fs/main.c

minix 2.0

Carlos J. Suarez Rodríguez
Alberto Martínez Pérez

Índice general

INTRODUCCIÓN

El archivo MAIN.C contiene el programa principal del "sistema de ficheros", el cual se inicializa una vez que lo hayan hecho el kernel y el MM (recordemos que ambos, una vez inicializados, quedan bloqueados a la espera de un mensaje).

Este fichero contiene una serie de funciones:

MAIN: Bucle principal del sistema de ficheros. No finalizará mientras el sistema operativo esté funcionando. De estructura muy similar al MAIN del memory manager. MAIN hace uso de estas funciones:

FS_INIT: Inicializa el sistema de ficheros. Hace uso de las funciones:

BUF_POOL: Inicializa la cache de bloques del file system.

LOAD_RAM: Inicializa el disco de RAM. Esto se hace así porque el sistema de ficheros root se suele instalar en este dispositivo por eficiencia.

LOAD_SUPER: Inicializa la tabla de superbloque y carga el superbloque del dispositivo raíz.

GET_BOOT_PARAMETERS: Pregunta a la tarea del sistema por los parámetros del boot.

GET_WORK: Espera por un mensaje de petición de servicio. Primero buscará si algún proceso está suspendido y lo resucitará.

REPLY: Envía un mensaje de respuesta al proceso que haya pedido el servicio.

ESTRUCTURAS DE DATOS

Las estructuras de datos mas significativas en MAIN.C (asi como en el resto del file system) podemos extraerlas directamente de parte de los ficheros cabecera incluidos en MAIN.C.

BUF.H: La estructura de datos definida aquí es **BUF** que es la caché de bloques del sistema de ficheros del MINIX.

FPROC.H: En este fichero se declara el array **FPROC** con una entrada por cada proceso. Se trata de una tabla de procesos particular del file system en los que para cada proceso se guardan entre cosas los punteros a los i-nodos para el directorio de trabajo, el descriptor de fichero, el gid, el uid, el id del proceso, etc... Entre los datos que se almacenan aquí están los parámetros de las llamadas al sistema que deben esperar por determinados recursos, como es el caso de intentar leer de un pipe vacío. Esta información está duplicada en parte en las tablas de procesos del Kernel y del Memory Manager.

DEV.H: En este fichero se define la tabla **DMAP** que luego es inicializada en el fichero TABLA.C que se muestra en la siguiente página. Esta tabla realiza el mapeado entre los numeros de dispositivo mayor y las tareas correspondientes. En TABLA.C también se inicializa el vector de punteros **CALL_VECTOR** que se comentará posteriormente.

FILE.H: Contiene la tabla intermedia **FILP** usada entre otras cosas para guardar posiciones de ficheros, punteros a i-nodos, etc...

INODE.H: Contiene los **I-NODOS** que son traidos a memoria principal cuando se abre un fichero, y que se conservan ahí hasta que el fichero es cerrado.

SUPER.H: En este fichero se realiza la declaración de la **TABLA DEL SUPERBLOQUE**. Cuando el sistema arranca el superbloque del dispositivo raiz se carga en esa tabla.

