# Write.c

Autores:

Miguel Ángel Umpiérrez Artiles
Enrique Pérez Díaz.

© Universidad de Las Palmas de Gran Canaria

# Índice

1.- Introducción: .......... 1
1.1.- Contiene los procedimientos: .......... 1
1.2.- Variables externas: .......... 1
1.3.- Constantes utilizadas. .......... 1
1.4.- Procedimientos que se utilizan. .......... 1
2.- Código Fuente. .......... 2
2.1.- Do_write: .......... 2
2.1.1.- Procedimiento llamado: .......... 2
2.1.2.- Código do_write: .......... 2
2.2.- Write_map: .......... 3
2.2.1.- Procedimientos llamados: .......... 3
2.2.2.- Parámetros de entrada: .......... 3
2.2.3.- Parámetros de salida: .......... 3
2.2.4.- Pseudo algoritmo: .......... 3
2.2.5.- Código: .......... 4
2.3.- Wr_indir: .......... 6
2.3.1.- Código: .......... 6
2.4.- New_block: .......... 6
2.4.1.- Procedimientos llamados: .......... 7
2.4.2.- Pseudo algoritmo. .......... 7
2.4.3.- Código: .......... 7
2.5.- Clear_Zone: .......... 8
2.5.1.- Procedimientos llamados: .......... 8
2.5.2.- Pseudo algoritmo. .......... 8
2.5.3.- Código: .......... 9
2.6.- Zero_block: .......... 10
2.6.1.- Procedimientos llamados: .......... 10
2.6.2.- Código: .......... 10

# Write.c

# 1.- Introducción:

Este fichero constituye un complemento al **READ.C** que incluye funciones específicas para realizar la escritura en ficheros.

La escritura de un archivo es similar a su lectura; Similares en que tanto la llamada al sistema ***do_read***, como ***do_write***, llaman a un procedimiento común ***read_write*** que realiza la mayor parte del trabajo, pero teniendo en cuenta que la escritura requiere la asignación de nuevos bloques de disco. Una diferencia es ***write_map*** que es análogo a ***read_map***, sólo que en lugar de buscar números de bloque físicos en el nodo i y en sus bloques indirectos, mete otros nuevos (en realidad mete números de zona, no números de bloque).

## 1.1.- Contiene los procedimientos:

- **do_write**: Ejecuta la llamada al sistema **write(fd, buffer, nbytes)**
- **write_map**: Escribe una nueva zona en un inode.
- **clear_zone**: Pone a cero una zona, que posiblemente comienza en el medio.
- **new_block**: Adquiere un nuevo bloque.
- **zero_block**: Pone a cero un bloque.
- **wr_indir:** Escribe una entrada en un bloque indirecto, la cual apunta a una zona. La entrada a usar se indica mediante un índice.

## 1.2.- Variables externas:

- **fd:** Descriptor de fichero.
- **nbytes:** Número de bytes a transferir.
- **buffer:** Dirección del buffer.

## 1.3.- Constantes utilizadas.

- **NR_DZONE_NUM**(NR_ZONE_NUMS-2): N° de zonas en el nodo i.
- **NIL_BUF** (struct buf *): Ausencia de un buffer.
- **NR_INDIRECTS** (BLOCK_SIZE/ZONE_NUM_SIZE): N° de zonas del bloque indirecto.
- **INDIRECT_BLOCK** (2+MAYBE_WRITE_INMED): Puntero a bloque.
- **FULL_DATA_BLOCK** : Datos completamente usados.
- **NO_READ**: Evita que get_block() haga la lectura del disco.
- **INTS_PER_BLOCK**(BLOCK_SIZE/sizeof(int)): N° de enteros por bloque.
- **NORMAL:** Fuerza a get_block() a leer el disco.

## 1.4.- Procedimientos que se utilizan.

- **get_super():** Búsqueda de un dispositivo en la tabla de superbloque (super.c).
- **get_block()**: Captura de un bloque para lectura o escritura (caché.c).
- **put_block()**: Retorno de un bloque antes solicitado con get_block() (caché.c).
- **free_zone()**: Liberación de una zona (cuando se elimina un archivo) (caché.c).
- **alloc_zone()**: Asignación de una nueva zona (para alargar un archivo) (caché.c).

- **memset():** Utilizada para vaciar el contenido de un bloque poniendo todo a 0.

Del fichero anterior **READ.C** utiliza las siguientes rutinas:

- **read_map()**: Dado un nodo i y la posición del fichero, determina su nº de zona.
- **rd_indir() :** Determina si existe una cierta entrada (mediante un índice) en un bloque indirecto.
- **read_write():** Se encarga de realizar la operación de escritura en sí.

# 2.- Explicación del Código Fuente.

*/* Este fichero es la contrapartida del "read.c". Contiene el código para escribir en cuanto que éste no está contenido en read_write().*

*Los puntos de entrada a este fichero son:*
*do_write: llama a read_write para llevar a cabo la llamada al sistema WRITE.*
*write_map: añade una nueva zona a un nodo i*
*clear_zone: borra una zona en medio de un fichero*
*new_block: obtiene un bloque nuevo*
**/*

```
#include "fs.h"
#include <string.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "super.h"

FORWARD _PROTOTYPE( int write_map, (struct inode *rip, off_t position,zone_t new_zone));

FORWARD _PROTOTYPE( void wr_indir, (struct buf *bp, int index, zone_t zone) );
```

## 2.1.- Do_write:

Este procedimiento ejecuta la llamada al sistema **write(fd, buffer, nbytes)** para escribir en un archivo, llamando al procedimiento **read_write**.

