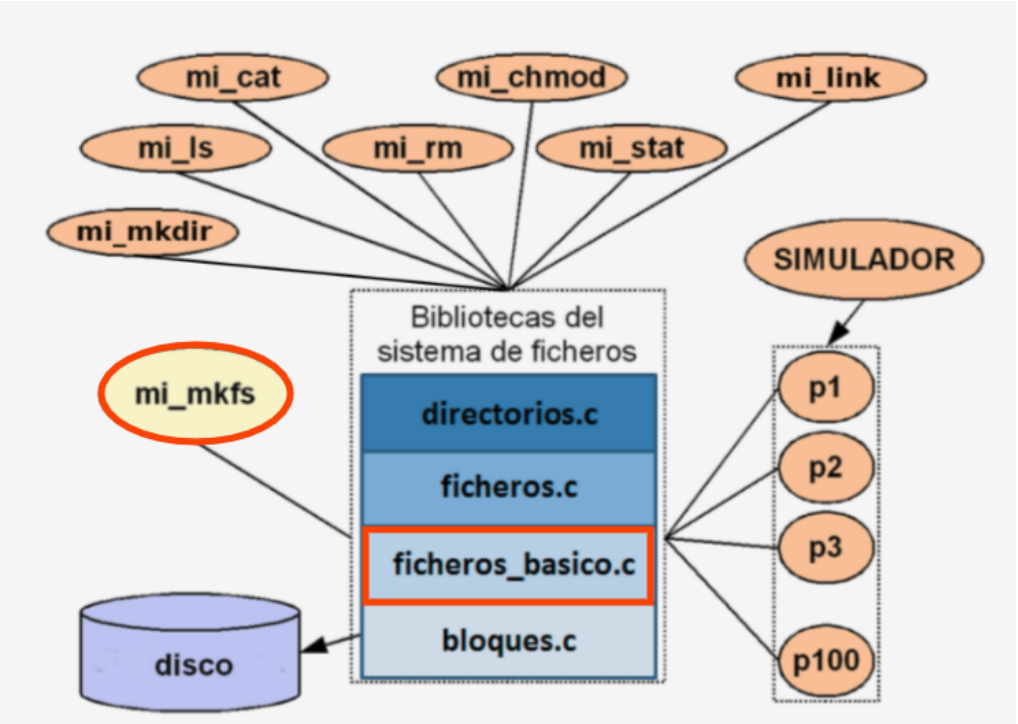
Nivel 2: ficheros_basico.c {tamMB(), tamAI(), initSB(), initMB(), initAI()}



En el nivel anterior estuvimos trabajando con la capa de **bloques.c**, implementando las operaciones básicas para trabajar con el dispositivo virtual: montar y desmontar el disco y leer/escribir un bloque, e inicializamos nuestro dispositivo virtual con todos los bloques a 0 mediante una primera versión de **mi_mkfs.c**.

A continuación vamos a iniciar otra capa de la biblioteca del sistema de ficheros, **ficheros_basico.c,** para poder establecer y gestionar las distintas áreas en las que se va a estructurar el dispositivo virtual: superbloque (*SB*), mapa de bits (*MB*), array de inodos (*AI*), y datos.

Super-bloque	Mapa de bits	Array de inodos	Datos

En este segundo nivel inicializaremos el superbloque (*initSB()*), el mapa de bits (*initMB()*) y el array de inodos (*initAI()*). Para delimitar cada una de las áreas del disco virtual necesitaremos averiguar cuántos bloques ocupa el mapa de bits (*tamMB()*) y los bloques que ocupa el array de inodos (*tamAI()*). El superbloque ya sabemos que ocupa un bloque, *tamSB*. Finalmente actualizaremos la función de formateo del disco virtual, **mi_mkfs.c** para llamar a las funciones de inicialización que habremos implementado.

Las 3 primeras áreas del disco virtual constituyen los **metadatos** del sistema de ficheros**:**

- El **superbloque,** *SB***,** es un bloque que contiene información general sobre el sistema de ficheros.
  - Su ubicación, *posSB*, en nuestro dispositivo virtual será el bloque físico 0[1], y su tamaño, *tamSB*, será de 1 bloque (*BLOCKSIZE*).
  - Los datos del superbloque se inicializarán con la función *initSB().*
- El **mapa de bits,** *MB***,** nos servirá para gestionar el espacio libre en nuestro sistema de ficheros.
  - Contendrá un bit por cada bloque físico del sistema de ficheros, que valdrá 0 para los bloques libres y 1 para bloques ocupados.
  - El tamaño en bloques del mapa de bits nos lo calculará la función *tamMB().*
  - El mapa de bits se inicializará con la función *initMB()*.
- Un **array de inodos,** *AI*. Un **inodo** es una estructura que contiene las características (metadatos) de un directorio o fichero del sistema de ficheros y los punteros necesarios a la zona de datos para obtener su contenido.
  - Un inodo se identifica por su **nº de inodo**, *ninodo*, que viene definido implícitamente por su posición dentro del array.
  - La **cantidad de inodos** , *ninodos*, definida por el administrador del sistema de archivos (de forma heurística) indica la máxima cantidad de directorios y ficheros que pueden llegar a existir en el sistema, en nuestro caso será igual a ***nbloques/4***.
  - El tamaño en bloques del array de inodos, *tamAI*, nos lo calculará la función *tamAI().*
  - Dentro del array de inodos, los **inodos libres** se organizan como una **lista enlazada**, mientras que los inodos ocupados se corresponden a directorios o ficheros existentes. La función *initAI()* se encargará de crear inicialmente esa lista enlazada.
- El contenido de los directorios (entradas) y el de los ficheros en sí se almacenan en la **zona de datos**.

