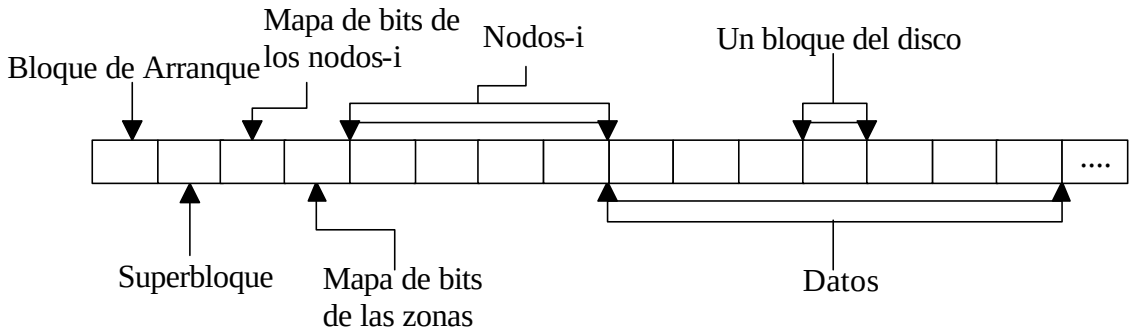
Superbloque

Enrique González Cabrera
David Hernández Cerpa

ESQUEMA DE UN SISTEMA DE ARCHIVOS

Un sistema de archivos MINIX se puede almacenar en cualquier dispositivo de bloques, como un disco flexible o una partición de un disco duro. En todos los casos el sistema de archivos tiene la misma estructura. La siguiente figura muestra este esquema para un disco flexible de 360K con 127 nodos-i y un tamaño de bloque de 1K. Los sistemas de archivos mayores o bien aquellos con más o menos nodos-i o un tamaño de bloque diferente tienen las mismas 6 componentes en el mismo orden, aunque sus tamaños relativos pueden ser distintos.



Bloque de arranque

Cada sistema de archivo comienza con un bloque de arranque, el cual ocupa siempre un sólo bloque. Sólo es necesario en el disco boot del sistema, sin embargo, con el fin de mantener uniforme la estructura del sistema de archivos, todos los dispositivos tienen uno aunque no se utilice. Para identificar el bloque que contiene código ejecutable del que no, existe un número mágico que se sitúa en una posición conocida del bloque. El código de la BIOS al cargar este bloque comprueba que ése número es válido.

Superbloque

Ocupa un solo bloque. Contiene información relativa al tamaño de las distintas partes del sistema de archivos. Se verá en detalle en la siguiente sección.

Mapa de bits de los nodos-i

Se utiliza para llevar el control de los nodos-i libres y ocupados. Por cada nodo-i existe un bit en dicha tabla que indica su estado. Si el bit es 0 significa que el nodo-i está libre, en caso contrario quiere decir que está ocupado.

Mapa de bits de zonas

Se utiliza para llevar el control de las zonas de datos. Su funcionamiento es idéntico al del mapa de bits de los nodos-i.

Nodos-i

Son estructuras de 64 bytes (32 bytes en la versión 1 del sistema de ficheros) en las que se encuentra información de cada fichero físico existente en el sistema de archivos, principalmente la relativa a la situación de las zonas de datos de dicho fichero. El número de

bloques necesarios para almacenar los nodos-i depende del número de nodos-i. Por ejemplo, para 128 nodos-i se necesitan 8 bloques de 1K.

Datos

Es el conjunto de bloques que se utiliza para el almacenamiento de los ficheros.

EL SUPERBLOQUE

El superbloque de un sistema de archivos contiene la información que describe a dicho sistema. Su función principal consiste en indicar al FS el tamaño de las distintas partes del propio sistema de archivos. Los campos de un superbloque se muestran en la siguiente figura. Hay que tener presente que cuando cargamos en memoria el superbloque vamos a tener una serie de campos más que no se almacenan en el disco.

En memoria existe una tabla de estructuras superbloque donde en cada entrada se colocarán los datos del superbloque de cada sistema de ficheros montando en el sistema.

Presente en el disco y en la memoria	ino_t s_ninodes;	Números de nodos i en el sistema de ficheros
	zone1_t s_nzones;	Tamaño del sistema de ficheros en zonas (v1). Incluidos los bitmaps
	short s_imap_blocks;	Número de bloques del mapa de bits de los nodos-i
	short s_zmap_blocks;	Número de bloques del mapa de bits de las zonas
	zone1_t s_firstdatazone;	Número de zona en el que comienza la zona de datos
	short s_log_zone_size;	Log2 bloques por zona. Para calcular mediante bit-shifting los bloques de una zona
	off_t s_max_size;	Tamaño máximo de un archivo
	short s_magic;	Número mágico identificativo del tipo de sistema de ficheros y versión
	short s_pad;	Relleno
	zone_t s_zones;	Número de zonas (v2)
Presente en memoria y no en el disco	struct inode *s_isup;	Apuntador al nodo-i del directorio raíz del sistema de archivos
	struct inode *s_imount;	Apuntador al nodo-i del directorio en el que se ha montado el sistema de archivos
	unsigned s_inodes_per_block;	Nodos-i por bloque
	dev_t s_dev;	Número del dispositivo
	int s_rd_only;	Flag de lectura solamente
	int s_native;	Big-endian flag
	int s_version;	Versión del sistema de ficheros
	int s_ndzones;	Número de entradas de zonas directas por nodo-i
	int s_nindirs;	Número de zonas indirectas por bloque indirecto
	bit_t s_isearch;	Primer bit libre en el mapa de bits de los inode
	bit_t s_zsearch;	Primer bit libre en el mapa de bits de las zonas

