- Si ha ido todo bien devolvemos 0.

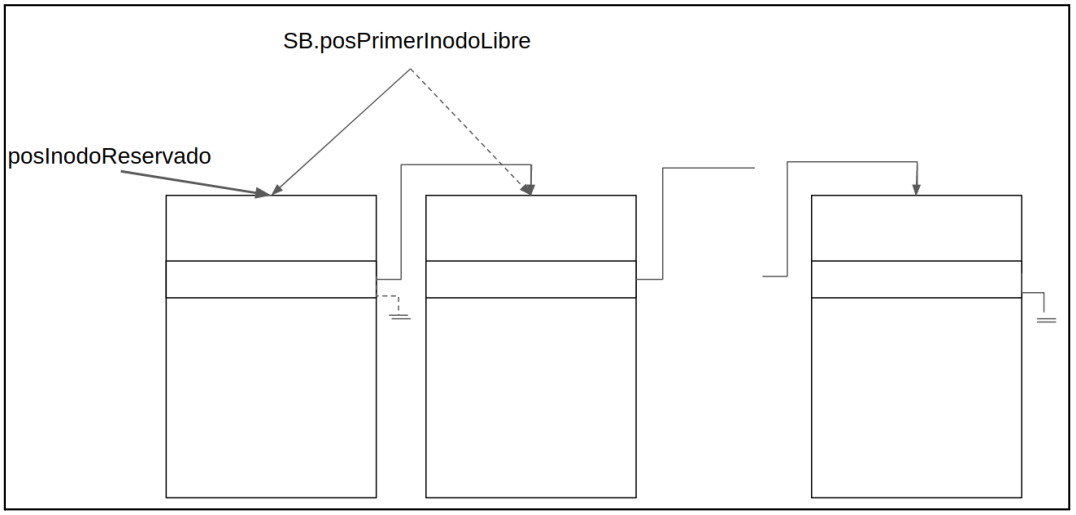
Finalmente nos queda programar funciones básicas para reservar y liberar inodos (ésta última la dejaremos para el Nivel 6 que ya hayamos podido escribir en bloques de los inodos):

7) int reservar_inodo(unsigned char tipo, unsigned char permisos);

Encuentra el primer inodo libre (dato almacenado en el superbloque), lo reserva (con la ayuda de la función *escribir_inodo()*), **devuelve su número** y actualiza la lista enlazada de inodos libres.

Veámoslo paso a paso:

- Comprobar si hay inodos libres y si no hay inodos libres indicar error y salir.
- Primeramente **actualizar la lista enlazada de inodos libres** de tal manera que el superbloque apunte al siguiente de la lista. Tendremos la precaución de guardar en una variable auxiliar *posInodoReservado* cual era el primer inodo libre, ya que éste es el que hemos de devolver.



Lista enlazada de nodos libres antes y después de reservar un inodo

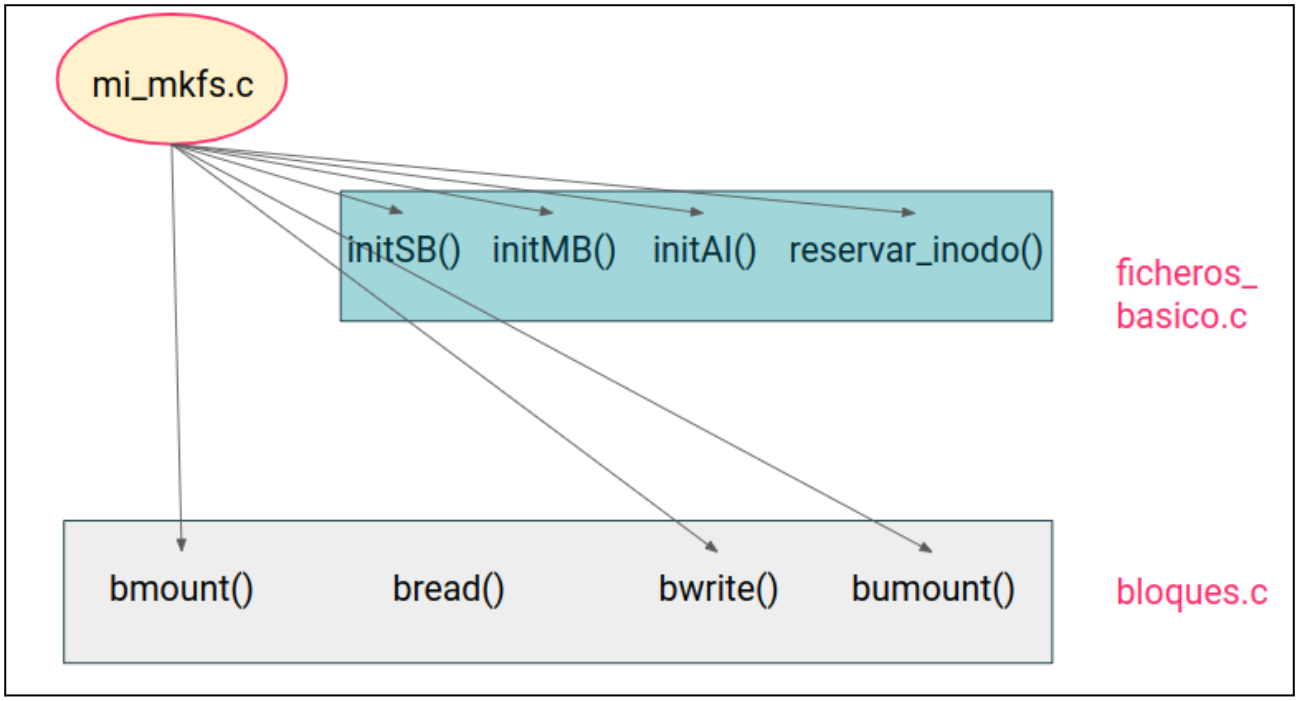
- A continuación **inicializamos todos los campos del inodo** al que apuntaba inicialmente el superbloque:
  - tipo (pasado como argumento)
  - permisos (pasados como argumento)
  - cantidad de enlaces de entradas en directorio: 1
  - tamaño en bytes lógicos: 0
  - *timestamps* de creación para todos los campos de fecha y hora: *time(NULL)*
  - cantidad de bloques ocupados en la zona de datos: 0
  - punteros a bloques directos: 0 (el valor 0 indica que no apuntan a nada)
  - punteros a bloques indirectos: 0 (el valor 0 indica que no apuntan a nada)
- Utilizar la función *escribir_inodo()* para escribir el inodo inicializado en la posición del que era el primer inodo libre, *posInodoReservado*.
- Decrementar la **cantidad de inodos libres**, y reescribir el superbloque.
- Devolver *posInodoReservado*.