### 2.1.1.- Procedimiento llamado:

Los procedimiento llamados en esta función son:

- Read_write(): está en read.c y se encarga de la escritura y lectura.

### 2.1.2.- Código do_write:

```
/*===========================================================*
 *                        do_write                           *
 *===========================================================*/

PUBLIC int do_write()
{
/* Lleva a cabo la llamada al sistema write(fd, buffer, nbytes)*/
  return (read_write(WRITING));
}
```

## 2.2.- Write_map:

Esta rutina actualiza el nodo i del fichero en el que se escribe, asignando la correcta zona dentro del mismo y especificando la dirección de la misma bien directamente o con indirección simple o doble, teniendo en cuenta la posible creación de los bloques indirectos.

Es llamada por el procedimiento **new_block()** que debe recurrir al write_map() para incluir la dirección de la nueva zona en el nodo i.

Se tratan varios casos:

- **Zona directa**. Si la zona a insertar está próxima al inicio del archivo, simplemente se inserta el nodo i.
- **Zona indirecta sencilla**.
- **Zona indirecta doble.** El peor de los casos se presenta cuando el archivo está en el tamaño máximo que puede manejar un solo bloque indirecto, de modo que se debe asignar un bloque indirecto doble. Después, se debe asignar un bloque indirecto individual y su dirección colocarse en el bloque indirecto doble. Si el bloque indirecto doble se asigna con buenos resultados, pero no se puede asignar el bloque indirecto individual(es decir, el disco repleto), entonces el doble se debe liberar con cuidado de manera que no se dañe el mapa de bits.

### 2.2.1.- Procedimientos llamados:

- put_block()
- alloc_zone()
- get_block()
- rd_indir()
- wr_indir()
- zero_block()

### 2.2.2.- Parámetros de entrada:

- **rip:** puntero al nodo i que se va a actualizar.
- **position:** dirección del fichero a mapear.
- **new_zone:** número de la zona que se insertará.

### 2.2.3.- Parámetros de salida:

- **ok:** no error.
- **err_code:** error en la asignación de una zona del dispositivo.
- **efbig:** fichero demasiado grande.

### 2.2.4.- Pseudo algoritmo:

1. Marcar nodo i como DIRTY.
2. Hallar el n° de la zona y calcular n° de zona relativa a insertar.
3. Si zona directa:
    - Colocar en el nodo i, puntero a la zona.
    - Retornar ok.

   finsi
4. Hallar n° de la zona en exceso.
5. Si zona indirecta sencilla: (n°zona < n° de indirectos)
    - Obtener el bloque indirecto

```
sino (Zona indirecta doble)
    ❑ Obtener la zona para el bloque indirecto doble si
      éste no existe, llamada a alloc_zone().
    ❑ Colocar puntero a esa zona en el i_node.
    ❑ Comprueba que el fichero no es excesivamente
      grande.
    ❑ Crear el bloque, llamada a get_block() y ponerlo a
      cero si se trata de un bloque nuevo.
    ❑ Obtener entrada en el bloque indirecto doble
      puntero al bloque indirecto simple
finsi
```

6. Crear el bloque indirecto simple si no existía (Este proceso es el mismo para los bloques indirectos señalados desde el i_node que para los bloques directos del direccionamiento doble) mediante la llamada a alloc_zone.
7. Actualiza puntero a la zona (Escribe la nueva zona, directamente si es direccionamiento simple o mediante wr_indir si es direccionamiento doble ).
8. Si se creó el bloque indirecto doble
    - ❑ marcar dicho bloque como DIRTY.

   finsi
9. Si no se pudo asignar zona al bloque indirecto simple se devuelve el bloque con put_block() retorna error. finsi
10. Liberar el bloque indirecto doble.
11. Crear bloque indirecto (get_block()) y si es nuevo se pone a cero(zero_block()).
12. Actualiza puntero a la zona (Escribe la nueva zona con wr_indir()).
13. Marca el bloque como DIRTY.
14. Libera el bloque (put_block()).
15. Retorna OK.