Comenzamos a desarrollar en **ficheros_basico.c** las funciones de tratamiento básico del sistema de ficheros y también definiremos la estructura del superbloque y la estructura de un inodo en **ficheros_basico.h**. Después actualizaremos **mi_mkfs.c** para llamar a las funciones de inicialización de cada zona de metadatos, que habremos desarrollado en **ficheros_basico.c**.

[1] Si tuviéramos un dispositivo real de memoria secundaria, el bloque 0 estaría reservado como bloque de arranque del disco, y el SB estaría en el bloque físico 1

ficheros_basico.h

En esta cabecera, haremos el *include* de **bloques.h**, además de declarar las funciones de **ficheros_basicos.c**.

También definiremos los símbolos para la posición del superbloque y su tamaño, y la estructura con sus campos.

```
#define posSB 0 // el superbloque se escribe en el primer bloque de nuestro FS
#define tamSB 1
```

```
struct superbloque {
  unsigned int posPrimerBloqueMB;       // Posición absoluta del primer bloque del mapa de bits
  unsigned int posUltimoBloqueMB;       // Posición absoluta del último bloque del mapa de bits
  unsigned int posPrimerBloqueAI;       // Posición absoluta del primer bloque del array de inodos
  unsigned int posUltimoBloqueAI;       // Posición absoluta del último bloque del array de inodos
  unsigned int posPrimerBloqueDatos;    // Posición absoluta del primer bloque de datos
  unsigned int posUltimoBloqueDatos;    // Posición absoluta del último bloque de datos
  unsigned int posInodoRaiz;            // Posición del inodo del directorio raíz (relativa al AI)
  unsigned int posPrimerInodoLibre;     // Posición del primer inodo libre (relativa al AI)
  unsigned int cantBloquesLibres;       // Cantidad de bloques libres (en todo el disco)
  unsigned int cantInodosLibres;        // Cantidad de inodos libres (en el AI)
  unsigned int totBloques;              // Cantidad total de bloques del disco
  unsigned int totInodos;               // Cantidad total de inodos (heurística)
  char padding[BLOCKSIZE - 12 * sizeof(unsigned int)]; // Relleno para ocupar el bloque completo
};
```

Las variables que declaremos de tipo *struct superbloque* serán locales en cada función y se tendrá que acceder al disco para obtener el valor de los campos que varíen a lo largo de la vida del sistema, ya que cuando lancemos varios procesos, si fuese global, cada proceso tendría su propia copia del *SB* y se trata de datos comunes a todo el sistema. Una mejora del sistema sería guardar el *SB* en memoria compartida usando *mmap()*[2].

En esta cabecera definiremos también el símbolo para el tamaño del inodo, *INODOSIZE*, y su estructura[3]:

```
#define INODOSIZE 128 // tamaño en bytes de un inodo
```

```
struct inodo {    // comprobar que ocupa 128 bytes haciendo un sizeof(inodo)!!!
  unsigned char tipo;    // Tipo ('l':libre, 'd':directorio o 'f':fichero)
  unsigned char permisos; // Permisos (lectura y/o escritura y/o ejecución)
```

[2] Veremos más adelante cómo hacerlo.
[3] Fijaos que entre los datos del inodo no aparece el **nombre del fichero**, ya que éste formará parte de la entrada de directorio correspondiente.

```
/* Por cuestiones internas de alineación de estructuras, si se está utilizando
 un tamaño de palabra de 4 bytes (microprocesadores de 32 bits):
unsigned char reservado_alineacion1 [2];
en caso de que la palabra utilizada sea del tamaño de 8 bytes
(microprocesadores de 64 bits): unsigned char reservado_alineacion1 [6]; */
unsigned char reservado_alineacion1[6];

time_t atime; // Fecha y hora del último acceso a datos
time_t mtime; // Fecha y hora de la última modificación de datos
time_t ctime; // Fecha y hora de la última modificación del inodo

/* comprobar que el tamaño del tipo time_t para vuestra plataforma/compilador es 8:
printf ("sizeof time_t is: %ld\n", sizeof(time_t)); */

unsigned int nlinks;          // Cantidad de enlaces de entradas en directorio
unsigned int tamEnBytesLog;   // Tamaño en bytes lógicos (EOF)
unsigned int numBloquesOcupados; // Cantidad de bloques ocupados zona de datos

unsigned int punterosDirectos[12];  // 12 punteros a bloques directos
unsigned int punterosIndirectos[3]; /* 3 punteros a bloques indirectos:
1 indirecto simple, 1 indirecto doble, 1 indirecto triple */

/* Utilizar una variable de alineación si es necesario  para vuestra plataforma/compilador   */
char padding[INODOSIZE - 2 * sizeof(unsigned char) - 3 * sizeof(time_t) - 18 * sizeof(unsigned int)
- 6 * sizeof(unsigned char)];
// Hay que restar también lo que ocupen las variables de alineación utilizadas!!!
};
```