mi_mkfs.c

En el programa **mi_mkfs.c** habrá que **crear el directorio raíz.** Podemos utilizar la función *reservar_inodo ('d', 7)* para ello.

En la inicialización del superbloque del nivel anterior habíamos indicado que el primer inodo libre era el 0 y que la cantidad de inodos libres inicial era *ninodos* (la función *reservar_inodo()* actualizará esos valores).

Tras la creación del directorio raíz, el primer inodo libre pasará a ser el 1 y en el sistema habrá un inodo libre menos.



Esquema de llamadas de mi_mkfs.c

Tests de prueba

Para comprobar el buen funcionamiento de las funciones de este nivel podéis modificar el programa de pruebas **leer_sf.c**[4] para:

- mostrar el superbloque (ya se habrán inicializado los metadatos y habrán esos bloques libres menos, y también se habrá creado el inodo raíz con lo cual habrá 1 inodo libre menos y se habrá actualizado la cabecera de la lista de inodos libres)
- mostrar el MB (y así comprobar el funcionamiento de *escribir_bit()* y *leer_bit()*). Si no queréis mostrar los 100.000 bits bastan el 1º y último de cada zona del dispositivo.
- reservar y liberar un bloque (y así comprobar las funciones *reservar_bloque()* y *liberar_bloque()*). Para mostrar los cambios en la cantidad de bloques libres tras cada acción habra que leer el superbloque.
- mostrar el inodo del directorio raíz (y así comprobar *reservar_inodo()*, *escribir_inodo()* y *leer_inodo()*).

Para mostrar en formato amigable los sellos de tiempo de un inodo que están en *epoch*:

```
#include <time.h> //esta librería incluirla en ficheros_basico.h
...
struct tm *ts;
char atime[80];
char mtime[80];
char ctime[80];

struct inodo inodo;
int ninodo;
...
leer_inodo(ninodo, &inodo);
ts = localtime(&inodo.atime);
strftime(atime, sizeof(atime), "%a %Y-%m-%d %H:%M:%S", ts);
ts = localtime(&inodo.mtime);
strftime(mtime, sizeof(mtime), "%a %Y-%m-%d %H:%M:%S", ts);
ts = localtime(&inodo.ctime);
strftime(ctime, sizeof(ctime), "%a %Y-%m-%d %H:%M:%S", ts);
printf("ID: %d ATIME: %s MTIME: %s CTIME: %s\n",ninodo,atime,mtime,ctime);
...
```

Ejemplo[5] de ejecución de **leer_sf** en este nivel para **100.000 bloques** con *BLOCKSIZE* = 1KB:

```
$ ./mi_mkfs disco 100000
```