## 2.2.5.- Código:

```
/*=========================================================== *
 *                          write_map                          *
 *=========================================================== */
PRIVATE int write_map(rip, position, new_zone)
register struct inode *rip;  /* puntero al nodo i que va  a cambiarse */
off_t position;              /* Dirección del fichero a mapear*/
zone_t new_zone;             /* Nº de zona que se insertará*/
{
/* Escribir una nueva zona en un nodo*/
  int scale, ind_ex, new_ind, new_dbl, zones, nr_indirects, single, zindex, ex;
  zone_t z, z1;
  register block_t b;
  long excess, zone;
  struct buf *bp;

  rip->i_dirt = DIRTY;                  /* El nodo i cambiará */
  bp = NIL_BUF;
  scale = rip->i_sp->s_log_zone_size;        /* para conversión zona bloque*/
  zone = (position/BLOCK_SIZE) >> scale; /* Nº de zona  relativa de la posición, a
                                            insertar en el nodo i*/
  zones = rip->i_ndzones;                    /* # zonas directas en el inode */
  nr_indirects = rip->i_nindirs ;            /* # zonas  indirectas  por bloque indirecto */

  /* Se encuentra 'position' en ese  nodo i? */
  if (zone < zones) {
        zindex = (int) zone;                  /* se  requiere un entero*/
        rip->i_zone[zindex] = new_zone;       /*Coloca el puntero a  la zona*/
```

```
        return(OK);
  }

 /* Si no está  en ese  nodo, tiene que ser indirecto sencillo o  indirecto doble */
  excess = zone - zones;               /* los primerosVx NR_DZONES no cuentan */
  new_ind = FALSE;
  new_dbl = FALSE;

  if (excess < nr_indirects) {
/* 'position' puede localizarse  via bloque indirecto  sencillo y obtiene su puntero de zona. */
        z1 = rip->i_zone[zones];
        single = TRUE;
  } else {
        /* 'position' puede localizarse  vía el bloque indirecto  doble */

        if ( (z = rip->i_zone[zones+1]) == NO_ZONE) {
              /* Si no existía crear al bloque indirecto doble */
              /*Alloc_zone() asigna una nueva zona en el dispositivo  indicado y devuelve su
              puntero*/
              if ( (z = alloc_zone(rip->i_dev, rip->i_zone[0])) == NO_ZONE)
                    return(err_code);
              rip->i_zone[zones+1] = z;/*Coloca  el puntero  de bloque indirecto doble*/
              new_dbl = TRUE;          /* activa  flag  para su posterior uso*/
        }
        /* En cualquier caso, 'z' es  el  número de zona para el  bloque indirecto doble */
        excess -= nr_indirects;      /* los bloques simplese  indirectos no cuentan */
        ind_ex = (int) (excess / nr_indirects);
        excess = excess % nr_indirects;  /*índice en el bloque indirecto simple */
        if (ind_ex >= nr_indirects) return(EFBIG); /*el fichero es muy grande? */
        b = (block_t) z << scale;
        /*Obtener un bloque indirecto doble*/
        bp = get_block(rip->i_dev, b, (new_dbl ? NO_READ : NORMAL));
        /*ponerlo a cero si trata de un bloque nuevo*/
        if (new_dbl) zero_block(bp);
        z1 = rd_indir(bp, ind_ex); /*Obtener una entrada del bloque indirecto simple*/
        single = FALSE;
  }

  /* z1 es ahora zona indirecta simple; 'excess' es  el  índice. */
  if (z1 == NO_ZONE) {

         /* Crea el bloque indirecto y almacena  la zona  # en el inode o en el bloque indirecto doble */
        z1 = alloc_zone(rip->i_dev, rip->i_zone[0]);
        if (single)
              rip->i_zone[zones] = z1;   /* update inode */
        else
              wr_indir(bp, ind_ex, z1);  /* update dbl indir */

        new_ind = TRUE;
        if (bp != NIL_BUF) bp->b_dirt = DIRTY;   /* si doble indirecto, poner a sucio*/
                          /* Si no pudo asignar una zona  al bloque indirecto simple */
        if (z1 == NO_ZONE) {
              put_block(bp, INDIRECT_BLOCK);
        /* Devolver el bloque indirecto doble a la cola de buffers, que previamente habíamos obtenido
        con get_block() */
              return(err_code); /* no se pudo crear  el indirecto simple. Error */
        }
  }
  put_block(bp, INDIRECT_BLOCK);        /* Liberar el bloque indirecto  doble */
```

```
  /* z1 apunta a un nº de zona de bloque indirecto simple  */
  b = (block_t) z1 << scale;
  bp = get_block(rip->i_dev, b, (new_ind ? NO_READ : NORMAL) );/* Obtiene bloque */

  if (new_ind) zero_block(bp);                /* Lo pone a cero si lo ha creado nuevo */
  ex = (int) excess;                          / * se  requiere un nº entero */
  wr_indir(bp, ex, new_zone);

  bp->b_dirt = DIRTY;      /*Se libera y escribe el bloque tras marcarlo como sucio */
  put_block(bp, INDIRECT_BLOCK);

  return(OK);
}
```

## 2.3.- Wr_indir:

Se utiliza para escribir un bloque de indirección. La necesidad de este procedimiento, es por que los datos pueden tener diferentes formatos en el disco, dependiendo de la versión del sistema de archivos y del hardware en el que se haya escrito el sistema de archivos. Las conversiones, si son necesarias, se realizan aquí para que el resto del sistema de archivos vea los datos en una sola forma.Para ello invoca a las rutinas de conversión conv2 o conv4, también coloca un nuevo número de zona en un bloque de indirección.

Debido a su sencillez, mostraremos solamente el código:

### 2.3.1.- Código:

```
/*===========================================================*
 *                        wr_indir                          *
 *==========================================================*/
PRIVATE void wr_indir(bp, index, zone)
struct buf *bp;                 /* puntero a bloque indirecto */
int index;                      /* índice dentro de *bp */
zone_t zone;                    /* zona a escribir */
{
/*Dado un puntero a un bloque indirecto, escribe una entrada*/

  struct super_block *sp;
/* requiere el superbloque para encontrar el tipo de sistema de fichero */
  sp = get_super(bp->b_dev);

  /* escribe una zona dentro de un bloque indirecto */
  if (sp->s_version == V1)
        bp->b_v1_ind[index] = (zone1_t) conv2(sp->s_native, (int)  zone);
  else
        bp->b_v2_ind[index] = (zone_t)  conv4(sp->s_native, (long) zone);
}
```

## 2.4.- New_block:

Obtiene un bloque nuevo devolviendo un puntero al mismo. Es invocada por rw_chunck cada vez que se necesita un bloque.
Se distinguen casos:

1. La zona actual puede tener todavía algunos bloques disponible.
2. Puede requerir asignar una nueva zona completa y entonces devolver el bloque inicial de la nueva zona.