A continuación se muestra el contenido del fichero **TABLE.C**. Este fichero realiza la inicialización de dos arrays. **CALL_VECTOR** contiene el array de punteros usado en el bucle principal para ejecutar el procedimiento adecuado a un numero de llamada al sistema dado. Existe una tabla similar en el Memory Manager. Por otro lado está **DMAP**. Esta tabla tiene una fila por cada dispositivo mayor. Cuando un dispositivo es abierto, cerrado, leido o escrito, esta tabla es la que proporciona el procedimiento a llamar para llevar a cabo la operación.

```
/* This file contains the table used to map system call numbers onto the
 * routines that perform them.
 */

#define _TABLE

#include "fs.h"
#include <minix/callnr.h>
#include <minix/com.h>
#include "buf.h"
#include "dev.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "lock.h"
#include "super.h"

PUBLIC _PROTOTYPE (int (*call_vector[NCALLS]), (void) ) = {
      no_sys,           /*  0 = unused    */
      do_exit,    /*  1 = exit      */
      do_fork,    /*  2 = fork      */
      do_read,    /*  3 = read      */
      do_write,   /*  4 = write     */
      do_open,    /*  5 = open      */
      do_close,   /*  6 = close     */
      no_sys,           /*  7 = wait      */
      do_creat,   /*  8 = creat     */
      do_link,    /*  9 = link      */
      do_unlink,  /* 10 = unlink    */
      no_sys,           /* 11 = waitpid   */
      do_chdir,   /* 12 = chdir     */
      do_time,    /* 13 = time      */
      do_mknod,   /* 14 = mknod     */
      do_chmod,   /* 15 = chmod     */
      do_chown,   /* 16 = chown     */
      no_sys,           /* 17 = break     */
      do_stat,    /* 18 = stat      */
      do_lseek,   /* 19 = lseek     */
      no_sys,           /* 20 = getpid    */
      do_mount,   /* 21 = mount     */
      do_umount,  /* 22 = umount    */
      do_set,           /* 23 = setuid    */
      no_sys,           /* 24 = getuid    */
      do_stime,   /* 25 = stime     */
      no_sys,           /* 26 = ptrace    */
      no_sys,           /* 27 = alarm     */
      do_fstat,   /* 28 = fstat     */
      no_sys,           /* 29 = pause     */
      do_utime,   /* 30 = utime     */
      no_sys,           /* 31 = (stty)    */
      no_sys,           /* 32 = (gtty)    */
      do_access,  /* 33 = access    */
      no_sys,           /* 34 = (nice)    */
      no_sys,           /* 35 = (ftime)   */
```

```
      do_sync,    /* 36 = sync      */
      no_sys,           /* 37 = kill      */
      do_rename,  /* 38 = rename    */
      do_mkdir,   /* 39 = mkdir     */
      do_unlink,  /* 40 = rmdir     */
      do_dup,           /* 41 = dup */
      do_pipe,    /* 42 = pipe      */
      do_tims,    /* 43 = times     */
      no_sys,           /* 44 = (prof)    */
      no_sys,           /* 45 = unused    */
      do_set,           /* 46 = setgid    */
      no_sys,           /* 47 = getgid    */
      no_sys,           /* 48 = (signal)*/
      no_sys,           /* 49 = unused    */
      no_sys,           /* 50 = unused    */
      no_sys,           /* 51 = (acct)    */
      no_sys,           /* 52 = (phys)    */
      no_sys,           /* 53 = (lock)    */
      do_ioctl,   /* 54 = ioctl     */
      do_fcntl,   /* 55 = fcntl     */
      no_sys,           /* 56 = (mpx)     */
      no_sys,           /* 57 = unused    */
      no_sys,           /* 58 = unused    */