s_ninodes

Contiene el número de nodos-i disponibles para los archivos y directorios. El nodo-i 0 está en el disco, pero nunca se utiliza, de forma que siempre está a 1 la posición que le corresponde en el mapa de bits, para que nunca se asigne cuando un nodo-i se solicita. Esto se

usa para que cuando se solicita un nodo-i libre se devuelva el número de dicho nodo si hay alguno libre, o 0 en caso contrario.

Como ejemplo, si este campo contuviera un 4 significaría que tenemos cuatro nodos-i y que se usan cinco bits en el mapa de bits: el bit 0, que siempre es 1 y no se usa, y los bits 1 al 4 para archivos y directorios.

Este campo es necesario para calcular cuántos bloques ocupa el mapa de bits de nodos-i y los propios nodos-i. Dados el tamaño del bloque y el número de nodos-i, es fácil determinar el tamaño del mapa de bits de nodos-i y el número de bloques de nodos-i. Por ejemplo, para un bloque de 1K, cada bloque del mapa de bits tiene 1K bytes (8K bits) y por tanto, puede llevar el control de hasta 8192 nodos-i.

s_nzones

Las operaciones de lectura y escritura son en base a bloques. Sin embargo, el almacenamiento en disco se asigna en unidades (zonas) de 1, 2, 4, 8 o en general 2^n bloques. Por ello, aumentar el tamaño de zona desperdicia espacio en el dispositivo mientras que aumentar el tamaño de bloque desperdicia ancho de banda (se transfieren más datos aunque se necesite leer 1 byte). Este campo contiene el número de zonas y es necesario para saber cuántos bloques ocupa el mapa de bits de zonas y las propias zonas.

s_imap_blocks

Número de bloques ocupados por el mapa de bits de nodos-i.

s_zmap_blocks

Número de bloques ocupados por el mapa de bits de la zona.

s_firstdatazone

Número de la primera zona de datos.

Estos tres últimos campos se pueden obtener a partir de los tres primeros, por lo tanto, información contenida en el superbloque es redundante. Esto es así porque la información a veces se necesita en una forma y a veces en otra. Con 1K dedicado al superbloque tiene sentido almacenar esta información en todas las formas que se necesiten, en vez de tener que volver a calcularla con frecuencia durante la ejecución, ya que hay espacio suficiente.

log_zone_size

El número de bloques por zona no se almacena en el superbloque, ya que nunca se necesita. Todo lo que se hace es el logaritmo en base 2 de la razón zona a bloque, que se utiliza como el número de posiciones a desplazar para convertir zonas en bloques y viceversa. Por ejemplo, con 8 bloques por zona, $\log_2 8=3$, de modo que para hallar la zona que contiene el bloque 128 se desplaza el 128 tres bits a la derecha (que equivale a dividir por 8) obteniéndose la zona 16.

s_max_size

Es el tamaño máximo del fichero.

s_magic

El número mágico es un número que identifica al tipo de sistema de ficheros y su versión. Cuando se va a montar un sistema de ficheros, se comprueba que sea compatible con MINIX examinando el valor de este campo.

s_pad

El relleno se hace añadiendo este campo, evitando así posibles rellenos introducidos por los distintos compiladores.

s_zones

Indica el numero total de zonas. En la version 2 del sistema de ficheros el valor de este campo reemplaza al valor del campo s_nzones.

Hasta aquí hemos especificado aquellos campos del superbloque que se encuentran almacenados en disco. Cuando el superbloque se ubica en memoria no solo se almacenan estos campos sino que se aparecen otros con información adicional que detallamos a continuación:

****s_isup***

Es un puntero al nodo-i del directorio raíz del sistema de archivos, al que este superbloque representa, una vez que ha sido montado.

****simount***

Es un puntero al nodo-i del directorio del cual este sistema de archivo ha sido colgado.

Con el fin de entender mejor el uso de estos dos campos se incluye posteriormente un apartado en el que se explica como realiza MINIX el manejo de rutas y directorios.

s_dev

Dispositivo del cual provino el superbloque. Es necesario por si la información del superbloque es modificada durante su estancia en memoria, por lo que habría que rescribirlo en disco (floppy o disco duro).

s_rd_only

Indica que el sistema de ficheros es de sólo lectura.

s_native

Indica si el sistema es big-endian o little-endian.

s_version

Indica cual es la versión del sistema de ficheros.

s_ndzones

Indica cuantas zonas directas poseen los i-nodes.

s_nindirs

Indica el numero de zonas indirectas que puede tene un bloque indirecto.

s_isearch

Indica cuál es el primer bit libre (su valor es 0) del mapa de bits de los nodos-i. Se usa para acortar en lo posible las búsquedas secuenciales en esta tabla pues el bit libre puede ser el actual o uno cercano.