### 2.4.1.- Procedimientos llamados:

- **alloc_zone(rip->i_dev,z):** Asigna una zona libre al dispositivo indicado.
- **read_map(rip,position):** Determina la zona correspondiente al nodo i y a la posición del fichero.
- **get_block(rip->i_dev,b,NO_READ):** Verifica si el bloque solicitado esta en la cache. Si no, expulsa algún otro bloque y captura el solicitado.
- **write_map(rip,position,z):** Incluir la dirección de la nueva zona en el nodo i.
- **free_zone(rip->i_dev,z):** Libera la zona indicada.
- **clear_zone(rip,position,1)**
- **zero_block(bp)**

### 2.4.2.- Pseudo algoritmo.

```
1. Si no está disponible otro bloque en la zona asignada (read_map)
   entonces
   1.1.  si no tiene nada escrito entonces
            ❑ Obtiene identificador del dispositivo.
            ❑ Halla la dirección de la primera zona del dispositivo
         sino
            ❑ Se coloca en la primera zona de datos del fichero en
              cuestión.
         fin si
   1.2.  Asigna zona nueva del fichero (alloc_zone).
   1.3.  Escribe la nueva zona en el nodo i (write_map)
   1.4.  si  hay fallo  entonces
            ❑ Libera zona (free_zone).
            ❑ Retorna nulo.
         sino
            ❑ si  se sale del EOF   entonces
                   vaciar la zona (clear_zone)
              fin si
         fin si
  fin si

2. Cálculo del bloque (get_block).
3. Lo pone a cero (zero_block).
4. Retorna el bloque.
```

### 2.4.3.- Código:

```
/*===========================================================*
 *                        new_block                          *
 *===========================================================*/
PUBLIC struct buf *new_block(rip, position)
register struct inode *rip;     /* puntero al nodo i */
off_t position;                 /* puntero al fichero */
{
/* Obtener un bloque nuevo y devolver un puntero a él. Hacerlo así puede requerir asignar una zona completa y entonces devlver el bloque inicial de la zona en cuestión. Por otra parte la zona actual puede tener todavía algunos bloques sin usar  */

  register struct buf *bp;
  block_t b, base_block;
  zone_t z;
  zone_t zone_size;
  int scale, r;
```

```
  struct super_block *sp;


/* Si no hay zona asignada para la posición especificada */
  if ( (b = read_map(rip, position)) == NO_BLOCK) {

        /* Elegir la primera zona si no hay nada escrito aun en el fichero */
        if (rip->i_zone[0] == NO_ZONE) {
                sp = rip->i_sp;
                z = sp->s_firstdatazone;  /* Halla la dirección de la 1ª zona de datos del dispositivo */
        } else {
                z = rip->i_zone[0];        /*se coloca en la primera zona de datos del fichero en cuestión */
        }
        /* Intenta asignar una nueva zona. Si no encuentra ninguna libre, retorna un nulo */
        if ( (z = alloc_zone(rip->i_dev, z)) == NO_ZONE) return(NIL_BUF);

        /* Si al escribir la nueva zona en el  nodo i hay error, libera la zona y retorna nulo */
        if ( (r = write_map(rip, position, z)) != OK) {
                free_zone(rip->i_dev, z);
                err_code = r;
                return(NIL_BUF);
        }

        /* Si no se está escribiendo en el  final del fichero (EOF),  limpiar la zona, por razones de seguridad */
        if ( position != rip->i_size) clear_zone(rip, position, 1);
        scale = rip->i_sp->s_log_zone_size;
        base_block = (block_t) z << scale;
        zone_size = (zone_t) BLOCK_SIZE << scale;
        b = base_block + (block_t)((position % zone_size)/BLOCK_SIZE);
  }
  /* Obtiene el bloque, el cual pone a cero */
  bp = get_block(rip->i_dev, b, NO_READ);
  zero_block(bp);
  return(bp);
}
```

## 2.5.- Clear_Zone:

Coloca a cero los bloques de la zona no utilizados. Es decir se ocupa de borrar bloques que repentinamente están en medio de un archivo. Esto sucede cuando se efectúa una búsqueda más allá del final del final del archivo, lo cual no suele ser muy frecuente.

Es llamada por read_write() y por new_block().

### 2.5.1.- Procedimientos llamados:

- **read_map(rip,next):** Determina la zona correspondiente al nodo i y a la posición del fichero.
- **zero_block(bp):** pone un bloque a cero.
- **get_block(rip->i_dev,b,NO_READ):** Verifica si el bloque solicitado esta en la cache. Si no, expulsa algún otro bloque y captura el solicitado.
- **put_block(bp,FULL_DATA_BLOCK) :** Devuelve un bloque a la lista de bloques disponibles.

### 2.5.2.- Pseudo algoritmo.

1. Calcula el tamaño de la zona.
2. Si ha sido llamado des new_block entonces

```
      ❑ ajusta al principio de la zona
   fin si
3. Calcula el principio del siguiente bloque de la zona
4. si esta en el último bloque de una zona entonces
      ❑ no vacía la zona y retorna
   fin si
5. si los siguiente bloques están disponibles read_map (rip,next)
   entonces
      ❑ obtiene el último bloque de la zona
      ❑ vacía los bloques desde el primero hasta el último
      ❑ calculados (blo,bhi). get_block y zero_block
      ❑ devuelve los buffers a la cola  put_block
   fin si
6. retornar.
```