      do_exec,    /* 59 = execve    */
      do_umask,   /* 60 = umask     */
      do_chroot,  /* 61 = chroot    */
      do_setsid,  /* 62 = setsid    */
      no_sys,           /* 63 = getpgrp   */

      no_sys,           /* 64 = KSIG: signals originating in the kernel */
      do_unpause, /* 65 = UNPAUSE   */
      no_sys,     /* 66 = unused  */
      do_revive,  /* 67 = REVIVE    */
      no_sys,           /* 68 = TASK_REPLY      */
      no_sys,           /* 69 = unused */
      no_sys,           /* 70 = unused */
      no_sys,           /* 71 = SIGACTION */
      no_sys,           /* 72 = SIGSUSPEND */
      no_sys,           /* 73 = SIGPENDING */
      no_sys,           /* 74 = SIGPROCMASK */
      no_sys,           /* 75 = SIGRETURN */
      no_sys,           /* 76 = REBOOT */
};


/* Some devices may or may not be there in the next table. */
#define DT(enable, open, rw, close, task) \
      { (enable ? (open) : no_dev), (enable ? (rw) : no_dev), \
        (enable ? (close) : no_dev), (enable ? (task) : 0) },

/* The order of the entries here determines the mapping between major
device
 * numbers and tasks.  The first entry (major device 0) is not used.  The
 * next entry is major device 1, etc.  Character and block devices can be
 * intermixed at random.  If this ordering is changed, the devices in
 * <include/minix/boot.h> must be changed to correspond to the new values.
 * Note that the major device numbers used in /dev are NOT the same as the
 * task numbers used inside the kernel (as defined in
<include/minix/com.h>).
 * Also note that if /dev/mem is changed from 1, NULL_MAJOR must be changed
 * in <include/minix/com.h>.
 */



PUBLIC struct dmap dmap[] = {
```

```
/*  ?   Open        Read/Write   Close        Task #       Device  File
    -   ----        ----------   -----        -------      ------  ----
*/
  DT(1, no_dev,    no_dev,      no_dev,      0)            /* 0 = not used
*/
  DT(1, dev_opcl,  call_task,   dev_opcl,    MEM)          /* 1 = /dev/mem
*/
  DT(1, dev_opcl,  call_task,   dev_opcl,    FLOPPY)       /* 2 = /dev/fd0
*/
  DT(ENABLE_WINI,
        dev_opcl,  call_task,   dev_opcl,    WINCHESTER)   /* 3 = /dev/hd0
*/
  DT(1, tty_open,  call_task,   dev_opcl,    TTY)          /* 4 = /dev/tty00
*/
  DT(1, ctty_open, call_ctty,   ctty_close, TTY)          /* 5 = /dev/tty
*/
  DT(1, dev_opcl,  call_task,   dev_opcl,     PRINTER)     /* 6 = /dev/lp
 */

#if (MACHINE == IBM_PC)
  DT(ENABLE_NETWORKING,
        net_open,  call_task,   dev_opcl,    INET_PROC_NR)/* 7 = /dev/ip
*/
  DT(ENABLE_CDROM,
        dev_opcl,  call_task,   dev_opcl,    CDROM)        /* 8 = /dev/cd0
*/
  DT(0, 0,         0,           0,           0)            /* 9 = not used
*/
  DT(ENABLE_SCSI,
        dev_opcl,  call_task,   dev_opcl,    SCSI)         /*10 = /dev/sd0
*/
  DT(0, 0,         0,           0,           0)            /*11 = not used
*/
  DT(0, 0,         0,           0,           0)            /*12 = not used
*/
  DT(ENABLE_AUDIO,
        dev_opcl,  call_task,   dev_opcl,    AUDIO)        /*13 = /dev/audio
*/
  DT(ENABLE_AUDIO,
        dev_opcl,  call_task,   dev_opcl,    MIXER)        /*14 = /dev/mixer
*/
#endif /* IBM_PC */

#if (MACHINE == ATARI)
  DT(ENABLE_SCSI,
        dev_opcl,  call_task,   dev_opcl,    SCSI)         /* 7 =
/dev/hdscsi0 */
#endif
};

PUBLIC int max_major = sizeof(dmap)/sizeof(struct dmap);
```