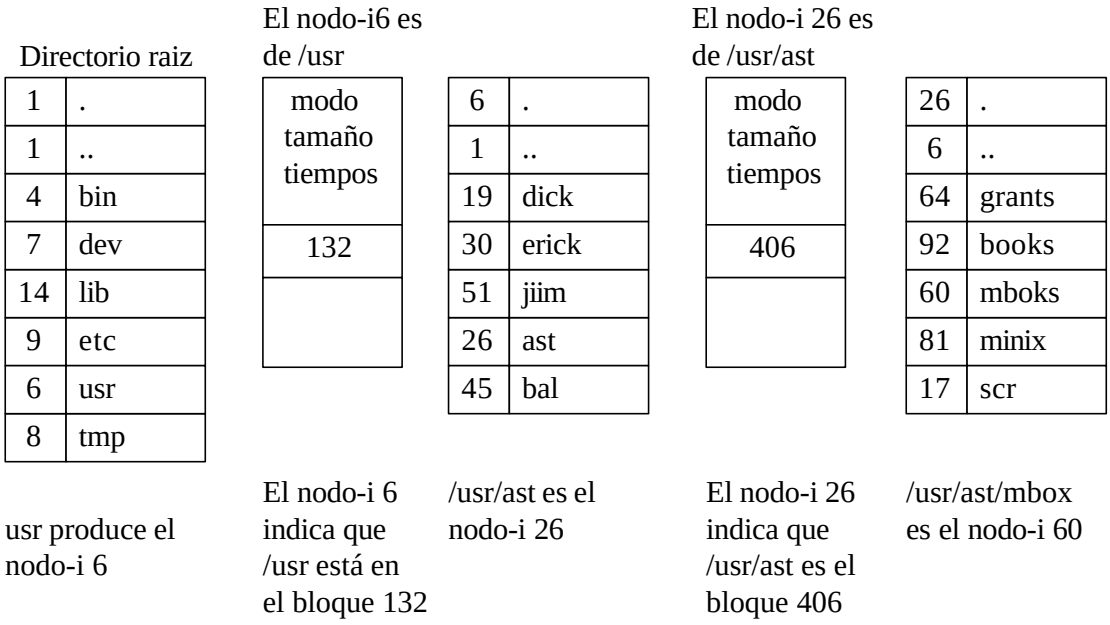
s_zsearch

Indica cuál es el primer bit libre (su valor es 0) del mapa de bits de las zonas. Se usa para acortar en lo posible las búsquedas secuenciales en esta tabla pues el bit libre puede ser el actual o uno cercano.

DIRECTORIOS Y RUTAS

Muchas llamadas al sistema, como OPEN, tienen el nombre de un archivo como parámetro. Lo que en realidad se necesita es el nodo-i de ese archivo, así que es tarea del sistema de archivo buscar el archivo en el árbol del directorio y localizar su nodo-i.