## 2.5.3.- Código:

```
/*===================================================================== *
 *                                   clear_zone                                     *
 *===================================================================== */
PUBLIC void clear_zone (rip, pos, flag)
register struct inode *rip;        /* Nodo i a vaciar */
off_t pos;                         /* Apunta al bloque a vaciar */
int flag;                          /* 0 si es llamado por read_write, 1 si new_block */
{

/* Pone a cero una zona, posiblemente empezando por el medio. El parámetro 'pos' da un byte en el primer bloque que tiene que ser puesto a cero. Clear_zone se llama desde read_write() y new_block() */

  register struct buf *bp;
  register block_t b, blo, bhi;
  register off_t next;
  register int scale;
  register zone_t zone_size;

 /* Si los tamaños de bloque y zona son el mismo, clear_zone()  no se necesita,  es decir, se  utiliza zero_block()  directamente*/

  scale = rip->i_sp->s_log_zone_size;
  if (scale == 0) return;

  zone_size = (zone_t) BLOCK_SIZE << scale;

/* Si llamó new_block(), ajusta 'pos'al principio de la zona */
  if (flag == 1) pos = (pos/zone_size) * zone_size;
  next = pos + BLOCK_SIZE - 1; /* Calcula el principio del siguiente bloque de la zona */

 /* si 'pos' está en el último bloque de una zona, no vacía la  zona y retorna */
  if (next/zone_size != pos/zone_size) return;
  if ( (blo = read_map(rip, next)) == NO_BLOCK) return;
  bhi = (  ((blo>>scale)+1) << scale)   - 1;

  /*  Vaciar todos los bloques entre 'blo' y 'bhi' */
  for (b = blo; b <= bhi; b++) {
        bp = get_block(rip->i_dev, b, NO_READ);
        zero_block(bp);
        put_block(bp, FULL_DATA_BLOCK);
  }
}
```

## 2.6.- Zero_block:

Despeja un bloque, borrando su contenido anterior, es decir llena un bloque con ceros.

### 2.6.1.- Procedimientos llamados:

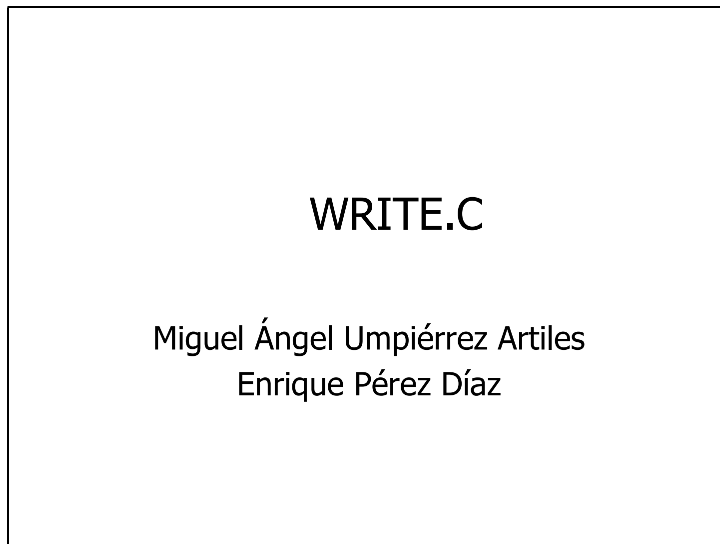
- memset(): Función que llena de un valor la zona de memoria indicada.

### 2.6.2.- Código:

```
/*===========================================================*
 *                        zero_block                         *
 *===========================================================*/
PUBLIC void zero_block(bp)
register struct buf *bp;     /* puntero al buffer a poner a cero */
{
/* Pone un bloque a cero */
  memset(bp->b_data, 0, BLOCK_SIZE);
  bp->b_dirt = DIRTY;
}
```

# 3.- Transparencias.

A continuación tenemos las transparencias que utilizamos para explicar el tema en clase, ver con detalle como llegamos hasta do_write, el resto está explicado con más detalle en el apartado 2.



# Introducción

## /src/fs/write.c

- Este fichero contiene el código de escritura que no está contenido en read_write(), función de read.c
- Contiene tres puntos de entrada:
  - **Do_write**: llama a read_write.
  - **Clear_zone**: borra una zona en medio de un archivo.
  - **New_Block**: adquiere un nuevo bloque.

## Llamada al sistema WRITE

- Cuando un programa de usuario ejecuta la instrucción

  n=write (fd,buffer,nbytes)

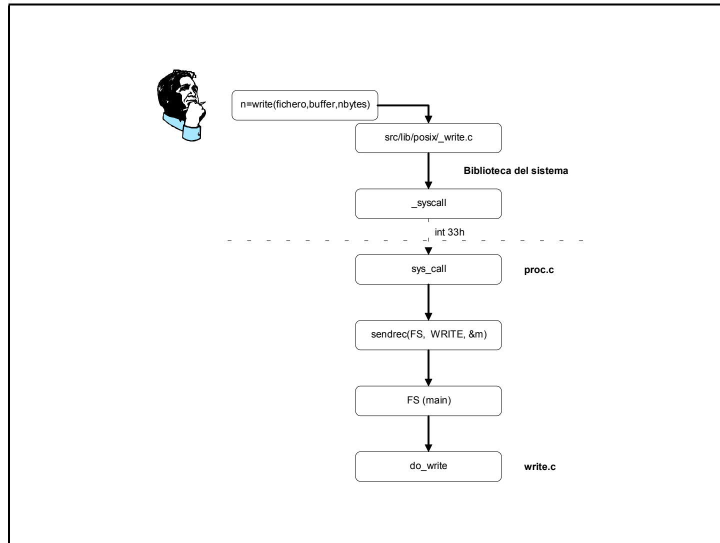
  para escribir en un archivo, se invoca al procedimiento de biblioteca write () con estos 3 parámetros.
- Ahora este procedimiento construye la estructura del mensaje con los 3 parámetros y hace la llamada _syscall con destino FS e indicando operación escritura:

## /src/lib/posix/_write.c

```
PUBLIC ssize_t write(fd, buffer, nbytes)
int fd;
_CONST void *buffer;
size_t nbytes;
{
  message m;
  /* Construimos el mensaje */
  m.m1_i1 = fd;
  m.m1_i2 = nbytes;
  m.m1_p1 = (char *) buffer;
 /* Hacemos la llamada al sistema */
 return( _syscall(FS, WRITE, &m));
}
```

## Ya estamos llegando a write.c

- ¡ojo el _syscall anterior no es este sys_call, _syscall termina con un int33h, que después de unos pasos se llega a sys_call repasar el primer parcial!
- Ahora en sys_call() que está en proc.c se construye un mensaje con destino FS (mini_rec), quedando bloqueado en espera de respuesta.
- El **main** del FS usa el tipo de mensaje como índice en el vector **call_vec** para llamar el procedimiento que se encarga de la operación de escritura: **do_write**.




# Write.c

## Contenido de Write.c

- Contiene los procedimientos:
  - **do_write**: Ejecuta la llamada al sistema write(fd, buffer, nbytes), llamando a read_write () que está en write.c
  - **new_block**: Adquiere un nuevo bloque.
  - **clear_zone**: Pone a cero una zona, que posiblemente comienza en el medio.
  - **zero_block**: Pone a cero un bloque.
  - **write_map**: Escribe una nueva zona en un inode.
  - **wr_indir:** Escribe una entrada en un bloque indirecto, la cual apunta a una zona.

## do_write

- Ejecuta la llamada al sistema write (fd, buffer, nbytes) invocando a la función read_write con el parámetro WRITING, dicha función está en read.c

```
PUBLIC int do_write()
{
  return ( read_write (WRITING) );
}
```

## new_block

- Obtiene un bloque nuevo y devuelve un puntero a él.
- Es invocada por rw_chunck cada vez que se necesita un bloque.
- Se distinguen casos:
  - La zona actual puede tener todavía algunos bloques disponible.
  - Puede requerir asignar una nueva zona completa y entonces devolver el bloque inicial de la nueva zona.
- Parámetros:
  - rip: puntero al i-node.
  - position: dirección en el fichero.

## Ejemplo práctico new_block

- Crecimiento de un archivo secuencial.
- Suponemos que tenemos un tamaño de bloque de 1K y un tamaño de zona de 2K (2 bloques por zona).
- También tenemos las siguientes zonas libres: 3,14,17,18,...
- Veamos entonces que sucede cuando se llama a new_block varias veces:

# Ejemplo práctico new_block (II)

| 1 | | 2 | | 3 | | 4 | | 5 | | 6 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| 7 | | 8 | | 9 | | 10 | | 11 | | 12 | |
| 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
| 13 | | 14 | | 15 | | 16 | | 17 | | 18 | |
| 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 |
| 19 | | 20 | | 21 | | 22 | | 23 | | 24 | |
| 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 |

# Código new_block

```
PUBLIC struct buf *new_block (rip, position)
register struct inode *rip;  /* puntero al nodo i */
off_t position;              /* dirección en el fichero */
{
  register struct buf *bp;
  block_t b, base_block;
  zone_t z;
  zone_t zone_size;
  int scale, r;
  struct super_block *sp;  /*  en caso de nueva zona es necesario leer del
                               super bloque */
```

# Código new_block (II)

```
/* Preguntamos si hay disponible un bloque en la zona actual asignadada */
 if ( ( b = read_map (rip, position) ) == NO_BLOCK) {

     /* Si no hay zona asignada, tenemos que elegir una nueva zona libre, que será
     la primera zona si es posible */
     if (rip->i_zone[0] == NO_ZONE) {
             /* No tiene zona asignada => Buscamos la primera zona de datos */
             sp = rip->i_sp;            /* obtenemos el identificador del dispositivo */
             z = sp->s_firstdatazone;  /* Halla la dirección de la 1ª zona de datos */
     } else {
             z = rip->i_zone[0];/*se coloca en la primera  zona de datos del fichero */
     }

  /*Intenta asignar una nueva zona. Si no encuentra ninguna libre, retorna un nulo*/
     if ( (z = alloc_zone(rip->i_dev, z)) == NO_ZONE) return(NIL_BUF);
```

## Código new_block (III)

```
/* Si al escribir la nueva zona en el  nodo i hay error,libera la zona y retorna nulo */
    if ( (r = write_map(rip, position, z)) != OK) {
            free_zone(rip->i_dev, z);
            err_code = r;
            return(NIL_BUF);
    }
/* Si no se está escribiendo en el  final del fichero (EOF),  limpiar la zona, por
razones   de seguridad */
    if ( position != rip->i_size) clear_zone(rip, position, 1);
    scale = rip->i_sp->s_log_zone_size;
    base_block = (block_t) z << scale;
    zone_size = (zone_t) BLOCK_SIZE << scale;
    b = base_block + (block_t)((position % zone_size)/BLOCK_SIZE);
  } // Zona activada
```

## Código new_block (IV)

```
/* Obtiene el bloque, si puede de caché y si no lo pone en caché */
 bp = get_block(rip->i_dev, b, NO_READ);

/* Pone a cero el bloque */
 zero_block(bp);

/* Devuelve el bloque nuevo que solicitamos */
 return(bp);
}
```

## clear_zone

- Se ocupa de borrar bloques que repentinamente están en medio de un archivo. Libera los bloques de una zona.
- Esto sucede cuando se efectúa una búsqueda más allá del final del final del archivo.
- Esta situación no es muy frecuente.
- Es llamada por read_write() y por new_block().

