A.S.O:

CACHE

Alejandro Valido Hernández
César J. Caraballo Viña

Indice.

1. Cache.c

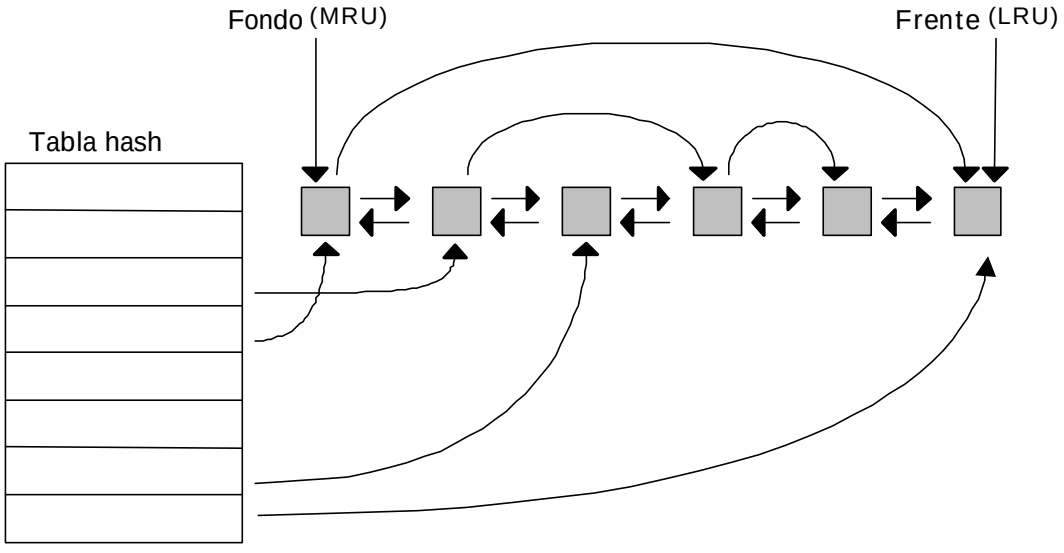
1.1. Introducción.

MINIX utiliza una caché para reducir el número de accesos a disco y de esta forma mejorar el rendimiento del sistema. Esta caché se implementa como un vector de buffers, cada uno de los cuales consta de un encabezado (donde se encuentran los punteros, contadores, etc., que son necesarios para mantener la lista) y espacio para almacenar un bloque del disco.

Todos los bloques se enlazan en una lista doblemente encadenada que va desde los bloques más recientemente usados (MRU) hasta los menos recientemente usados (LRU). Esta lista se utiliza para seleccionar la víctima en caso de que haya que desalojar una posición de la cache.

Además, para poder determinar de manera rápida si un bloque está en la caché o no, se utiliza una tabla hash, de forma que todos los bloques que tienen el código hash ***k*** se encuentran encadenados en una lista apuntada por la entrada ***k*** de la tabla hash. La función hash aquí usada simplemente extrae los *n* bits menos significativos del número de bloque, de manera que los bloques de distintos dispositivos aparezcan en la misma lista encadenada de hash.

Por tanto, cada buffer pertenecerá a dos listas encadenadas: por un lado, a la lista doblemente encadenada que contiene todos los buffers ordenados por su uso más o menos reciente; y, por otro, a su lista hash correspondiente.



1.2. Estructuras y tipos de datos.

```
EXTERN struct buf {
            /* Zona de datos de cada buffer */
       union{
              char b_data[BLOCK_SIZE];          /* datos de usuario */
              dir_struct b_dir[NR_ENTRIES];           /* bloque de directorio*/
              zone_nr b_ind[NR_INDIRECTS];            /* bloque indirecto */
              d_inode b_inode[INODES_PER_BLOCK];      /* nodo_i */
              int b_int[INTS_PER_BLOCK];        /* bloque de enteros */
       }b;
           /* Cabecera de cada buffer*/
       struct buf *b_next;
       struct buf *b_prev; /* usados para encadenar doblemente los buffers */
       struct buf *b_hash; /* lista encadenada hash */
       block_nr b_blocknr; /* número de bloque en el dispositivo */
       dev_t b_dev;        /* dispositivo en el que está el bloque */
       char b_dirt;        /* CLEAN o DIRTY */
       char b_count;       /* número de usuarios del buffer */
} buf[NR_BUFS];
```

Cada buffer consta de dos partes:

Cabecera: en ella se encuentra la siguiente información:

b_next, *b_prev*: punteros al predecesor y al sucesor, respectivamente, de un buffer dentro de la lista doblemente encadenada.

b_hash: puntero por el que se encadenan las listas de hash para el manejo de colisiones.

b_blocknr: número del bloque en el dispositivo.

b_dev: dispositivo donde se encuentra el bloque.

b_count: contador del número de usuarios del buffer.

b_dirt: variable que indica si el bloque ha sido modificado.