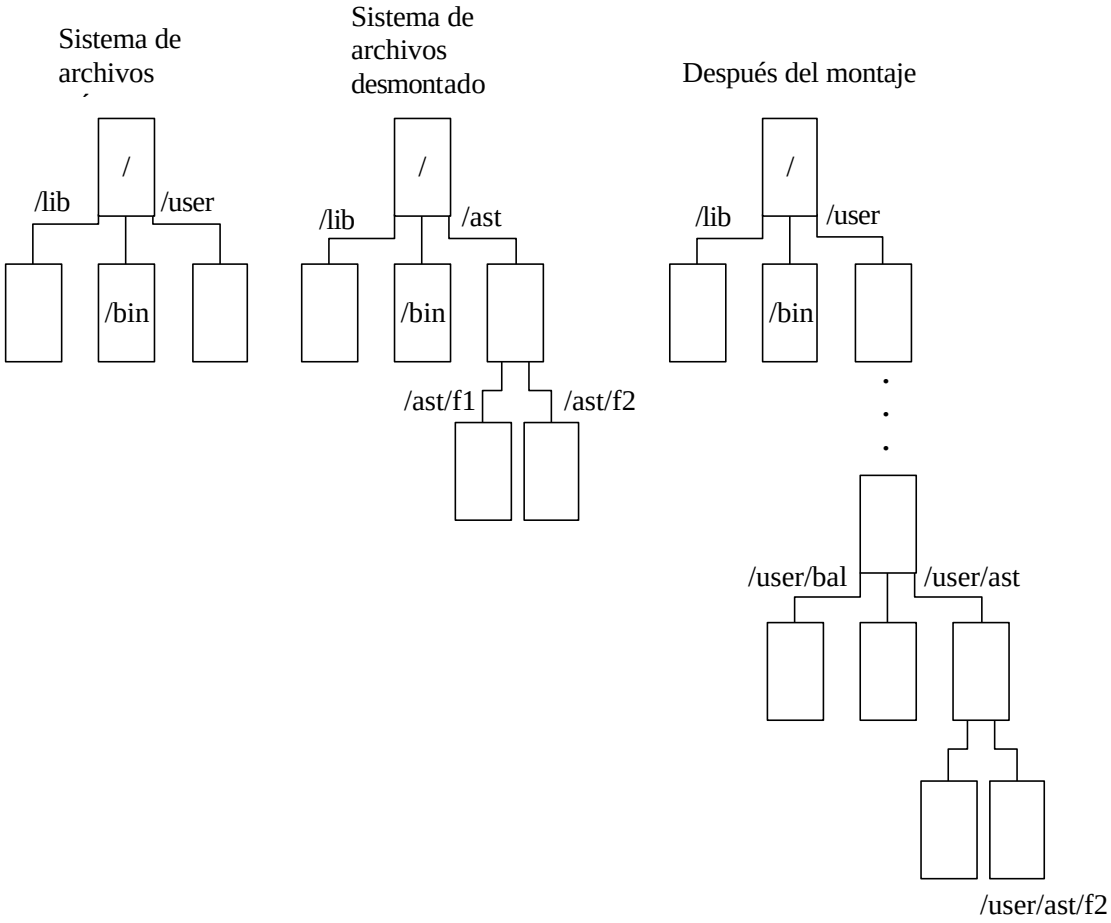
Un directorio de MINIX es un archivo que contiene entradas de 16 bytes. Los dos primeros bytes forman un número de nodo-i de 16 bits y los 14 bytes restantes son el nombre del archivo. Para buscar la ruta */usr/ast/mbox* el sistema primero busca *usr* en el directorio raíz, después busca *ast* en */usr* y por último *mbox* en */usr/ast.* Este proceso se muestra en la siguiente figura:



La única complicación es la que se presenta cuando se encuentra un sistema de archivo montado. Para observar cómo trabaja, debemos apreciar cómo se realiza el montaje. Cuando el usuario teclea el comando:

/etc/mount /dev/fd1/ /user

En el terminal, el sistema de archivos contenido en el disco flexible 1 se monta en el directorio */user* en el sistema de archivo raíz. Los sistemas de archivos antes y después del montaje se muestran en la siguiente figura:



La clave de toda la tarea del montaje es un valor colocado en uno de los campos del nodo-i de */user* después de un montaje óptimo. Este valor indica que se ha montado un sistema de ficheros bajo ese directorio. La llamada a MOUNT también carga el superbloque del nuevo sistema de archivo montado en la tabla *super_block*. Además, coloca el nodo-i raíz del sistema de archivos montado en la tabla *inode*.

Como ya hemos dicho, en los superbloques de la memoria hay dos campos relacionados con los sistemas de archivo montados. El primero de éstos apunta siempre al nodo-i del directorio raíz del sistema de archivos montado. El segundo, el que indica el nodo-i del directorio en el que se ha montado el sistema de ficheros apunta en este caso al nodo-i de */user*.

Cuando se acceda a una ruta como */user/ast/f2*, el sistema de archivo verá el valor comentado en el campo del nodo-i de */user* y entenderá que debe continuar la búsqueda en el nodo-i del directorio raíz del sistema de archivos montado en */user*.

Para encontrar este nodo-i raíz el sistema busca todos los superbloques en la memoria hasta que encuentra aquél cuyo campo *nodo-i del directorio en el que se ha montado* apunta a */user*. Una vez que tiene el superbloque, usa el otro puntero para encontrar el nodo-i del

directorio raíz del sistema de archivos montado. Ahora el sistema de archivo puede proseguir la búsqueda. En este ejemplo, busca *ast* en el directorio del disco flexible 1.

MAPAS DE BITS

Con respecto a los mapas de bits, los procedimientos del fichero super.c sólo se encargan de buscar bits libres y marcarlos como ocupados o libres. Es en otro fichero (cache.c) donde se actualizan los valores *s_zsearch* y *s_isearch* con el fin, tal y como ya se ha comentado, de acortar en lo posible las búsquedas lineales a través de los mapas de bits de un bit libre. En el caso de la liberación de un bit, no es necesario buscar pues se indica qué bit hay que marcar como libre.

Las funciones *alloc_bit* y *free_bit* son llamadas desde *alloc_zone*, *alloc_inode*, *free_zone* y *free_inode*, cuándo necesitan de una zona o un nodo-i. A su vez, estas funciones pueden ser llamadas en diferentes casos. Por ejemplo:

- Cuando se abre un fichero con la llamada al sistema *open*, si se ha especificado el flag O_CREAT, se debe crear el fichero si es que no existe. Esto implica pedir un nodo-i.
- También con la llamada al sistema *open*, si se especifica el flag O_TRUNC, es necesario liberar todas las zonas de datos del fichero.
- Otro ejemplo es cuando se escribe, mediante la llamada al sistema *write*, después del final del fichero. Esto puede requerir una nueva zona de datos.