# Código clear_zone

```
PUBLIC void clear_zone (rip, pos, flag)
register struct inode *rip;     /* Nodo i a vaciar */
off_t pos;                      /* Apunta al bloque a vaciar */
int flag;                       /* 0  si es llamado por read_write, 1 si new_block */
{

  register struct buf *bp;
  register block_t b, blo, bhi;
  register off_t next;
  register int scale;
  register zone_t zone_size;
```

# Código clear_zone (II)

```
/* Si los tamaños de bloque y zona son el mismo, clear_zone()  no se necesita,  es
    decir, se  utiliza zero_block()  directamente*/
  scale = rip->i_sp->s_log_zone_size;
  if (scale == 0) return;

/* Si no tienen el mismo tamaño => Calculamos tamaño zona */
  zone_size = (zone_t) BLOCK_SIZE << scale;

/* Si llamó new_block(), ajusta 'pos' al principio de la zona ???*/
  if (flag == 1) pos = (pos/zone_size) * zone_size;
/* Calcula el principio del siguiente bloque de la zona */
  next = pos + BLOCK_SIZE - 1;
```

# Código clear_zone (III)

```
/* si 'pos' está en el último bloque de una zona, no vacía la  zona y retorna */
  if (next/zone_size != pos/zone_size) return;

/* calcula bloque inicial y final a liberar*/
  if ( (blo = read_map(rip, next)) == NO_BLOCK) return;
  bhi = (  ((blo>>scale)+1) << scale)   - 1;

 /*  Vaciar todos los bloques entre 'blo' y 'bhi' */
  for (b = blo; b <= bhi; b++) {
    bp = get_block(rip->i_dev, b, NO_READ);
    zero_block(bp);
    put_block(bp, FULL_DATA_BLOCK); // Pone el bloque en la lista de disponibles
  }
}
```

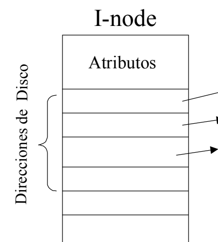
# zero_block

- Despeja un bloque, borrando su contenido anterior.

```
PUBLIC void zero_block (bp)
register struct buf *bp;          /* puntero al buffer a poner a cero */
{
 /* Pone un bloque a cero */
 memset(bp->b_data, 0, BLOCK_SIZE);
 bp->b_dirt = DIRTY;
}
```

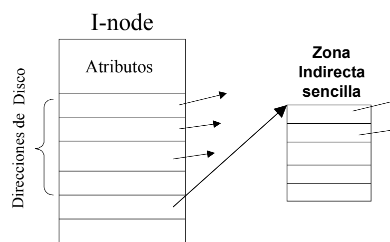
# write_map

- Es llamada por el procedimiento **new_block()** que debe recurrir a **write_map()** para incluir la dirección de la nueva zona en el nodo i.
- Se tratan varios casos:
  - **Zona directa.** Si la zona a insertar está próxima al inicio del archivo, simplemente se inserta el i-node.



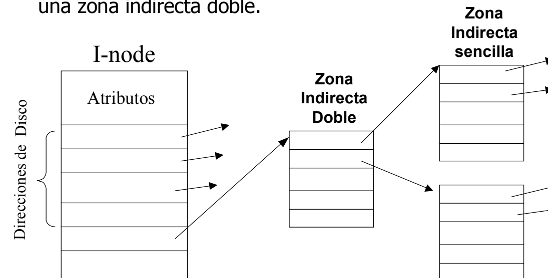

# write_map (II)

- **Zona indirecta sencilla: No hay problema.**



# write_map (III)

- **Zona indirecta doble.** El peor de los casos se presenta cuando el archivo está en el tamaño máximo que puede manejar una sola zona indirecta, de modo que se debe asignar una zona indirecta doble.




# Código write_map

- **Parámetros de entrada:**

  *int write_map(rip, position, new_zone)*

  - **rip:** puntero al nodo i que se va a actualizar.
  - **position:** dirección del fichero donde se va a escribir.
  - **new_zone:** número de la zona que se insertará.

# Código write_map (II)

```
PRIVATE int write_map (rip, position, new_zone)
register struct inode *rip;  /* puntero al nodo i que va  a cambiarse */
off_t position;              /* Dirección del fichero a mapear*/
zone_t new_zone;             /* Nº de zona que se insertará*/
{
/* Escribir una nueva zona en un nodo*/
  int scale, ind_ex, new_ind, new_dbl, zones, nr_indirects, single, zindex,
    ex;
  zone_t z, z1;
  register block_t b;
  long excess, zone;
  struct buf *bp;
```

## Código write_map (III)

```
rip->i_dirt = DIRTY;                     /* El  nodo i cambiará */
bp = NIL_BUF;
scale = rip->i_sp->s_log_zone_size;      /* para conversión zona bloque*/
zone = (position/BLOCK_SIZE) >> scale;   /* Nº de zona  relativa de la posición,
                                            a  insertar en el nodo i*/
zones = rip->i_ndzones;                  /* nº zonas directas en el inode */
nr_indirects = rip->i_nindirs ;     /* nº zonas  indirectas  por bloque indirecto */

 /* Primer caso: Zona Directa  */
 /* Se encuentra 'position' en ese  nodo i? */
if (zone < zones) {
   zindex = (int) zone;                /* se  requiere un entero */
   rip->i_zone[zindex] = new_zone;     /* Coloca  el  puntero a  la nueva zona */
   return(OK);
}
```