Datos del buffer: en él podemos encontrar un bloque de datos del usuario, un bloque de directorio, un bloque indirecto o un nodo-i.

BUF[NR_BUFS]: Lista de bloques doblemente encadenada, ordenada de tal manera que los bloques que se hayan usado más recientemente (MRU) se colocarán en el fondo de la lista (**rear**) y los que haga más tiempo que no se utilizan (LRU), se colocarán en el frente (**front**).

NR_BUFS: Número de bloques en la caché. Está definido como 32.

```
/* Definición de la tabla hash. Vector de punteros a la estructura buf */
EXTERN struct buf *buf_hash[NR_BUF_HASH];
```

BUF_HASH[NR_BUF_HASH]: Vector de punteros a la lista de bloques. Cada elemento apunta a una lista encadenada en la cual todos los números de los bloques terminan en la misma cadena de *n* bits. El campo que sirve de enlace en estas cadenas es **b_hash**.

NR_BUF_HASH: Tamaño de la tabla hash. Tal y como se ha definido la función hash, el tamaño de la tabla debe ser de 2^n elementos, siendo *n* el número de bits que se extraen

del número de bloque. Como en este caso la función de hash extrae 5 bits, esta constante vale 32.

```
EXTERN struct buf *front;  /* puntero al bloque menos recientemente usado */
EXTERN struct buf *rear;   /* puntero al bloque más recientemente usado */

EXTERN int bufs_in_use;          /* número de buffers en uso */
```

Las variables comunes más importantes que se manejan son:

front: apunta al bloque usado menos recientemente.
rear: apunta al bloque usado más recientemente.
bufs_in_use: número de buffers actualmente en uso.

A continuación se pueden ver las macros más utilizados e importantes:

```
#define NIL_BUF (struct buf *) 0 /* ausencia de buffer */

/* Asignación de un nombre más sencillo a los campos de la estructura buf;
permite usar      bp->b_data en vez de bp->b.data */
#define b_data      b.b_data
#define b_dir       b.b_dir
#define b_ind       b.b_ind
#define b_inode     b.b_inode
#define b_int       b.b_int


#define WRITE_IMMED       0100   /* debe ser escrito inmediatamente */

#define ONE_SHOT                0200         /* El bloque probablemente no
                                                se necesitará pronto */

/* Tipos de bloques */
#define INODE_BLOCK       (0+ MAYBE_WRITE_IMMED)
#define DIRECTORY_BLOCK   (1+ MAYBE_WRITE_IMMED)
#define INDIRECT_BLOCK    (2+ MAYBE_WRITE_IMMED)
#define I_MAP_BLOCK       (3+ MAYBE_WRITE_IMMED)
#define ZMAP_BLOCK        (4+ WRITE_IMMED + ONE_SHOT)
#define ZUPER_BLOCK       (5+ WRITE_IMMED + ONE_SHOT)
#define FULL_DATA_BLOCK           6
#define PARTIAL_DATA_BLOCK  7


/* Posibles valores de 'only_search' */
#define NORMAL      0     /* Fuerza a get_block a hacer una  operación de
lectura */
#define NO_READ     1     /* Previene a get_block de hacer una lectura de
disco */
#define PREFETCH    2     /* Indica a get_block que no lea ni marque dev */


#define NO_BIT      (bit_nr) 0   /*Devuelto por alloc_bit indicando fallo */


#define CLEAN 0     /* Las copias de memoria y disco son idénticas */
#define DIRTY 1     /* Las copias de memoria y disco son diferentes*/

#define NO_DEV      (dev_nr) ~0         /* indica la ausencia de un número de
dispositivo */
```

1.3. GET_BLOCK()

Este procedimiento es usado por el sistema de ficheros (fs) para adquirir bloques de datos. En primer lugar, se hace una búsqueda en la caché para ver si el bloque solicitado se encuentra allí. Pueden presentarse dos situaciones:

1.- El bloque está en la caché: se incrementa el contador de número de usuarios de dicho bloque y se devuelve un puntero con la dirección del bloque.