[4] En este nivel ya no hay que mostrar la inicialización de la lista enlazada de inodos. No borrar el código, solo dejarlo comentado.
[5] El valor de *posbyte* mostrado por la función leer_bit() es antes de relativizarlo a 1024.

```
$ ./leer_sf disco

DATOS DEL SUPERBLOQUE
posPrimerBloqueMB = 1
posUltimoBloqueMB = 13
posPrimerBloqueAI = 14
posUltimoBloqueAI = 3138
posPrimerBloqueDatos = 3139
posUltimoBloqueDatos = 99999
posInodoRaiz = 0
posPrimerInodoLibre = 1
cantBloquesLibres = 96861
cantInodosLibres = 24999
totBloques = 100000
totInodos = 25000

RESERVAMOS UN BLOQUE Y LUEGO LO LIBERAMOS
Se ha reservado el bloque físico nº 3139 que era el 1º libre indicado por el MB
SB.cantBloquesLibres = 96860
Liberamos ese bloque y después SB.cantBloquesLibres = 96861

MAPA DE BITS CON BLOQUES DE METADATOS OCUPADOS[6]
[leer_bit(0)→ posbyte:0, posbit:0, nbloqueMB:0, nbloqueabs:1)]
posSB: 0 → leer_bit(0) = 1
[leer_bit(1)→ posbyte:0, posbit:1, nbloqueMB:0, nbloqueabs:1)]
SB.posPrimerBloqueMB: 1 → leer_bit(1) = 1
[leer_bit(13)→ posbyte:1, posbit:5, nbloqueMB:0, nbloqueabs:1)]
SB.posUltimoBloqueMB: 13 → leer_bit(13) = 1
[leer_bit(14)→ posbyte:1, posbit:6, nbloqueMB:0, nbloqueabs:1)]
SB.posPrimerBloqueAI: 14 → leer_bit(14) = 1
[leer_bit(3138)→ posbyte:392, posbit:2, nbloqueMB:0, nbloqueabs:1)]
SB.posUltimoBloqueAI: 3138 → leer_bit(3138) = 1
[leer_bit(3139)→ posbyte:392, posbit:3, nbloqueMB:0, nbloqueabs:1)]
SB.posPrimerBloqueDatos: 3139 → leer_bit(3139) = 0
[leer_bit(99999)→ posbyte:12499, posbit:7, nbloqueMB:12, nbloqueabs:13)]
SB.posUltimoBloqueDatos: 99999 → leer_bit(99999) = 0

DATOS DEL DIRECTORIO RAIZ
tipo: d
permisos: 7
atime: Tue 2021-03-09 18:16:08
ctime: Tue 2021-03-09 18:16:08
mtime: Tue 2021-03-09 18:16:08
```

[6] Por simplicidad basta mostrar los bits de los bloques de inicio y fin de cada zona del dispositivo en vez de un listado de los nbloques del dispositivo.

```
nlinks: 1
tamEnBytesLog: 0
numBloquesOcupados: 0
```

Ejemplo[7] de ejecución de **leer_sf** en este nivel para **1.000.000 bloques** con *BLOCKSIZE* = 1KB:

```
$ rm disco
$ ./mi_mkfs disco 1000000
$ ./leer_sf disco

DATOS DEL SUPERBLOQUE
posPrimerBloqueMB = 1
posUltimoBloqueMB = 123
posPrimerBloqueAI = 124
posUltimoBloqueAI = 31373
posPrimerBloqueDatos = 31374
posUltimoBloqueDatos = 999999
posInodoRaiz = 0
posPrimerInodoLibre = 1
cantBloquesLibres = 968626
cantInodosLibres = 249999
totBloques = 1000000
totInodos = 250000


RESERVAMOS UN BLOQUE Y LUEGO LO LIBERAMOS
Se ha reservado el bloque físico nº 31374 que era el 1º libre indicado por el MB
SB.cantBloquesLibres = 968625
Liberamos ese bloque y después SB.cantBloquesLibres = 968626


MAPA DE BITS CON BLOQUES DE METADATOS OCUPADOS
[leer_bit(0)→ posbyte:0, posbit:0, nbloqueMB:0, nbloqueabs:1)]
posSB: 0 → leer_bit(0) = 1
[leer_bit(1)→ posbyte:0, posbit:1, nbloqueMB:0, nbloqueabs:1)]
SB.posPrimerBloqueMB: 1 → leer_bit(1) = 1
[leer_bit(123)→ posbyte:15, posbit:3, nbloqueMB:0, nbloqueabs:1)]
SB.posUltimoBloqueMB: 123 → leer_bit(123) = 1
[leer_bit(124)→ posbyte:15, posbit:4, nbloqueMB:0, nbloqueabs:1)]
SB.posPrimerBloqueAI: 124 → leer_bit(124) = 1
[leer_bit(31373)→ posbyte:3921, posbit:5, nbloqueMB:3, nbloqueabs:4)]
SB.posUltimoBloqueAI: 31373 → leer_bit(31373) = 1
```

[7] El valor de *posbyte* mostrado por la función leer_bit() es antes de relativizarlo a 1024.

```
[leer_bit(31374)→ posbyte:3921, posbit:6, nbloqueMB:3, nbloqueabs:4)]
SB.posPrimerBloqueDatos: 31374 → leer_bit(31374) = 0
[leer_bit(999999)→ posbyte:124999, posbit:7, nbloqueMB:122, nbloqueabs:123)]
SB.posUltimoBloqueDatos: 999999 → leer_bit(999999) = 0

DATOS DEL DIRECTORIO RAIZ
tipo: d
permisos: 7
atime: Wed 2022-07-20 12:44:26
ctime: Wed 2022-07-20 12:44:26
mtime: Wed 2022-07-20 12:44:26
nlinks: 1
tamEnBytesLog: 0
numBloquesOcupados: 0
```