MAIN

El **bucle principal del procedimiento** es semejante al del manejador de memoria y al de las tareas de E/S. Mediante la llamada a GET_WORK, espera a que le llegue un mensaje de solicitud de servicio. Una vez recibido, actualiza la variable global ***who***, que nos va a indicar qué proceso (dentro de la tabla de procesos) realiza la llamada al FS y ***fs_call*** que nos informa del nº de la llamada del FS a efectuar. La variable ***who*** servirá como entrada, una vez retornado al programa principal, a la tabla de procesos (fproc), donde se comprobará si es superusuario o no (super_user).

Seguidamente se colocará ***don't_reply*** a falso, lo que indica que a no ser que se especifique lo contrario se enviará respuesta al solicitante de la llamada (un caso en el que se inhibe la respuesta es en el proceso de revivir, en el que los procesos son despertados por el manejador de tareas y a éste nunca se le envía respuesta).

A continuación realiza la llamada, empleando fs_call como índice al vector de punteros de procedimientos ***call_vector*** (el cual maneja los distintos tipos de mensaje).

Las llamadas manejadas por el FS son:

exit, fork, revive, setuid, setgid.	chmod, chown, umask, acces.
mount, umount.	link, unlink, rename.
sync, dup, fcntl.	pipe, unpause.
ioctl.	read, write.
creat, mknod, open, close, lseek.	time, utime, stime, times.
chdir, chroot, stat, fstat.	

Una vez efectuada dicha llamada, si la variable ***don't_reply*** no ha sido modificada, se envía una respuesta.

Para finalizar con este procedimiento, se hará una llamada al READ_AHEAD (read.c), para permitir leer un bloque en el pool antes de que se necesite con el fin de mejorar el rendimiento. (Esta operación se efectuará después de enviar la respuesta de manera que el usuario podrá continuar la ejecución, aunque el sistema de archivos tenga que esperar un bloque de disco para realizar la lectura anticipada).

```
/* This file contains the main program of the File System.  It consists of
 * a loop that gets messages requesting work, carries out the work, and
 * sends replies.
 * The entry points into this file are
 *   main:  main program of the File System
 *   reply: send a reply to a process after the requested work is done
 */

struct super_block;           /* proto.h needs to know this */

#include "fs.h"
#include <fcntl.h>
#include <string.h>
#include <sys/ioctl.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include <minix/boot.h>
#include "buf.h"
#include "dev.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"
#include "super.h"

FORWARD _PROTOTYPE( void buf_pool, (void)                   );
FORWARD _PROTOTYPE( void fs_init, (void)                    );
FORWARD _PROTOTYPE( void get_boot_parameters, (void)        );
FORWARD _PROTOTYPE( void get_work, (void)                   );
FORWARD _PROTOTYPE( void load_ram, (void)                   );
FORWARD _PROTOTYPE( void load_super, (Dev_t super_dev)      );


/*===========================================================================*
 *                        main                                *
 *===========================================================================*/
PUBLIC void main()
{
/* This is the main program of the file system.  The main loop consists of
 * three major activities: getting new work, processing the work, and
sending
 * the reply.  This loop never terminates as long as the file system runs.
 */
  int error;

  fs_init();

  /* This is the main loop that gets work, processes it, and sends replies.
*/
  while (TRUE) {
      get_work();       /* sets who and fs_call */

      fp = &fproc[who]; /* pointer to proc table struct */
      super_user = (fp->fp_effuid == SU_UID ? TRUE : FALSE);   /* su? */
      dont_reply = FALSE;     /* in other words, do reply is default */

      /* Call the internal function that does the work. */
      if (fs_call < 0 || fs_call >= NCALLS)
            error = EBADCALL;
      else
            error = (*call_vector[fs_call])();

      /* Copy the results back to the user and send reply. */
      if (dont_reply) continue;
      reply(who, error);
```

```
      if (rdahed_inode != NIL_INODE) read_ahead(); /* do block read ahead
*/
   }
}
```

GET_WORK

Este procedimiento se encarga de:

A) Revivir procesos suspendidos (en caso de haberlos). Se verificará si algunos procedimientos que antes estaban bloqueados, en espera de respuesta por parte del sistema de archivos (que mantiene suspendidos a aquellos procesos a los que no ha podido completar alguna operación de lectura o escritura en un terminal o archivo de interconexión etc.), ahora han "vuelto a la vida", es decir, han finalizado las operaciones por las que estaban esperando. Para ello comprueba si la variable que contabiliza el nº de procesos a revivir es distinta de cero. Luego procede a la búsqueda del primer proceso pendiente de revivir, recupera los parámetros que se almacenaron (parámetros de E/L) al producirse la suspensión y retorna al bucle principal del main.