2.- El bloque no está en la caché: es necesario eliminar el bloque menos recientemente usado, retirándolo de la lista de hash y grabándolo en disco si ha sido modificado (DIRTY). Una vez que el buffer esté disponible, se cargan los parámetros del nuevo bloque, que se leerá de disco *salvo que vayamos a hacer una reescritura completa del mismo*, ya que sería una pérdida de tiempo leer la versión antigua del bloque.

Se devuelve un puntero al bloque solicitado.

Parámetros de entrada

dev : Dispositivo donde se encuentra el bloque solicitado.
block : Número de bloque requerido.
only_search : Forma de actuar respecto al dispositivo. Posibles valores:
NORMAL: fuerza a get_block a hacer una operación de lectura.
NO_READ: previene a get_block de hacer una lectura de disco.
PREFETCH: indica a get_block que no lea ni marque el dispositivo.

Parámetros de salida

bp : Puntero al bloque solicitado.

CÓDIGO:

```
PUBLIC struct buf *get_block(dev, block, only_search)
register dev_t dev;
register block_t block;
int only_search;
{

  int b;
  register struct buf *bp, *prev_ptr;

     /* Busca en la cadena hash por (dev,block) */
  if (dev != NO_DEV) {
       b = (int) block & HASH_MASK;   /* aplicación de la función hash */
       bp = buf_hash[b];
       while (bp != NIL_BUF) {
              if (bp->b_blocknr == block && bp->b_dev == dev) {
                     /* Se ha encontrado el bloque buscado */
                     if (bp->b_count == 0)
                            rm_lru(bp);    /* elimina el bloque de la cadena */
                            bp->b_count++; /* bloque tiene un usuario más */
                            return(bp);
              } else {
```

```
                    bp = bp->b_hash;      /* Avanza en la lista */

              }
        }
  }

/* El bloque deseado no está disponible en la cadena, por lo que tenemos que
buscar un buffer en la cache para traer el bloque. Tomamos de la lista LRU el
bloque más viejo y que no está en uso (b_count=0). Si todos los buffers están
en uso, entonces PÁNICO. Si no, incrementamos el número de buffers que se
están utilizando */
  if ((bp = front) == NIL_BUF) panic("all buffers in use", NR_BUFS);
  rm_lru(bp);

  /* Eliminamos el bloque elegido como víctima de su cadena HASH */
  b = (int) bp->b_blocknr & HASH_MASK;
  prev_ptr = buf_hash[b];
  if (prev_ptr == bp) {  /* si es el primero de su lista hash ... */
       buf_hash[b] = bp->b_hash;
  } else {
       while (prev_ptr->b_hash != NIL_BUF)
              if (prev_ptr->b_hash == bp) {
                     prev_ptr->b_hash = bp->b_hash;   /* lo encontramos */
                     break;
              } else {
                     prev_ptr = prev_ptr->b_hash;           /* avanzamos */
              }
  }


  /* Si el bloque víctima había sido modificado (DIRTY), se salva en el disco
y se aprovecha para escribir todos los bloques del mismo dispositivo que se
hubieran modificado. */
  if (bp->b_dev != NO_DEV) {
       if (bp->b_dirt == DIRTY) flushall(bp->b_dev);

/* Si caché secundaria disponible, inserta también el bloque en ella */
#if ENABLE_CACHE2
       put_block2(bp);
#endif
  }

/* Rellena parámetros de bloque y lo añade a su lista encadenada HASH */
  bp->b_dev = dev;        /* número de dispositivo */
  bp->b_blocknr = block;  /* número de bloque */
  bp->b_count++;                /* incrementamos número de usuarios */
  b = (int) bp->b_blocknr & HASH_MASK;
  bp->b_hash = buf_hash[b];
  buf_hash[b] = bp;       /* lo añadimos a la lista hash*/


/* Vamos a buscar el bloque a menos que 'only_search' <> NORMAL */
  if (dev != NO_DEV) {
#if ENABLE_CACHE2
       if (get_block2(bp, only_search)) ;  /* en la caché secundaria */
       else
#endif
       if (only_search == PREFETCH) bp->b_dev = NO_DEV;
       else
       if (only_search == NORMAL) rw_block(bp, READING);
  }
  return(bp);             /* Devuelve puntero al bloque adquirido */
}
```