Una estructura, mejor que la del inodo con el padding, para que ocupe exactamente 128 bytes y funcione en todas las arquitecturas es utilizar en realidad una **unión** de la estructura del inodo y una variable que ocupe 128 bytes (ver Anexo). Su utilización es voluntaria.

Los **permisos** se manejan como los permisos básicos característicos de GNU/Linux: lectura ('r'), escritura ('w') y ejecución ('x'). Sólo tendremos en cuenta los de **usuario**.

Así que las posibles combinaciones de permisos son las que van de 000 a 111 en binario (de 0 a 7 en octal): el primer bit para la 'r', el segundo bit para la 'w' y el tercer bit para la 'x'.

Para cada ***timestamp*** (fecha y hora) atime, mtime y ctime, se pueden utilizar los tipos y las funciones declaradas en *time.h*. Se pueden inicializar con la función *time(NULL)*.

Hay que estar familiarizado con el concepto de fecha y hora en forma *epoch*: tipo *time_t* [4].

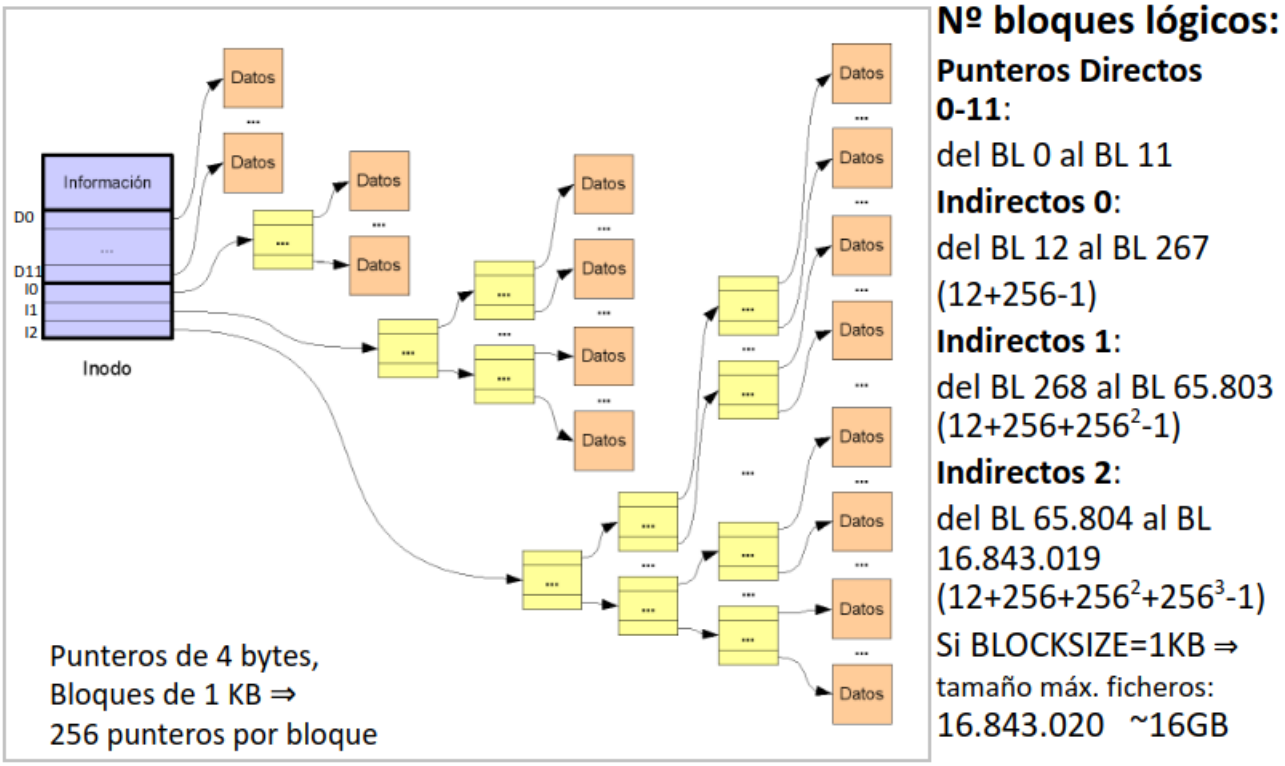
[4] time_t es un long int que contiene un *timestamp* expresado en segundos después del inicio de la época UNIX (1 de Enero de 1970 00:00:00 GMT).