## Código write_map (IV)

```
/* Si no está  en ese  nodo, tiene que ser indirecto sencillo o  indirecto doble */
 excess = zone - zones;        /* los primerosVx NR_DZONES no cuentan */
 new_ind = FALSE;
 new_dbl = FALSE;

 if (excess < nr_indirects) {

/* 'position' puede localizarse  via bloque indirecto  sencillo y obtiene su puntero de
    zona. */
    z1 = rip->i_zone[zones];
    single = TRUE;
 } else {
    /* 'position' puede localizarse  vía el bloque indirecto  doble */
```

## Código write_map (V)

```
/* Guardamos en z el numero de zona para el bloque  indirecto doble */
     if ( (z = rip->i_zone[zones+1]) == NO_ZONE) {

   /* Si no existía crear al bloque indirecto doble */
   /*  Alloc_zone() asigna una nueva zona en el dispositivo  indicado y devuelve su
       puntero*/
         if ( (z = alloc_zone(rip->i_dev, rip->i_zone[0])) == NO_ZONE)
                return(err_code);
         rip->i_zone[zones+1] = z;   /* Coloca  el puntero  de bloque indirecto
                                        doble */
         new_dbl = TRUE;             /* activa  flag  para su posterior uso */
   }

/* En cualquier caso, 'z' es  el  número de zona para el  bloque indirecto doble */
```

## Código write_map (VI)

```
    excess -= nr_indirects;      /* los bloques simples  indirectos no cuentan */
   ind_ex = (int) (excess / nr_indirects);  /* índice en el bloque indirecto simple */
    excess = excess % nr_indirects;
    if (ind_ex >= nr_indirects) return(EFBIG); /*el fichero es muy grande? */
    b = (block_t) z << scale;               /* obtenemos el numero de bloque */
   /* Obtener un bloque indirecto doble */
    bp = get_block(rip->i_dev, b, (new_dbl ? NO_READ : NORMAL));
    /* ponerlo a cero si trata de un bloque nuevo */
    if (new_dbl) zero_block(bp);
    z1 = rd_indir(bp, ind_ex); /*Obtener una entrada del bloque indirecto simple*/
    single = FALSE;
}
```

## Código write_map (VII)

```
/* z1 es ahora zona indirecta simple; 'excess' es  el  índice. */
if (z1 == NO_ZONE) {
   z1 = alloc_zone(rip->i_dev, rip->i_zone[0]);   /* Crea el bloque indirecto */
   if (single) rip->i_zone[zones] = z1; /* actualiza el inode, apuntando al bloq.
   else                                   indirecto */
          wr_indir(bp, ind_ex, z1); /* actualiza el bloque indirecto doble */
   new_ind = TRUE;
   if (bp != NIL_BUF) bp->b_dirt = DIRTY; /* si doble indirecto, poner a sucio*/
   if (z1 == NO_ZONE) { /* Si no pudo asignar una zona  al bloque indirecto
                              simple */
       put_block(bp, INDIRECT_BLOCK);
    /* Devolver el bloque indirecto doble a la cola de buffers, que previamente
   habíamos obtenido con get_block() */
      return(err_code);  }      /* no se pudo crear  el indirecto simple. Error */
}
```

## Código write_map (VIII)

```
  put_block(bp, INDIRECT_BLOCK);      /* Liberar el bloque indirecto  doble */
  /* z1 apunta a un nº de zona de bloque indirecto simple  */
  b = (block_t) z1 << scale;
  bp = get_block(rip->i_dev, b, (new_ind ? NO_READ : NORMAL) ); /* Obtiene
                                                                  bloque */
  if (new_ind) zero_block(bp);       /* Lo pone a cero si lo ha creado nuevo */
  ex = (int) excess;                 /* se  requiere un nº entero */
  wr_indir(bp, ex, new_zone);        /* escribe la nueva zona en el bloque indirecto*/

  bp->b_dirt = DIRTY;              /* Se libera y escribe el bloque tras marcarlo como
                                      sucio */
  put_block(bp, INDIRECT_BLOCK);
  return(OK);
}
```

## wr_indir

- Se utiliza para escribir un bloque de indirección.
- La necesidad de este procedimiento, es por que los datos pueden tener diferentes formatos en el disco, dependiendo de la versión del sistema de archivos y del hardware en el que se haya escrito el sistema de archivos.
- Las conversiones, si son necesarias, se realizan aquí para que el resto del sistema de archivos vea los datos en una sola forma.
- Para ello invoca a las rutinas de conversión conv2 o conv4, también coloca un nuevo número de zona en un bloque de indirección.

## Código wr_indir

```
PRIVATE void wr_indir (bp, index, zone)
struct buf *bp;              /* puntero a bloque indirecto */
int index;                   /* indice dentro de *bp */
zone_t zone;                 /* zona a escribir */
{
/*Dado un puntero a un bloque indirecto, escribe una entrada*/
  struct super_block *sp;
/* requiere el superbloque para encontrar el tipo de sistema de fichero */
  sp = get_super(bp->b_dev);
/* escribe una zona dentro de un bloque indirecto */
  if (sp->s_version == V1) /* Pregunta por la versión del sistema */
     bp->b_v1_ind[index] = (zone1_t) conv2(sp->s_native, (int)  zone);
  else
     bp->b_v2_ind[index] = (zone_t)  conv4(sp->s_native, (long) zone);
}
```

## Preguntas:

1. ¿ Hacer una traza de cómo se llega a do_write, desde que un usuario hace un write?
2. ¿ Explicar como da los bloques la función new_block?
3. ¿ Explicar la función write_map?.