1.4. PUT_BLOCK()

Cuando el procedimiento que solicitó el bloque ya lo ha utilizado, se llama a este procedimiento para liberar dicho bloque. Uno de los parámetros de entrada (block_type) indica el tipo de bloque (datos, nodo-i, directorio, etc.) que se está liberando. Según este parámetro se deciden dos cosas:

1.- Si el bloque se debe colocar en el frente o en el fondo de la lista doblemente encadenada (*ONE_SHOT*). Los bloques que probablemente no se vuelvan a necesitar pronto, se colocan en el frente de la lista (LRU), de manera que serán las próximas víctimas cuando se necesite liberar un buffer. Aquellos bloques que muy probablemente se vayan a necesitar pronto, pasan al fondo de la lista (MRU), de manera que sean los últimos en utilizarse como víctimas.

2.- Si el bloque se debe escribir de inmediato en el disco o no (*WRITE_INMED*). Se escribirán de forma inmediata aquellos bloques que se han modificado y que además contienen información básica, cuya pérdida afectaría directamente a la integridad del sistema de ficheros, o sea, nodos_i, bloques de directorio, mapas de bits, etc.

Nótese que un bloque de datos ordinario que se ha modificado no se reescribe en el disco hasta que llegue al 'front' de la lista doblemente encadenada y se le tome como víctima o hasta que se ejecute una llamada al sistema *sync*.

Parámetros de entrada

bp : Puntero al bloque a liberar.
block_type : Indica el tipo de bloque que se va a liberar.

CÓDIGO:

```
PUBLIC void put_block(bp, block_type)
register struct buf *bp;
int block_type;
{
       register struct buf *next_ptr, * prev_ptr;

/* Si bloque nulo, retornar. Es más cómodo chequearlo aquí que en el
procedimiento que llama */
if (bp == NIL_BUF) return;

  bp->b_count--; /* decrementamos el número de usuarios del bloque */
  if (bp->b_count != 0) return;  /* aún está en uso */

  bufs_in_use--;          /* decrementamos el número bloques en uso */

/* Situamos el bloque al principio o al fin de la lista doblemente encadenada
según el tipo de bloque (en frente si no esperamos utilizarlo pronto, en el
fondo si lo esperamos) */
  if (block_type & ONE_SHOT) {
       /* No necesitamos el bloque pronto; lo ponemos en el frente */
       bp->b_prev = NIL_BUF;
       bp->b_next = front;
       if (front == NIL_BUF)
              rear = bp;   /* La lista LRU está vacía */
       else
              front->b_prev = bp;
       front = bp;
```

```
  } else {
       /* Necesitamos el bloque pronto; lo ponemos en el fondo */
       bp->b_prev = rear;
       bp->b_next = NIL_BUF;
       if (rear == NIL_BUF)
              front = bp;
       else
              rear->b_next = bp;
       rear = bp;
  }

  /* Si los bloques son de escritura inmediata y se han modificado, escribirlo
en el dispositivo */
  if ((block_type & WRITE_IMMED) && bp->b_dirt==DIRTY && bp->b_dev != NO_DEV)
       rw_block(bp, WRITING);
}
```

1.5. ALLOC_ZONE()

A medida que un archivo crece, de cuando en cuando se debe asignar una zona para contener los nuevos datos. Para ello, se busca una zona libre en el mapa de bits del dispositivo procurando que esté próxima a la zona 0 del fichero actual, de forma que esté almacenado de la forma más compacta posible.

Parámetros de entrada

dev : Dispositivo donde se ubicará la nueva zona.
z : Se buscará una zona lo más cercana posible a la indicada por z.

Parámetros de salida

Esta función puede devolver:

NO_ZONE cuando no se puede asignar una zona a ese fichero, o el número de la zona asignada.