El **nº de enlaces** por defecto es 1. Aumentará con el uso de la función *mi_link()* y se decrementará con *mi_unlink()* de la capa de directorios.

El **tamaño en bytes lógico** nos indica la posición del byte lógico más alejado en el fichero, independientemente de cuantos bytes se hayan escrito. Por ejemplo si el byte lógico más alejado está en el *offset* 10568 (en bytes), entonces el *tamEnBytesLog* = 10569

La **cantidad de bloques ocupados** incluye tanto los bloques de datos propiamente dichos como los bloques índices de la zona de datos correspondientes a tal inodo.

Los **punteros Directos e Indirectos de varios niveles** nos permiten hacer la correspondencia de los bloques lógicos del inodo con los bloques físicos del dispositivo.



Los **punteros directos D0-D11** contienen la dirección (nº de bloque del dispositivo virtual) de los bloques físicos de datos correspondientes a los bloques lógicos del 0 al 11.

De esta manera, si *BLOCKSIZE = 1.024*, para ficheros pequeños de hasta 12KBs se podría acceder directamente a su contenido, a través de los punteros directos.

El **puntero indirecto I0** contiene la dirección (nº de bloque del dispositivo virtual) de un bloque índice con *NPUNTEROS*, siendo *NPUNTEROS = BLOCKSIZE / sizeof(unsigned int)*. Para *BLOCKSIZE = 1024*, *NPUNTEROS = 256*. Esos punteros son las direcciones (nº de bloque) de los bloques físicos de datos correspondientes a los bloques lógicos del 12 al *INDIRECTOS0* - 1, siendo *INDIRECTOS0 = 12 + NPUNTEROS*.

Para *BLOCKSIZE = 1.024*, nos direccionaría los bloques lógicos del 12 al 267 (12+256-1).

El **puntero indirecto I1** contiene la dirección del bloque físico índice con *NPUNTEROS* a los bloques índices de *NPUNTEROS* a los bloques físicos de datos correspondientes a los bloques lógicos del 268 al *INDIRECTOS1 - 1*, siendo *INDIRECTOS1 = INDIRECTOS0 + NPUNTEROS*^{2}.

Para *BLOCKSIZE = 1.024*, nos direccionaría los bloques lógicos del 268 al 65.803 (268+256^2-1).

El **puntero indirecto I2** contiene la dirección del bloque físico índice con *NPUNTEROS* a los bloques físicos de índices de *NPUNTEROS*, los cuales apuntan a los bloques físicos de índices que contienen la dirección de los bloques físicos de datos correspondientes a los bloques lógicos del 65.804 al *INDIRECTOS2 - 1*, siendo *INDIRECTOS2* = *INDIRECTOS1* + *NPUNTEROS*^{3}.

Para *BLOCKSIZE = 1.024*, nos direccionaría los bloques lógicos del 65.804 al 16.843.019 (65.804 + 256^3 - 1).

Con esta estructura de punteros del inodo, para un tamaño de bloque de 1KB podríamos tener ficheros de hasta 16.842.020 KBs (o sea de un tamaño lógico de unos 16 GBs!!!).

ficheros_basico.c

Veamos ahora cómo se ha de llevar a cabo la implementación de las funciones de este nivel:

1)int tamMB(unsigned int nbloques);

Calcula el tamaño en bloques necesario para el mapa de bits.

Dado que cada bit representa un bloque y que los bits se agrupan de 8 en 8 para constituir bytes, y los bytes se agrupan en bloques de tamaño *BLOCKSIZE*, el tamaño del mapa de bits lo obtendremos mediante:

(*nbloques* / 8bits) / *BLOCKSIZE*

Utilizaremos el operador módulo **%** para saber si necesitamos esa cantidad justa o si necesitamos añadir un bloque adicional para los bytes restantes (el resto de la división).

Si por ejemplo tenemos 100.000 bloques de tamaño 1KB , el tamaño del *MB* será: (100.000/8)/1.024=12 → 13 bloques (hemos incrementado en 1 el resultado de la división entera con 1024 porque el módulo no es 0)

2)int tamAI(unsigned int ninodos);

Calcula el tamaño en bloques del array de inodos.

Hay que determinar **de manera heurística** la cantidad de inodos de nuestro sistema (nbloques/2, nbloques/4, nbloques/8...), cantidad que se mantendrá durante toda la vida del sistema de ficheros. En nuestro sistema de ficheros usaremos *ninodos=nbloques/4*. El programa **mi_mkfs.c** le pasará este dato a esta función como parámetro al llamarla.