B) Recoger los mensajes que se envían para ejecutar las llamadas al sistema. En caso de que no exista algún procedimiento para revivir, esperará a recibir algún mensaje de llamada al sistema de ficheros.

En ambos casos se cargan las variables: ***who*** (quién realizó la llamada al FS y nos servirá como entrada a la tabla de proceso), y ***fs_call*** (indicará el nº de llamada al sistema solicitada). La diferencia estriba en dónde obtiene dicha información ya que en el caso de revivir un proceso (A), la toma de la tabla de procesos, y en el otro caso (B) la toma del propio mensaje.

Parámetros de entrada: ninguno .
Parámetros de salida : " " .
Procedimientos que utiliza :

- **Receive(source, &m):** (Externo), para recibir un mensaje.

```
/*===========================================================================*
 *                        get_work                                           *
 *===========================================================================*/
PRIVATE void get_work()
{
  /* Normally wait for new input.  However, if 'reviving' is
   * nonzero, a suspended process must be awakened.
   */

  register struct fproc *rp;

  if (reviving != 0) {
      /* Revive a suspended process. */
      for (rp = &fproc[0]; rp < &fproc[NR_PROCS]; rp++)
            if (rp->fp_revived == REVIVING) {
                  who = (int)(rp - fproc);
                  fs_call = rp->fp_fd & BYTE;
                  fd = (rp->fp_fd >>8) & BYTE;
                  buffer = rp->fp_buffer;
                  nbytes = rp->fp_nbytes;
                  rp->fp_suspended = NOT_SUSPENDED; /*no longer hanging*/
                  rp->fp_revived = NOT_REVIVING;
                  reviving--;
                  return;
            }
      panic("get_work couldn't revive anyone", NO_NUM);
  }

  /* Normal case.  No one to revive. */
  if (receive(ANY, &m) != OK) panic("fs receive error", NO_NUM);

  who = m.m_source;
  fs_call = m.m_type;
}
```

REPLY

Se encarga de enviar una respuesta al proceso de usuario que solicitó la llamada al FS.

Parámetros de entrada: Whom : A quién se envía la respuesta.

Result : Resultado de la llamada (OK o Código de error).

Parámetros de salida : Ninguno .

Procedimientos que utiliza:

- **Send (destino, &m):** (Externo), Para enviar un mensaje.

```
/*===========================================================================*
 *                          reply                                            *
*===========================================================================*/
PUBLIC void reply(whom, result)
int whom;               /* process to reply to */
int result;             /* result of the call (usually OK or error #) */
{
/* Send a reply to a user process. It may fail (if the process has just
 * been killed by a signal), so don't check the return code.  If the send
 * fails, just ignore it.
 */

  reply_type = result;
  send(whom, &m1);
}
```

FS_INIT

Este procedimiento lleva a cabo la inicialización del FS, realizando las siguientes tareas:

ALGORITMO

1. Forma las listas entrelazadas que empleará la reserva del bloque. | **(BUF_POOL)**
2. Realiza una copia de los parámetros_boot. | **(GET_BOOT_PARAMETERS)**
3. Inicializa el disco RAM, y lo carga si es el disco raíz. | **(LOAD_RAM)**
4. Inicializa la tabla del superbloque y carga el superbloque del dispositivo raíz. | **(LOAD_SUPER)**
5. Inicializa las entradas a la tabla de los procesos MM (proceso 0) e INIT (proceso 2).
6. Por último comprueba si se cumplen ciertas relaciones para que el sistema de archivo trabaje al máximo. (tales como que el tamaño del superbloque no supere el tamaño del bloque, que el tamaño de los inodes sea un divisor del tamaño del bloque).

Parámetros de entrada y/o salida : Ninguno.