CÓDIGO:

```
PUBLIC zone_nr alloc_zone(dev, z)
dev_t dev;
zone_nr z;
{
       bit_nr b, bit;
       struct super_block *sp;
       int major, minor;

       /* Encuentra el superbloque del dispositivo */
       sp = get_super(dev);

/* Transforma la zona a número de bit */

/* Si z=0 nos olvidamos de la parte inicial del mapa, sabiendo que está
completamente en uso */
  if (z == sp->s_firstdatazone) {
       bit = sp->s_zsearch;
  } else {
       bit = (bit_t) z - (sp->s_firstdatazone - 1);
  }
```

```
/* Busca una zona próxima a la dada */

       b = alloc_bit(sp, ZMAP, bit);

/* Si no se encontró dicha zona, se advierte con un mensaje, indicando el
número del dispositivo, y se devuelve NO_ZONE */

       if (b == NO_BIT)
       {
              err_code = ENOSPC;
              major = (int) (sp->s_dev >> MAJOR) & BYTE;
              minor = (int) (sp->s_dev >> MINOR) & BYTE;
              printf("No space on %sdevice &d/&d\n",
                     sp->s_dev==ROOT_DEV ? "root" : "", major, minor);
              return(NO_ZONE);
       }

/* Se devuelve el número de la zona asignada */
       if (z == sp->s_firstdatazone) sp->s_zsearch = b;  /* para la próxima vez
*/
       return( sp->s_firstdatazone - 1 + (zone_nr) b);
}
```

1.6. FREE_ZONE()

Su función es devolver al mapa de bits aquellas zonas que pertenecen a un archivo que se suprime. Todo lo que hace es llamar a *free_bit* pasándole el mapa de zonas y el número de bits como parámetros. *Free_bit* también se utiliza para devolver nodos-i libres, pero en ese caso el primer parámetro debe ser el mapa de nodos-i.

Parámetros de entrada

dev : Dispositivo al que pertenece la zona.
numb : Número de la zona que se va a liberar.

CÓDIGO:

```
PUBLIC void free_zone(dev,numb)
dev_t dev;
zone_nr numb;
{
  register struct super_block *sp;
  bit_t bit;

  /* Colocamos el  super_block apropiado y retornamos el bit */
  sp = get_super(dev);
     /* Comprobamos la validez de la zona */
  if (numb < sp->s_firstdatazone || numb >= sp->s_zones) return;
  bit = (bit_t) (numb - (sp->s_firstdatazone - 1));
  free_bit(sp, ZMAP, bit);
  if (bit < sp->s_zsearch) sp->s_zsearch = bit;
}
```

1.7. RW_BLOCK()

Este procedimiento actúa como interfaz entre el disco y la memoria. Realiza las operaciones reales de lectura y escritura de un bloque en disco. Ésta es la única rutina en la que se invoca realmente al disco. Si ocurre un error, se escribe un mensaje, pero no se informa al procedimiento que invocó esta función (de todas maneras, si el error ocurrió mientras se reescribía un bloque de la caché en el disco, no está claro qué habría podido hacer el procedimiento llamador si se le hubiese informado).

Parámetros de entrada

bp : Puntero al buffer.
rw_flag : Indica si la operación a realizar es una lectura o una escritura.

CÓDIGO:

```
PUBLIC void rw_block(bp, rw_flag)
register struct buf *bp;
int rw_flag;
{
	int r,op;
	off_t pos;
	dev_t dev;

	/* Si el bloque pertenece a un dispositivo, calcula la posición a partir
	de la cual se va a realizar la operación */
	if ((dev = bp->b_dev) != NO_DEV)
	{
		pos = (off_t) bp->b_blocknr * BLOCK_SIZE;

		/* Realiza la lectura o escritura física */
		op = (rw_flag == READING ? DEV_READ : DEV_WRITE);
		r=dev_io(op,FALSE,dev,pos,LOCK_SIZE,FS_PROC_NR,bp-> b_data);


		/* Si no se leyó el bloque completo, comprueba si se acabó el
		fichero. Si no es así, se trata de un error y se escribe un
		mensaje */
		if (r!=BLOCK_SIZE)
		{
			if (r >= 0) r = END_OF_FILE;
			if (r!= END_OF_FILE)
				printf("Unrecoverable disk error on device &d/&d,
				block %u\n"), (dev>>MAJOR)&BYTE, (dev>>MINOR)&BYTE,
				bp->b_blocknr);

			bp->b_dev = NO_DEV;	/* Se marca el bloque como
inválido */

			/* Se devuelve un código de error */
			if (rw_flag == READING) rdwt_err = r;
		}
	}

	/* Se marca el bloque como no modificado */
	bp->b_dirt = CLEAN;
}
```

1.8. INVALIDATE()

Marca en la caché todos los bloques que pertenecen a un dispositivo como no válidos. Por ejemplo, se le llama cuando se desmonta un disco para retirar de la cache todos los bloques que pertenecen al sistema de archivo recién desmontado. Si esto no se hiciera, cuando se volviese a utilizar el dispositivo con un disco flexible diferente, el sistema de archivo podría hallar en la caché los bloques anteriores en lugar de los nuevos y usarlos equivocadamente.