Una vez determinada la cantidad de nodos del sistema, ya podemos calcular el tamaño del array de inodos, en bloques:

tamAI = (*ninodos* * *INODOSIZE*) / *BLOCKSIZE*

que es lo mismo que *ninodos* / (*BLOCKSIZE* / *INODOSIZE*)

Utilizaremos el operador módulo % con *BLOCKSIZE* para saber si necesitamos esa cantidad justa o si necesitamos añadir un bloque adicional para el resto, resultado de la división.

El tamaño de nuestros inodos (*INODOSIZE*) será de **128 bytes**, así que en un bloque de tamaño 1024 bytes nos caben exactamente 8 inodos (1024/128).

Ejemplos:

- *ninodos*= 5.000, *INODOSIZE*= 128 bytes, *BLOCKSIZE* = 1.024 ⇒ *tamAI* = 625 bloques
- *ninodos*= 125.500, *INODOSIZE*= 128 bytes, *BLOCKSIZE* = 1.024 ⇒ *tamAI* = 15.688 bloques
- *ninodos*= 25.000, *INODOSIZE*= 128 bytes, *BLOCKSIZE* = 1.024 ⇒ *tamAI* = 3.125 bloques

3)int initSB(unsigned int nbloques, unsigned int ninodos);

Inicializa los datos del superbloque.

Se trata de definir una variable de tipo struct superbloque que se vaya rellenando con la información pertinente[5]:

- **Posición[6] del primer bloque del mapa de bits**

```
SB.posPrimerBloqueMB = posSB + tamSB //posSB = 0, tamSB = 1
```

SB.posPrimerBloqueMB = 0+1 = 1

- **Posición del último bloque del mapa de bits**

```
SB.posUltimoBloqueMB = SB.posPrimerBloqueMB + tamMB(nbloques) - 1
```

tamMB(100000) = (100000/8/1024)+1 = 13

SB.posUltimoBloqueMB = 1+13-1 = 13

- **Posición del primer bloque del array de inodos**

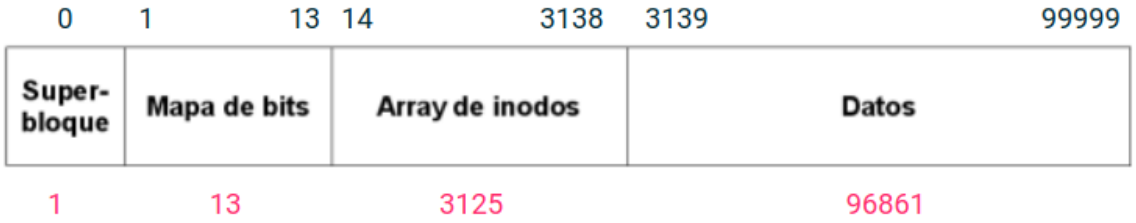
```
SB.posPrimerBloqueAI = SB.posUltimoBloqueMB + 1
```

SB.posPrimerBloqueAI = 13+1 = 14

[5] Debajo de cada asignación genérica os pongo un ejemplo numérico particular para *nbloques*=100.000, *BLOCKSIZE*=1024 e *INODOSIZE* = 128

[6] Todas las posiciones se refieren al nº de bloque

- **Posición del último bloque del array de inodos**
  `SB.posUltimoBloqueAI = SB.posPrimerBloqueAI + tamAI(ninodos) - 1`
  tamAI(ninodos) = 25000*128/1024 = 3125
  SB.posUltimoBloqueAI = 14+3125-1 = 3138
- **Posición del primer bloque de datos**
  `SB.posPrimerBloqueDatos = SB.posUltimoBloqueAI + 1`
  SB.posPrimerBloqueDatos = 3138+1 = 3139
- **Posición del último bloque de datos**
  `SB.posUltimoBloqueDatos = nbloques-1`
  SB.posUltimoBloqueDatos = 100000-1 = 99999



- **Posición del inodo del directorio raíz en el array de inodos**
  `SB.posInodoRaiz = 0`
- **Posición del primer inodo libre en el array de inodos**
  - Inicialmente: `SB.posPrimerInodoLibre = 0`
  - Cuando llamemos a *reservar_inodo('d', 7)* para crear el directorio raíz (Nivel 3), pasará a valer 1.
  - Posteriormente se irá actualizando para apuntar a la cabeza de la lista de inodos libres (mediante las llamadas a las funciones *reservar_inodo()* y *liberar_inodo()*)
- **Cantidad de bloques libres en el SF**
  - Inicialmente: `SB.cantBloquesLibres = nbloques`
  - Cuando indiquemos en el mapa de bits los bloques que ocupan los metadatos (el *SB*, el propio *MB* y el *AI*), restaremos esos bloques de la cantidad de bloques libres.
  - Al reservar un bloque ⇒ *SB.cantBloquesLibres--*
  - Al liberar un bloque ⇒ *SB.cantBloquesLibres++*
- **Cantidad de inodos libres en el array de inodos**
  - Inicialmente: `SB.cantInodosLibres = ninodos` (el 1er inodo será para el directorio raíz)
  - Al reservar un inodo ⇒ *SB.cantInodosLibres--*
  - Al liberar un inodo ⇒ *SB.cantInodosLibres++*
- **Cantidad total de bloques**
  - Se pasará como argumento en la línea de comandos al inicializar el sistema:

$ ./mi_mkfs <nombre_fichero> <nbloques>

y lo recibimos como parámetro

```
SB.totBloques = nbloques
```

- **Cantidad total de inodos**
  - Determinada por el administrador del sistema de forma heurística (*ninodos = nbloques/4*) y recibida por parámetro

    ```
    SB.totinodos = ninodos
    ```
  - Se indicará su valor en mi_mkfs.c y se pasará también como parámetro a la función de inicialización del array de inodos, *initAI()*

Al finalizar las inicializaciones de los campos, se escribe la estructura en el bloque *posSB* mediante la función *bwrite()*.

4)int initMB();

Inicializa el mapa de bits. En este nivel, de momento, simplemente pondremos a 0 todos los bits del mapa de bits.

Para ello utilizaremos un *buffer* (será un array de tipo *unsigned char* del tamaño de un bloque) con todos los bits a cero. La función *memset()* puede sernos útil para asignar de golpe un valor a todos los elementos de un array.

El contenido del *buffer* se escribe en los bloques correspondientes al mapa de bits[7], mediante sucesivas llamadas a la función *bwrite()* (habrá que leer primeramente el superbloque para obtener la localización del mapa de bits).

En el nivel siguiente, cuando ya dispongamos de la función *escribir_bit()*, la podríamos utilizar para escribir a 1 los bits que representan los bloques ocupados por los metadatos, o sea el superbloque, el mapa de bits y el array de inodos (y restando esos bloques al total de bloques libres en el superbloque, si no los hemos restado al inicializarlo) pero implica un enorme trasiego de bloques. También se puede hacer en este nivel sin necesidad de usar tal función y además de forma **más eficiente**. Os dejo un ejemplo de un caso para que lo generalicéis:

> Si inicializamos el dispositivo con 100.000 bloques de tamaño *BLOCKSIZE* = 1024, ya hemos visto que los metadatos ocupan 3139 bloques (*tamSB*+*tamMB*+*tamAI*), por tanto hemos de poner 3139 bits a 1.
>
> Si hacemos la operación 3139 / 8 / 1024 vemos que todos esos bits caben en un bloque, el bloque 0 del MB. Así que llevamos ese bloque a memoria usando un array de caracteres, *bufferMB*.
>
> Vamos a ver cuántos de esos 1024 bytes hay que poner a 1:

[7] Sabemos cuántos bloques ocupa el mapa de bits gracias a la función *tamMB()*, y también sabemos cuál es la primera posición del mapa de bits en el dispositivo y cuál es la última (esa información está en el superbloque)

3139 / 8 = 392

Para ello podemos iterar desde *i* = 0 a 391 haciendo la siguiente asignación:

bufferMB[*i*] = 255 (en binario es 11111111)

Pero aún tenemos un resto de 3 bits (3139 % 8 = 3) que habrá que poner a 1 a la izquierda en un byte adicional, en el 392. Eso implica escribir en binario 11100000, o sea 224 en decimal ($2^7+2^6+2^5$):

bufferMB[392] = 224.

Los restantes bytes de ese bloque (desde el 393 al 1023) se tendrían que poner a 0. Y luego salvar *bufferMB* al dispositivo.

También habría que tener en cuenta que los bits correspondientes a los metadatos podrían ocupar más de 1 bloque, dependiendo de la cantidad total de bloques de nuestro dispositivo virtual, *nbloques*, y de *BLOCKSIZE*. En tal caso habría que poner los bloques previos directamente a 1 (usando *memset()* para igualar a 255 todos los bytes del buffer de memoria y escribiéndolo en el disco), y realizar las operaciones anteriormente descritas en el bloque que sólo tuviera algunos bits a 1. Con el test de prueba del nivel 3 se podrá comprobar el funcionamiento de esta función para diferentes valores de *nbloques*.

Y no hay que olvidar actualizar la cantidad de bloques libres en el superbloque y grabarlo.

Cuando marquemos en el mapa de bits los bloques que ocupan los metadatos (el *SB*, el propio *MB* y el *AI*), restaremos esos bloques de la cantidad de bloques libres en el campo del superbloque, *SB.cantBloquesLibres*[8].