Procedimientos que utiliza :

- **Get_inode(dev, num_inode):** (Externo:fs/inode.c) Busca en la tabla de nodos-i uno dado; si no está, lo lee.
- **Dup_inode(ip):** (Externo:fs/inode.c) Indicará que alguien más está usando una entrada en la tabla de inode.

```
/*===========================================================================*
 *                        fs_init                                            *
 *===========================================================================*/
PRIVATE void fs_init()
{
/* Initialize global variables, tables, etc. */

  register struct inode *rip;
  int i;
  message mess;

  /* The following initializations are needed to let dev_opcl succeed .*/
  fp = (struct fproc *) NULL;
  who = FS_PROC_NR;

  buf_pool();                   /* initialize buffer pool */
  get_boot_parameters();        /* get the parameters from the menu */
  load_ram();                   /* init RAM disk, load if it is root */
  load_super(ROOT_DEV);         /* load super block for root device */

  /* Initialize the 'fproc' fields for process 0 .. INIT. */
  for (i = 0; i <= LOW_USER; i+= 1) {
      if (i == FS_PROC_NR) continue;      /* do not initialize FS */
      fp = &fproc[i];
      rip = get_inode(ROOT_DEV, ROOT_INODE);
      fp->fp_rootdir = rip;
      dup_inode(rip);
      fp->fp_workdir = rip;
      fp->fp_realuid = (uid_t) SYS_UID;
      fp->fp_effuid = (uid_t) SYS_UID;
      fp->fp_realgid = (gid_t) SYS_GID;
      fp->fp_effgid = (gid_t) SYS_GID;
      fp->fp_umask = ~0;
  }

  /* Certain relations must hold for the file system to work at all. */
  if (SUPER_SIZE > BLOCK_SIZE) panic("SUPER_SIZE > BLOCK_SIZE", NO_NUM);
  if (BLOCK_SIZE % V2_INODE_SIZE != 0)    /* this checks V1_INODE_SIZE too
*/
      panic("BLOCK_SIZE % V2_INODE_SIZE != 0", NO_NUM);
  if (OPEN_MAX > 127) panic("OPEN_MAX > 127", NO_NUM);
  if (NR_BUFS < 6) panic("NR_BUFS < 6", NO_NUM);
  if (V1_INODE_SIZE != 32) panic("V1 inode size != 32", NO_NUM);
  if (V2_INODE_SIZE != 64) panic("V2 inode size != 64", NO_NUM);
  if (OPEN_MAX > 8 * sizeof(long)) panic("Too few bits in fp_cloexec",
NO_NUM);

  /* Tell the memory task where my process table is for the sake of ps(1).
*/
  mess.m_type = DEV_IOCTL;
  mess.PROC_NR = FS_PROC_NR;
  mess.REQUEST = MIOCSPSINFO;
  mess.ADDRESS = (void *) fproc;
  (void) sendrec(MEM, &mess);
}
```

BUF_POOL

Lleva a cabo la inicialización de la cache de bloques del file system, formando las listas doblemente encadenada y hash.

ALGORITMO

1. Inicialización de variables.
2. Creación de la lista doblemente encadenada, inicializando a nulo para cada uno de los buffers los campos siguientes a libres: b_DEV (número de dispositivo donde se encuentra el bloque) y b_BLOCKNR (número de bloque en el dispositivo).
3. Encadenar la lista hash.

Parámetros de entrada/salida : Ninguno.

Procedimientos utilizados:

- **get_physbase ():** (interno). Obtiene la dirección física de comienzo del F.S.

```
/*===========================================================================*
 *                        buf_pool                                           *
*===========================================================================*/
PRIVATE void buf_pool()
{
/* Initialize the buffer pool. */

  register struct buf *bp;

  bufs_in_use = 0;
  front = &buf[0];
  rear = &buf[NR_BUFS - 1];

  for (bp = &buf[0]; bp < &buf[NR_BUFS]; bp++) {
      bp->b_blocknr = NO_BLOCK;
      bp->b_dev = NO_DEV;
      bp->b_next = bp + 1;
      bp->b_prev = bp - 1;
  }
  buf[0].b_prev = NIL_BUF;
  buf[NR_BUFS - 1].b_next = NIL_BUF;

  for (bp = &buf[0]; bp < &buf[NR_BUFS]; bp++) bp->b_hash = bp->b_next;
  buf_hash[0] = front;
}
```