Parámetros de entrada

device : Dispositivo cuyos bloques se van a invalidar

CÓDIGO:

```
PUBLIC void invalidate(device)
dev_t device;
{
	register struct buf *bp;

	for ( bp = &buf[0] ; bp < &buf[NR_BUFS] ; bp++)
		if ( bp->b_dev == device ) bp->b_dev = NO_DEV;

	/* Si hay caché secundaria, también se invalidan los bloques de este
dispositivo en ella */
	#if ENABLE_CACHE2
		invalidate2(device);
	#endif

}
```

1.9. FLUSHALL()

Escribe en disco todos los bloques de un cierto dispositivo que hayan sido modificados (DIRTY).

Parámetros de entrada

dev : Dispositivo cuyos bloques se van a escribir en disco.

CÓDIGO:

```
PUBLIC void flushall(dev)
dev_t dev;
{
	register struct buf *bp;
	static struct buf *dirty[NR_BUFS];
	int ndirty;

/* Recorre todos los buffers e introduce en un vector (dirty) aquellos que
pertenezcan al dispositivo en cuestión y hayan sido modificados */
	for ( bp = &buf[0], ndirty = 0; bp < &buf[NR_BUFS]; bp++)
		if ( bp->b_dirt == DIRTY  &&  bp->b_dev == device )
			dirty[ndirty++] = bp;

/* Realiza una lectura/escritura múltiple */
	rw_scattered(dev, dirty, ndirty, WRITING);
}
```

1.10. RW_SCATTERED()

Este procedimiento realiza una lectura/escritura múltiple. Primero ordena las peticiones de menor a mayor. Si el dispositivo nos ofrece la posibilidad de realizar entrada/salida múltiple, utilizaremos el procedimiento que ya existe para tal fin; en caso contrario, realiza las operaciones una a una.

Parámetros de entrada

dev : Dispositivo en el que se va a realizar la operación.
bufq : Puntero al array de buffers donde se encuentran las peticiones.
bufqsize : Indica el número de peticiones.
rw_flag : Indica si la operación es una lectura o una escritura.

CÓDIGO:

```
PUBLIC void rw_scattered(dev, bufq, bufqsize, rw_flag)
dev_t dev;
struct buf **bufq;
int bufqsize;
int rw_flag;
{
       register struct buf *bp;
       int gap;
       register int i;
       register struct iorequest_s *iop;
       static struct iorequest_s iovec[NR_BUFS];
       int j;
/* ordena los buffers según b_blocknr */
  gap = 1;
  do
       gap = 3 * gap + 1;
  while (gap <= bufqsize);
  while (gap != 1) {
       gap /= 3;
       for (j = gap; j < bufqsize; j++) {
              for (i = j - gap;
                   i >= 0 && bufq[i]->b_blocknr > bufq[i + gap]->b_blocknr;
                   i -= gap) {
                     bp = bufq[i];
                     bufq[i] = bufq[i + gap];
                     bufq[i + gap] = bp;
              }
       }
  }
/* Configuramos vector de E/S y hacemos E/S. El resultado de 'dev_io' se
descarta porque todos los resultados se retornan en el vector. Si dev_io falla
por completo, el vector no se modifica y todos sus resultados se toman como
erróneos */
  while (bufqsize > 0) {
       for (j = 0, iop = iovec; j < NR_IOREQS && j < bufqsize; j++, iop++) {
              bp = bufq[j];
              iop->io_position = (off_t) bp->b_blocknr * BLOCK_SIZE;
              iop->io_buf = bp->b_data;
              iop->io_nbytes = BLOCK_SIZE;
              iop->io_request = rw_flag == WRITING ?
                            DEV_WRITE : DEV_READ | OPTIONAL_IO;
       }
       (void) dev_io(SCATTERED_IO, 0, dev, (off_t) 0, j, FS_PROC_NR,
                                              (char *) iovec);
```

```
/* Recoge los resultados. Deja los errores de lectura para 'rw_block()' */
       for (i = 0, iop = iovec; i < j; i++, iop++) {
              bp = bufq[i];
              if (rw_flag == READING) {
                  if (iop->io_nbytes == 0)
                     bp->b_dev = dev;    /* Bloque validado */
                  put_block(bp, PARTIAL_DATA_BLOCK);
              } else {
                  if (iop->io_nbytes != 0) {
                    printf("Unrecoverable write error on device %d/%d, block
              %ld\n",(dev>>MAJOR)&BYTE, (dev>>MINOR)&BYTE, bp->b_blocknr);
                     bp->b_dev = NO_DEV; /* Bloque invalidado */
                  }
                  bp->b_dirt = CLEAN;
              }
       }
       bufq += j;
       bufqsize -= j;
  }
}
```

1.11. RM_RLU()

Este procedimiento elimina un bloque de su cadena LRU.