CÓDIGO FUENTE

ESTRUCTURAS DE DATOS

Las estructuras de datos que utiliza son: **super_block** (descrita en super.h) e **inode** (descrita en inode.h).

La más importante en super.c es **super_block** ya que contiene la tabla del superbloque. El sistema de archivo raíz y todos los sistemas de archivo montados tienen una entrada en dicha tabla.

PROCEDIMIENTOS QUE UTILIZA

- Definidos en fs/cache.c:

get_block (dev, block, only_search):

Lee un bloque desde el dispositivo especificado por dev. Los valores posibles del campo only_search son:

- NORMAL obliga a que se realice la lectura/escritura del disco.
- NOREAD indica que no es necesario leer el bloque del disco porque se va a escribir.

put_block (bp, block_type):

Devuelve un bloque a la lista de bloques disponibles. bp es un puntero al buffer que se libera y block_type indica el tipo de bloque que es (INODE, DIRECTORY, ...).

- Definidos en fs/utility.c:

panic (format, num):

Llamada para abortar el sistema.

conv2(norm,w)

Realiza conversión de w a 16-bits si norm es FALSE.

conv4(norm,x)

Realiza conversión de x a 32-bits si norm es FALSE.

SUPER.H

A continuación mostramos el código del fichero super.h. En él se define la tabla que contendrá los superbloques de los sistemas de ficheros montados. Los diferentes campos ya han sido comentados en las secciones anteriores.

```
EXTERN struct super_block {
  ino_t s_ninodes;    /* # usable inodes on the minor device */
  zone1_t  s_nzones;  /* total device size, including bit maps etc */
  short s_imap_blocks;        /* # of blocks used by inode bit map */
  short s_zmap_blocks;        /* # of blocks used by zone bit map */
  zone1_t s_firstdatazone;    /* number of first data zone */
  short s_log_zone_size;      /* log2 of blocks/zone */
  off_t s_max_size;           /* maximum file size on this device */
  short s_magic;              /* magic number to recognize super-blocks */
  short s_pad;        /* try to avoid compiler-dependent padding */
  zone_t s_zones;             /* number of zones (replaces s_nzones in V2) */

  /* The following items are only used when the super_block is in memory. */
  struct inode *s_isup;               /* inode for root dir of mounted file sys */
  struct inode *s_imount;     /* inode mounted on */
  unsigned s_inodes_per_block;        /* precalculated from magic number */
  dev_t s_dev;        /* whose super block is this? */
  int s_rd_only;              /* set to 1 iff file sys mounted read only */
  int s_native;               /* set to 1 iff not byte swapped file system */
  int s_version;              /* file system version, zero means bad magic */
  int s_ndzones;              /* # direct zones in an inode */
  int s_nindirs;              /* # indirect zones per indirect block */
  bit_t s_isearch;    /* inodes below this bit number are in use */
  bit_t s_zsearch;    /* all zones below this bit number are in use*/
} super_block[NR_SUPERS];

#define NIL_SUPER (struct super_block *) 0
#define IMAP          0       /* operating on the inode bit map */
#define ZMAP          1       /* operating on the zone bit map */
```

SUPER.C

El fichero super.c contiene procedimientos que manejan la tabla del superbloque y las estructuras de datos relacionadas, es decir, los mapas de bits, que llevan el control de qué -i están asignados y cuáles están libres. Cuando se necesita un nuevo nodo-i o zona, se busca una entrada libre en el mapa de bits adecuado.

Contiene los siguientes procedimientos:

- **alloc_bit**: Busca un bit libre en el mapa de bits de las zonas o nodos-i, y devuelve su número de bit.
- **free_bit**: Libera un bit en el mapa de las zonas o del nodos-i.
- **get_super**: Busca un dispositivo en la tabla de superbloques.
- **mounted**: Informa si el nodo-i es un sistema de archivos montado (o ROOT).
- **read_super**: Lee un superbloque desde un dispositivo.

PROCEDIMIENTOS

alloc_bit

Busca un bit del mapa de bits de la zona o del nodo-i libre, y devuelve su número de bit. Cuando se necesita un nodo-i o una zona, se llama a alloc_inode (en inode.c) o a alloc_zone (en cache.c). Estos dos procedimientos llaman a alloc_bit para buscar en realidad en el mapa de bits relevante. En esta búsqueda intervienen tres ciclos anidados:

1) El exterior se repite con todos los bloques de un mapa de bits.
2) El del medio se repite con todas las palabras de un bloque.
3) El interno se repite con todos los bits de una palabra.

El ciclo del medio trabaja observando si la palabra actual es igual al complemento a uno de cero, es decir, una palabra completa llena de unos. Si es así, no tiene nodos-i o zonas libres, de manera que se prueba con la siguiente palabra. Cuando se encuentra una palabra con un valor diferente, ésta debe tener cuando menos un bit 0 en ella, de modo que se mete en el ciclo interno con el fin de hallar el bit libre. Si se han probado todos los bloques sin resultados óptimos, no existen nodos-i o zonas libres, y se devuelve el código NO_BIT (es 0).