LOAD_RAM

Cuando el MINIX se inicia, el único dispositivo presente es el dispositivo raíz, que es aquél donde reside el sistema de ficheros raíz. Este sistema contendrá normalmente los directorios estándares tales como:

/dev Contiene los archivos especiales de bloques y caracteres (dispositivos de E/S).
/bin Programas ejecutables (binarios)más importantes.
/etc Diversos archivos y programas para la administración del sistema.
/tmp Archivos temporales.
/usr Donde se monta el disco del sistema.
/user Para montar el sistema de ficheros del usuario.

El sistema de ficheros raíz puede residir en disquete, en una partición del disco duro o en disco RAM: Si reside en disco RAM, deberá disponerse de una copia del disco, de manera que una vez concluida la carga del sistema, dicha copia sea trasladada al disco RAM, antes de que aparezca por pantalla el prompt LOGIN. Esta copia (imagen del disco RAM) puede residir en un floppy o en una partición de disco duro - partición 3 - siendo ésta la más usada. Si se quiere ubicar en cualquier otra zona, deberá modificarse RAM_IMAGEN en fs/main.c y recompilar el sistema operativo.

Hay que resaltar el hecho de que concluido el MINIX, si se han realizado cambios en el disco RAM, éstos se perderán ya que no se vuelve a realizar una copia del disco RAM hacia la partición (o floppy). Esta es la razón por la que la mayoría de los usuarios almacenan en el disco RAM los ficheros binarios del sistema, librerías, compilador, es decir aquellos programas que raramente serán modificados.

A nivel general podemos describir la función LOAD_RAM diciendo que realiza una copia completa del sistema de ficheros root desde el dispositivo imagen en el que está almacenado (por ejemplo un floppy) al disco RAM siempre que se haya especificado que el disco RAM es el dispositivo root. En caso contrario tan solo habilita espacio para el disco RAM en la cantidad que indican los parámetros de arranque (boot).

ALGORITMO

1. Asigna a ram_size el valor dado en los parámetros de arranque.
2. Se abre el dispositivo root, donde se va a alojar la imagen del sistema de ficheros root.
3. Si el dispositivo root es el DISCO RAM
   3.1. Se abre el dispositivo que tiene la imagen del sistema de ficheros root.
   3.2. Se obtiene el tamaño del sistema de ficheros root a partir del superbloque.
   3.3. Se ajusta el tamaño del disco RAM al tamaño leido con anterioridad (cambia el valor de ram_size)
4. Se llama al manejador del DISCO RAM para que este tenga el tamaño indicado por ram_size.
5. Se informa al MM del tamaño del DISCO RAM.
6. Si sistema configurado para tener una segunda cache de bloques esta estará en el DISCO RAM.
7. Si el dispositivo raiz no es el DISCO RAM retornar...
8. Copia de los bloques uno a uno desde el dispositivo imagen al DISCO RAM.

Parámetros de entrada: Ninguno.

Parámetros de salida: Ninguno.

Procedimientos que utiliza:

- **get_block (dev, block, only-search):** (Externo, fs\cache.c). Adquiere bloques de datos, para lectura o escritura.
- **put_block (bp, block-type):** (Externo, fs\cache.c). Devuelve un bloque a la lista de bloques disponibles.
- **panic:** (externo, fs\utility.c).
- **sendrec (dest, &m):** (externo).