Parámetros de entrada

bp : Puntero al bloque a borrar.

CÓDIGO:

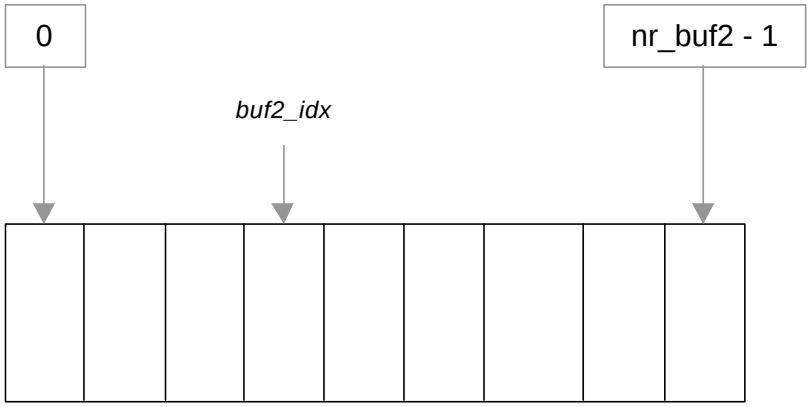
```
PRIVATE void rm_lru(bp)
struct buf *bp;
{
  struct buf *next_ptr, *prev_ptr;

  bufs_in_use++;
  next_ptr = bp->b_next;  /* Sucesor en la cadena */
  prev_ptr = bp->b_prev;  /* Predecesor en la cadena */
  if (prev_ptr != NIL_BUF)
       prev_ptr->b_next = next_ptr;
  else
       front = next_ptr;     /* bloque al principio de la cadena (frente) */

  if (next_ptr != NIL_BUF)
       next_ptr->b_prev = prev_ptr;
  else
       rear = prev_ptr;      /* bloque al final de la cadena (fondo) */
}
```

2. Cache2.c

2.1. Introducción.

La caché de bloques de un sistema Minix de 16 bits es demasiado pequeña como para evitar el "trashing", lectura y escritura continua de los mismos bloques porque no hay espacio suficiente para mantenerlos en memoria. Un sistema genérico de 32 bits tampoco tiene una caché muy grande como para permitirle ejecutarse en sistemas con poca memoria. En un sistema con grandes cantidades de memoria se puede usar la RAM de disco como una cache secundaria de solo lectura, en la que se guarden los bloques expulsados de la caché de primer nivel.

En el módulo *cache2.c* se indican los procedimientos que se encargan de manejar esta caché de segundo nivel, implementada como una simple cola FIFO en la que se introducen los bloques antiguos y se eliminan por el otro extremo. Esta cola deberá ser recorrida desde el final hasta el principio para localizar un bloque.



Los siguientes procedimientos se compilarán si está declarada la macro ENABLE_CACHE2.

2.2. Estructuras y tipos de datos.

```
#define MAX_BUF2    (256 * sizeof(char *))

PRIVATE struct buf2 {
  block_t b2_blocknr;           /* número de bloque */
  dev_t b2_dev;                 /* número de dispositivo */
  u16_t b2_count;               /* contador de los grupos de bloques en caché
*/
} buf2[MAX_BUF2];

PRIVATE unsigned nr_buf2;       /* tamaño real de la caché */
PRIVATE unsigned buf2_idx;      /* índice para gestionar la cola */

#define hash2(block)      ((unsigned) ((block) & (MAX_BUF2 - 1)))
```

MAX_BUF2: Número máximo de bloques en la caché secundaria.

BUF2[MAX_BUF2] : Donde se implementará la cola FIFO.
Cada buffer consta de :
b2_blocknr : Número de bloque.
b2_dev : Número de dispositivo.
b2_count : Contador del número de grupos de bloques en caché.

nr_buf2 : Tamaño actual de la caché.
buf2_idx : Índice para gestionar BUF2.