Parámetros de entrada:

struct superblock *sp;	Puntero al superbloque del sistema de ficheros.
int map;	Mapa de nodos-i o de zonas
bit_t origin;	Número de bit en el que empezar la búsqueda.

Parámetros de salida:

int b;	Número del bit asignado.
NO_BIT	No existen bits disponibles.

ALGORITMO

```
SI el sistema es de solo lectura ENTONCES panic FINSI
SI  map = Mapa de Inode ENTONCES
        Obtenemos el bloque de comienzo del mapa de bits de nodos-i
        Obtenemos cuántos bits hay en el mapa de bits de nodos-i
        Obtenemos cuántos bloques ocupa el mapa de bits de nodos-i
ELSE  /* Mapa de zona */
        Obtenemos el bloque de comienzo del mapa de bits de zonas
        Obtenemos cuántos bits hay en el mapa de bits de zonas
        Obtenemos cuántos bloques ocupa el mapa de bits de zonas
FINSI

/* comienza la busque a partir de origin */
SI origin apunte fuera del mapa de bits ENTONCES
        oringin=0
FINSI
Calcular bloque de comienzo de la búsqueda
Calcular palabra de comienzo de la búsqueda
```

HACER
    Obtener el bloque del disco que contiene el bloque del bitmap
    Obtener el límite superior del bloque cargado

    PARA cada una de las palabras del bloque HACER
        SI esta palabra no contiene un bit libre ENTONCES
            continuar
        FINSI

        hacer reordenación de bytes de la palabra si es necesaria

        PARA cada bit de la palabra HACER
            SI está libre ENTONCES
                Break
            FINSI

        b=número de bit libre desde el comienzo del bitmap

        SI b sale del limite ENTONCES
            Break
        FINSI

        Marcar el bit como usado
        Hacer reordenación de bytes de la palabra si es necesaria
        Marcar el bloque como DIRTY
        Almacenar el bloque modificado en el disco
        retorna b /*número de bit */
    FINPARA

    Almacenar el bloque modificado en el disco
    Incrementa el bloque
    SI el bloque actual es mayor que el número de bloques ENTONCES
        Bloque actual = 0
    FINSI
    Palabra actual = 0
HASTA que se recorran todos los bloques

```
/*===========================================================================*
 *                          alloc_bit                                        *
 *===========================================================================*/
PUBLIC bit_t alloc_bit(sp, map, origin)
struct super_block *sp;              /* the filesystem to allocate from */
int map;                      /* IMAP (inode map) or ZMAP (zone map) */
bit_t origin;                 /* number of bit to start searching at */
{
```

```
/* Allocate a bit from a bit map and return its bit number. */

  block_t start_block;        /* first bit block */
  bit_t map_bits;             /* how many bits are there in the bit map? */
  unsigned bit_blocks;        /* how many blocks are there in the bit map? */
  unsigned block, word, bcount, wcount;
  struct buf *bp;
  bitchunk_t *wptr, *wlim, k;
  bit_t i, b;

  if (sp->s_rd_only)
       panic("can't allocate bit on read-only filesys.", NO_NUM);

  if (map == IMAP) {
       start_block = SUPER_BLOCK + 1;
       map_bits = sp->s_ninodes + 1;
       bit_blocks = sp->s_imap_blocks;
  } else {
       start_block = SUPER_BLOCK + 1 + sp->s_imap_blocks;
       map_bits = sp->s_zones - (sp->s_firstdatazone - 1);
       bit_blocks = sp->s_zmap_blocks;
  }

  /* Figure out where to start the bit search (depends on 'origin'). */
  if (origin >= map_bits) origin = 0; /* for robustness */

  /* Locate the starting place. */
  block = origin / BITS_PER_BLOCK;
  word = (origin % BITS_PER_BLOCK) / BITCHUNK_BITS;

  /* Iterate over all blocks plus one, because we start in the middle. */
  bcount = bit_blocks + 1;
  do {
       bp = get_block(sp->s_dev, start_block + block, NORMAL);
       wlim = &bp->b_bitmap[BITMAP_CHUNKS];

       /* Iterate over the words in block. */
       for (wptr = &bp->b_bitmap[word]; wptr < wlim; wptr++) {

              /* Does this word contain a free bit? */
              if (*wptr == (bitchunk_t) ~0) continue;

              /* Find and allocate the free bit. */
              k = conv2(sp->s_native, (int) *wptr);
              for (i = 0; (k & (1 << i)) != 0; ++i) {}

              /* Bit number from the start of the bit map. */
              b = ((bit_t) block * BITS_PER_BLOCK)
                  + (wptr - &bp->b_bitmap[0]) * BITCHUNK_BITS
                  + i;

              /* Don't allocate bits beyond the end of the map. */
              if (b >= map_bits) break;

              /* Allocate and return bit number. */
              k |= 1 << i;
              *wptr = conv2(sp->s_native, (int) k);
```

```
                bp->b_dirt = DIRTY;
                put_block(bp, MAP_BLOCK);
                return(b);
        }
        put_block(bp, MAP_BLOCK);
        if (++block >= bit_blocks) block = 0; /* last block, wrap around */
        word = 0;
  } while (--bcount > 0);
  return(NO_BIT);             /* no bit could be allocated */
}
```

free_bit

Libera un bit del mapa de bits de la zona o del nodo-i. Es decir, marca un determinado bit del mapa de bits como libre.