5)int initAI();

Esta función se encargará de inicializar la lista de **inodos libres**. Dado que al principio todos los inodos están libres, hay que crear una función que enlace todos los inodos entre sí. Cuando el sistema de ficheros esté en funcionamiento, serán las funciones *reservar_inodo()* y *liberar_inodo()*, del siguiente nivel, las que gestionarán esta lista, actualizándola siempre por la cabecera.

No es necesario definir nuevos campos en el inodo para apuntar al siguiente inodo libre, dado que la mayoría sólo tienen sentido cuando el inodo está ocupado. **Utilizaremos el campo de *punterosDirectos[0]* para enlazar la lista de inodos libres.**

De nuevo, se trata de definir un *buffer* para ir recorriendo el array de inodos, pasando cada vez un bloque a memoria principal desde el dispositivo virtual, e ir actualizándolo con esos inodos enlazados, para después salvarlo en el dispositivo mediante una llamada a la función *bwrite()*. Ese *buffer* será de tamaño *BLOCKSIZE* y tendrá la siguiente estructura de datos: *struct inodo inodos[BLOCKSIZE/INODOSIZE]*.

Pasos:

- Primeramente leeremos el superbloque para obtener la localización del array de inodos.

[8] Si no lo hemos hecho en este nivel, porque esperemos a utilizar la función *escribir_bit()* del nivel 3, entonces los restaremos después en ese nivel.

- Habrá que inicializar el primer elemento del array de punteros directos de cada inodo, *punterosDirectos[0]*, con una variable incremental (ya que inicialmente todos los inodos están libres y en la lista enlazada cada uno apunta al siguiente). El último de la lista tendrá que apuntar a un nº muy grande (*NULL*), que podemos expresar con *UINT_MAX* (el máximo valor para un *unsigned int*) y en tal caso se requiere un `#include <limits.h>` en **ficheros_basico.h**.
- Iteraremos para cada bloque (desde la posición del 1er bloque del array de inodos hasta el último), y para cada inodo dentro de un bloque. Cada bloque contiene una cantidad de *inodos* = (*BLOCKSIZE* / *INODOSIZE*), excepto el último bloque que no tiene porqué estar completamente lleno.

```
SB.totinodos = ninodos
struct inodo inodos [BLOCKSIZE/INODOSIZE]
...
contInodos := SB.posPrimerInodoLibre + 1;  //si hemos inicializado SB.posPrimerInodoLibre = 0
para (i = SB.posPrimerBloqueAI; i <= SB.posUltimoBloqueAI; i++) hacer //para cada bloque del AI
   leer el bloque de inodos i del dispositivo virtual
   para (j = 0; j < BLOCKSIZE / INODOSIZE;  j++) hacer  //para cada inodo del AI
      inodos[j].tipo := 'l';  //libre
      si (contInodos < SB.totInodos) entonces  //si no hemos llegado al último inodo
            inodos[j].punterosDirectos[0] := contInodos;  //enlazamos con el siguiente
            contInodos++;
       si_no //hemos llegado al último inodo
            inodos[j].punterosDirectos[0] := UINT_MAX;
            //hay que salir del bucle, el último bloque no tiene por qué estar completo !!!
       fsi
   fpara
   escribir el bloque de inodos i  en el dispositivo virtual
fpara
```

Indicamos que el tipo de inodo es libre ('l'). El resto de campos del inodo no es necesario inicializarlos.

---

mi_mkfs.c

Ahora hay que incorporar en **mi_mkfs.c** un include de *ficheros_basico.h* (en vez de *bloques.h*) y las **llamadas** a las funciones *initSB(), initMB(), initAI()* para mejorar el formateo del sistema de ficheros del nivel 1.

Compilación

$ gcc -o mi_mkfs mi_mkfs.c bloques.c ficheros_basico.c

o mucho mejor modificar el **Makefile** del nivel 1.

Tests de prueba

Comenzar a desarrollar un programa de pruebas **leer_sf.c**[9] que nos ayude a determinar la información almacenada en el superbloque, en el mapa de bits o en el array de inodos:. Sintaxis:

$ ./leer_sf <nombre_dispositivo>

(hay que montar y desmontar el dispositivo virtual y hacer un *include* de **ficheros_basico.h**!!!)

- De momento podéis mostrar por pantalla todos los campos del superbloque.
- Mostrar también el tamaño del *struct inodo:*

```
printf ("sizeof struct inodo is: %lu\n", sizeof(struct inodo));
```

- Podéis hacer también un recorrido de la lista de inodos libres (mostrando para cada inodo el campo *punterosDirectos[0]*).

Más adelante podréis ampliar el programa para leer las otras estructuras de metadatos.