2.3. INIT_CACHE2()

Inicializa la caché de disco de segundo nivel a 'size' bloques (o a MAX_BUF2 bloques si 'size' es demasiado grande).

Parámetros de entrada

size : Tamaño al que se desea inicializar la caché secundaria.

CÓDIGO:

```
PUBLIC void init_cache2(size)
unsigned long size;
{
  nr_buf2 = size > MAX_BUF2 ? MAX_BUF2 : (unsigned) size;
}
```

2.4. GET_BLOCK2()

Busca un bloque en la caché de segundo nivel. La búsqueda se hace hacia atrás. Retorna verdadero si se encontró el bloque.

Parámetros de entrada

bp : Buffer a buscar en la caché.
only_search : Si es NO_READ, no hace nada. En otro caso actúa normal.

CÓDIGO:

```
PUBLIC int get_block2(bp, only_search)
struct buf *bp;
int only_search;
{
  unsigned b;
  struct buf2 *bp2;

  /* Si el bloque deseado está en la RAM de disco, no tenemos nada que hacer*/
  if (bp->b_dev == DEV_RAM) nr_buf2 = 0;
```

```
 /* Se pregunta:
       - si la caché está habilitada
       - si estamos en modo NO_READ
       - si hay dos bloques con la misma clave hash
     Si la respuesta es afirmativa se retorna falso.  */

if (nr_buf2 == 0 || only_search == NO_READ
                   || buf2[hash2(bp->b_blocknr)].b2_count == 0) return(0);

  /* Se hace una búsqueda hacia atrás para comprobar si hay versiones
anteriores */
  b = buf2_idx;
  for (;;) {
       if (b == 0) b = nr_buf2;
       bp2 = &buf2[--b];
       if (bp2->b2_blocknr == bp->b_blocknr && bp2->b2_dev == bp->b_dev)
            break;
       /* Se ha realizado una vuelta completa y no estaba */
       if (b == buf2_idx) return(0);
  }

  /* El bloque está en la caché. Se hace la lectura y se retorna verdadero si
todo ha salido bien. */
  if (dev_io(DEV_READ, 0, DEV_RAM, (off_t) b * BLOCK_SIZE, BLOCK_SIZE,
                                  FS_PROC_NR, bp->b_data) == BLOCK_SIZE) {
       return(1);
  }
  return(0);
}
```

2.5. PUT_BLOCK2()

Coloca un bloque en la caché de segundo nivel.

Parámetros de entrada

bp : Buffer a colocar en la caché de segundo nivel.

CÓDIGO:

```
PUBLIC void put_block2(bp)
struct buf *bp;
{
  unsigned b;
  struct buf2 *bp2;

  if (nr_buf2 == 0) return;      /* No hay caché de segundo nivel */

  /* Se actualiza el índice 'buf2_idx' */
  b = buf2_idx++;
  if (buf2_idx == nr_buf2) buf2_idx = 0;

  bp2 = &buf2[b];

  /* Se rellenan los campos correspondientes y se escribe el bloque */
  if (dev_io(DEV_WRITE, 0, DEV_RAM, (off_t) b * BLOCK_SIZE, BLOCK_SIZE,
                                  FS_PROC_NR, bp->b_data) == BLOCK_SIZE) {
       if (bp2->b2_dev != NO_DEV) buf2[hash2(bp2->b2_blocknr)].b2_count--;
       bp2->b2_dev = bp->b_dev;
       bp2->b2_blocknr = bp->b_blocknr;
       buf2[hash2(bp2->b2_blocknr)].b2_count++;
  }
```

```
}
```

2.6. INVALIDATE2()

Invalida todos los bloques de un dispositivo determinado que estén en la caché.

Parámetros de entrada

device : Dispositivo al que pertenecen los bloques a eliminar.

CÓDIGO:

```
PUBLIC void invalidate2(device)
dev_t device;
{
  unsigned b;
  struct buf2 *bp2;

  /* Para invalidarlos se coloca el tipo de dispositivo a NO_DEV */
  for (b = 0; b < nr_buf2; b++) {
       bp2 = &buf2[b];
       if (bp2->b2_dev == device) {
              bp2->b2_dev = NO_DEV;
              buf2[hash2(bp2->b2_blocknr)].b2_count--;
       }
  }
}
```