La liberación de un bit es más simple que la asignación, ya que no se necesita realizar una búsqueda. El procedimiento free_bit determina qué bloque del mapa de bits contiene el bit a liberar y pone el bit adecuado a 0.

Parámetros de entrada:

struct superblock *sp;	Puntero al superbloque del sistema de ficheros.
int map;	Mapa de nodos-i o de zonas
bit_nr bit_returned;	Número de bit a liberar en el mapa

Parámetros de salida:

No tiene.

ALGORITMO

SI sistema de ficheros es de solo lectura ENTONCES panic

Obtener la dirección de comienzo del bitmap adecuado.
Obtener la posición del bit en relación al número de bloque, palabra y bit dentro de palabra

Obtener el bloque afectado desde el disco

Hacer reordenación de bytes de la palabra si es necesaria

SI estaba a cero ENTONCES panic.

Marcar el bit como libre

Hacer reordenación de bytes de la palabra si es necesaria

Marcar dicho buffer como Dirty.
Almacenar el bloque modificado en el disco

```
/*===========================================================================*
 *                           free_bit                                        *
 *===========================================================================*/
PUBLIC void free_bit(sp, map, bit_returned)
struct super_block *sp;              /* the filesystem to operate on */
int map;                        /* IMAP (inode map) or ZMAP (zone map) */
bit_t bit_returned;             /* number of bit to insert into the map */
{
/* Return a zone or inode by turning off its bitmap bit. */

  unsigned block, word, bit;
  struct buf *bp;
  bitchunk_t k, mask;
  block_t start_block;

  if (sp->s_rd_only)
       panic("can't free bit on read-only filesys.", NO_NUM);

  if (map == IMAP) {
       start_block = SUPER_BLOCK + 1;
  } else {
       start_block = SUPER_BLOCK + 1 + sp->s_imap_blocks;
  }
  block = bit_returned / BITS_PER_BLOCK;
  word = (bit_returned % BITS_PER_BLOCK) / BITCHUNK_BITS;
  bit = bit_returned % BITCHUNK_BITS;
  mask = 1 << bit;

  bp = get_block(sp->s_dev, start_block + block, NORMAL);

  k = conv2(sp->s_native, (int) bp->b_bitmap[word]);
  if (!(k & mask)) {
       panic(map == IMAP ? "tried to free unused inode" :
             "tried to free unused block", NO_NUM);
  }

  k &= ~mask;
  bp->b_bitmap[word] = conv2(sp->s_native, (int) k);
  bp->b_dirt = DIRTY;

  put_block(bp, MAP_BLOCK);
}
```

get_super

Se utiliza para buscar un dispositivo específico en la tabla del superbloque. Por ejemplo, cuando se va a montar un sistema de archivo, se necesita verificar que no está montado ya. Esta verificación se puede realizar pidiendo a get_super que encuentre el dispositivo del sistema de archivo. Si éste no halla el dispositivo, entonces no es
archivo.

Parámetro de entrada:

dev_t dev; Número de dispositivo cuyo superbloque se busca.

Parámetro de salida:

Devuelve un puntero al superbloque buscado.

Procedimiento que utiliza:

panic(format, num).

ALGORITMO

Barrer toda la tabla de superbloques buscando el que tiene asociado dicho dispositivo.
Si no lo encuentra, entonces panic().
Si lo encuentra, devuelve un puntero al superbloque asociado con el dispositivo.

```
/*===========================================================================*
 *                              get_super                                    *
 *===========================================================================*/
PUBLIC struct super_block *get_super(dev)
dev_t dev;                    /* device number whose super_block is sought */
{
/* Search the superblock table for this device.  It is supposed to be there. */

  register struct super_block *sp;

  for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++)
       if (sp->s_dev == dev) return(sp);

  /* Search failed.  Something wrong. */
  panic("can't find superblock for device (in decimal)", (int) dev);

  return(NIL_SUPER);          /* to keep the compiler and lint quiet */
}
```

mounted

Informa si un determinado nodo-i es un sistema de archivos montado (o ROOT).

Parámetro de entrada:

register struct inode *rip; Puntero al nodo-i.

Parámetro de salida:

Devuelve TRUE en caso de que esté montado.

ALGORITMO

Obtener el dispositivo que se encuentra sobre el nodo-i.
Si es el dispositivo raíz devolver TRUE.
Barrer toda la tabla de superbloques buscando el que tiene asociado dicho dispositivo.
Si lo encontró entonces devolver TRUE sino devolver FALSE.

```
/*===========================================================================*
 *                              mounted                                      *
 *===========================================================================*/
PUBLIC int mounted(rip)
register struct inode *rip;   /* pointer to inode */
{
/* Report on whether the given inode is on a mounted (or ROOT) file system. */
  register struct super_block *sp;
```

```
  register dev_t dev;

  dev = (dev_t) rip->i_zone[0];
  if (dev == ROOT_DEV) return(TRUE);  /* inode is on root file system */

  for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++)
       if (sp->s_dev == dev) return(TRUE);

  return(FALSE);
}
```

read_super

Lee un superbloque desde el disco colocando en memoria estos datos. Si es necesario, se reordenan los bytes en memoria (BIG_ENDIAN flag). También rellena el resto de los campos que no se leen desde el disco.