Ejemplo de ejecución de **leer_sf** para 100.000 bloques:

```
$ make clean
rm -rf *.o *~ mi_mkfs leer_sf disco* ext*
$ make
gcc -c -g -Wall -std=c99 -o bloques.o -c bloques.c
gcc -c -g -Wall -std=c99 -o ficheros_basico.o -c ficheros_basico.c
gcc -c -g -Wall -std=c99 -o mi_mkfs.o -c mi_mkfs.c
gcc -c -g -Wall -std=c99 -o leer_sf.o -c leer_sf.c
gcc bloques.o ficheros_basico.o  mi_mkfs.o -o mi_mkfs
gcc bloques.o ficheros_basico.o  leer_sf.o -o leer_sf
$ ./mi_mkfs disco 100000
$ ./leer_sf disco
DATOS DEL SUPERBLOQUE
posPrimerBloqueMB = 1
posUltimoBloqueMB = 13
posPrimerBloqueAI = 14
posUltimoBloqueAI = 3138
```

[9] leer_sf.c no forma parte del sistema de ficheros, es un programa auxiliar que usamos temporalmente para poder ir explorando nuestro sistema de ficheros hasta que tengamos desarrolladas las funciones necesarias de capas superiores.

```
posPrimerBloqueDatos = 3139
posUltimoBloqueDatos = 99999
posInodoRaiz = 0
posPrimerInodoLibre = 0
cantBloquesLibres = 100000
cantInodosLibres = 25000
totBloques = 100000
totInodos = 25000

sizeof struct superbloque: 1024
sizeof struct inodo: 128


RECORRIDO LISTA ENLAZADA DE INODOS LIBRES
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28[10] ...
... 24980 24981 24982 24983 24984 24985 24986 24987 24988 24989 24990 24991
24992 24993 24994 24995 24996 24997 24998 24999 -1
```

[10] Sólo he puesto los primeros y los últimos valores por cuestiones de espacio pero os debería mostrar la lista completa. Podéis redireccionar el comando a un fichero externo para ver mejor los resultados.

Para comprobar que has entendido los cálculos correspondientes al tamaño del MB y al del AI, calcula **sin el programa** los datos del superbloque para las siguientes inicializaciones del sistema de ficheros y luego compruébalos con el programa (entre prueba y prueba recuerda eliminar primero el disco con el comando **$ rm disco**[11] ya que el *open()* del *bmount()* si ya existe el fichero que actúa como dispositivo virtual, simplemente lo abre y no lo crea de nuevo):

$ ./mi_mkfs disco 200000 [12]
$ ./mi_mkfs disco 500000 [13]
$ ./mi_mkfs disco 1000000 [14]

---

[11] Si hacéis un **$ make clean** ya va incluída la eliminación del disco

[12] DATOS DEL SUPERBLOQUE
posPrimerBloqueMB = 1
posUltimoBloqueMB = 25
posPrimerBloqueAI = 26
posUltimoBloqueAI = 6275
posPrimerBloqueDatos = 6276
posUltimoBloqueDatos = 199999
posInodoRaiz = 0
posPrimerInodoLibre = 0
cantBloquesLibres = 193724
cantInodosLibres = 50000
totBloques = 200000
totInodos = 50000

[13] DATOS DEL SUPERBLOQUE
posPrimerBloqueMB = 1
posUltimoBloqueMB = 62
posPrimerBloqueAI = 63
posUltimoBloqueAI = 15687
posPrimerBloqueDatos = 15688
posUltimoBloqueDatos = 499999
posInodoRaiz = 0
posPrimerInodoLibre = 0
cantBloquesLibres = 484312
cantInodosLibres = 125000
totBloques = 500000
totInodos = 125000

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

Anexo

Otra manera de “forzar” que el inodo ocupe 128 bytes es definiendo un tipo creado por el usuario con *typedef* de nombre *inodo_t* (el “_t “ lo veis en muchos tipos estándares y es para indicar que es un "typedef") que sea la union de los campos reales del inodo con una variable de tamaño INODOSIZE. Podéis comprobar con el siguiente programa que así también el inodo ocupa 128 bytes:

```
#include <time.h>   //para los sellos de tiempo
#include <stdio.h>

#define INODOSIZE 128 // tamaño en bytes de un inodo

typedef union _inodo {
  struct {
    unsigned char tipo;     // Tipo ('l':libre, 'd':directorio o 'f':fichero)
    unsigned char permisos; // Permisos (lectura y/o escritura y/o ejecución)
    time_t atime; // Fecha y hora del último acceso a datos: atime
    time_t mtime; // Fecha y hora de la última modificación de datos: mtime
    time_t ctime; // Fecha y hora de la última modificación del inodo: ctime
    unsigned int nlinks;            // Cantidad de enlaces de entradas en directorio
    unsigned int tamEnBytesLog;     // Tamaño en bytes lógicos
    unsigned int numBloquesOcupados; // Cantidad de bloques ocupados zona de datos
    unsigned int punterosDirectos[12];
    unsigned int punterosIndirectos[3];
  };
  char padding[INODOSIZE];
} inodo_t;

void main() {
  inodo_t inodo;
  printf("Size: %ld\n", sizeof(inodo_t));
  return;
}
```