La escritura en el superbloque no es necesaria ya que es una operación realizado por otros procedimientos (formateo del sistema de ficheros).

Parámetros de entrada:

register struct super_block *sp; Puntero a un superbloque.

Parámetro de salida:

Un entero que indica el código de error o éxito.

Procedimientos que utiliza:

get_block(dev, block, only_search).
put_block (bp, block_type).
copy (dev, source, bytes).

ALGORITMO

Guardar el número de dispositivo, ya que se puede perder posteriormente
Cargar el superbloque desde disco
Copiar el bloque leído de disco en la estructura pasada p
Escribir el bloque en el disco

A partir del número mágico determinar la versión del sistema de ficheros y si es necesaria la reordenación de los bytes en memoria.

Reordenar los campos del superbloque en memoria si es necesario

Asignar a los campos dependientes de la versión del sistema de ficheros los valores adecuados

Rellenar el resto de campos

Hacer algunas comprobaciones básicas para comprobar que el superbloque está bien

```
/*===========================================================================*
 *                              read_super                                   *
 *===========================================================================*/
PUBLIC int read_super(sp)
register struct super_block *sp; /* pointer to a superblock */
{
/* Read a superblock. */

  register struct buf *bp;
  dev_t dev;
  int magic;
  int version, native;

  dev = sp->s_dev;            /* save device (will be overwritten by copy) */
  bp = get_block(sp->s_dev, SUPER_BLOCK, NORMAL);
  memcpy( (char *) sp, bp->b_data, (size_t) SUPER_SIZE);
  put_block(bp, ZUPER_BLOCK);
  sp->s_dev = NO_DEV;         /* restore later */
  magic = sp->s_magic;        /* determines file system type */

  /* Get file system version and type. */
  if (magic == SUPER_MAGIC || magic == conv2(BYTE_SWAP, SUPER_MAGIC)) {
       version = V1;
       native  = (magic == SUPER_MAGIC);
  } else if (magic == SUPER_V2 || magic == conv2(BYTE_SWAP, SUPER_V2)) {
       version = V2;
       native  = (magic == SUPER_V2);
  } else {
       return(EINVAL);
  }

  /* If the super block has the wrong byte order, swap the fields; the magic
   * number doesn't need conversion. */
  sp->s_ninodes =       conv2(native, (int) sp->s_ninodes);
  sp->s_nzones =        conv2(native, (int) sp->s_nzones);
  sp->s_imap_blocks =   conv2(native, (int) sp->s_imap_blocks);
  sp->s_zmap_blocks =   conv2(native, (int) sp->s_zmap_blocks);
  sp->s_firstdatazone = conv2(native, (int) sp->s_firstdatazone);
  sp->s_log_zone_size = conv2(native, (int) sp->s_log_zone_size);
  sp->s_max_size =      conv4(native, sp->s_max_size);
  sp->s_zones =         conv4(native, sp->s_zones);

  /* In V1, the device size was kept in a short, s_nzones, which limited
   * devices to 32K zones.  For V2, it was decided to keep the size as a
   * long.  However, just changing s_nzones to a long would not work, since
   * then the position of s_magic in the super block would not be the same
   * in V1 and V2 file systems, and there would be no way to tell whether
   * a newly mounted file system was V1 or V2.  The solution was to introduce
   * a new variable, s_zones, and copy the size there.
   *
   * Calculate some other numbers that depend on the version here too, to
   * hide some of the differences.
   */
  if (version == V1) {
       sp->s_zones = sp->s_nzones;   /* only V1 needs this copy */
       sp->s_inodes_per_block = V1_INODES_PER_BLOCK;
       sp->s_ndzones = V1_NR_DZONES;
```

```
        sp->s_nindirs = V1_INDIRECTS;
  } else {
        sp->s_inodes_per_block = V2_INODES_PER_BLOCK;
        sp->s_ndzones = V2_NR_DZONES;
        sp->s_nindirs = V2_INDIRECTS;
  }

  sp->s_isearch = 0;          /* inode searches initially start at 0 */
  sp->s_zsearch = 0;          /* zone searches initially start at 0 */
  sp->s_version = version;
  sp->s_native  = native;

  /* Make a few basic checks to see if super block looks reasonable. */
  if (sp->s_imap_blocks < 1 || sp->s_zmap_blocks < 1
                            || sp->s_ninodes < 1 || sp->s_zones < 1
                            || (unsigned) sp->s_log_zone_size > 4) {
        return(EINVAL);
  }
  sp->s_dev = dev;            /* restore device number */
  return(OK);
}
```