DISCO RAM : Porción preasignada de la memoria central (ver figura 1) para almacenar bloques. Asignando memoria principal como disco RAM permite obtener un alto rendimiento ya que esta memoria operará como un disco de alta velocidad y baja latencia y podrá ser usado para guardar los programas binarios, ficheros de usuario u otros datos a los cuales se quieren acceder con rapidez y frecuencia. Para emplear este disco RAM , deberá decidirse cuánta memoria principal se asignará para ejecutar programas y cuánta para disco RAM - cuanto más grande sea la memoria para programas, más programas podrán ejecutarse a la vez

Figura 1

```
/*===========================================================================*
 *                        load_ram                                           *
 *===========================================================================*/
PRIVATE void load_ram()
{
/* If the root device is the RAM disk, copy the entire root image device
 * block-by-block to a RAM disk with the same size as the image.
 * Otherwise, just allocate a RAM disk with size given in the boot
 * parameters.
 */

  register struct buf *bp, *bp1;
  long k_loaded, lcount;
  u32_t ram_size, fsmax;
  zone_t zones;
  struct super_block *sp, *dsp;
  block_t b;
  int major, task;
  message dev_mess;

  ram_size = boot_parameters.bp_ramsize;

  /* Open the root device. */
  major = (ROOT_DEV >> MAJOR) & BYTE;     /* major device nr */
  task = dmap[major].dmap_task;           /* device task nr */
  dev_mess.m_type = DEV_OPEN;       /* distinguish from close */
  dev_mess.DEVICE = ROOT_DEV;
  dev_mess.COUNT = R_BIT|W_BIT;
  (*dmap[major].dmap_open)(task, &dev_mess);
  if (dev_mess.REP_STATUS != OK) panic("Cannot open root device",NO_NUM);

  /*If the root device is the ram disk then fill it from the image device*/
  if (ROOT_DEV == DEV_RAM) {
      major = (IMAGE_DEV >> MAJOR) & BYTE;      /* major device nr */
      task = dmap[major].dmap_task;       /* device task nr */
      dev_mess.m_type = DEV_OPEN;         /* distinguish from close */
      dev_mess.DEVICE = IMAGE_DEV;
      dev_mess.COUNT = R_BIT;
      (*dmap[major].dmap_open)(task, &dev_mess);
      if (dev_mess.REP_STATUS != OK) panic("Cannot open root device",
NO_NUM);

      /* Get size of RAM disk by reading root file system's super block. */
      sp = &super_block[0];
      sp->s_dev = IMAGE_DEV;
      if (read_super(sp) != OK) panic("Bad root file system", NO_NUM);

      lcount = sp->s_zones << sp->s_log_zone_size;/* # blks on root dev*/

      /* Stretch the RAM disk file system to the boot parameters size, but
       * no further than the last zone bit map block allows.
       */
      if (ram_size < lcount) ram_size = lcount;
      fsmax = (u32_t) sp->s_zmap_blocks * CHAR_BIT * BLOCK_SIZE;
      fsmax = (fsmax + (sp->s_firstdatazone-1)) << sp->s_log_zone_size;
      if (ram_size > fsmax) ram_size = fsmax;
  }

  /* Tell RAM driver how big the RAM disk must be. */
  m1.m_type = DEV_IOCTL;
  m1.PROC_NR = FS_PROC_NR;
  m1.REQUEST = MIOCRAMSIZE;
  m1.POSITION = ram_size;
  if (sendrec(MEM, &m1) != OK || m1.REP_STATUS != OK)
      panic("Can't set RAM disk size", NO_NUM);
```

```
  /* Tell MM the RAM disk size, and wait for it to come "on-line". */
  m1.m1_i1 = ((long) ram_size * BLOCK_SIZE) >> CLICK_SHIFT;
  if (sendrec(MM_PROC_NR, &m1) != OK)
      panic("FS can't sync up with MM", NO_NUM);

#if ENABLE_CACHE2
  /*The RAM disk is a second level block cache while not otherwise used. */
  init_cache2(ram_size);
#endif

  /* If the root device is not the RAM disk, it doesn't need loading. */
  if (ROOT_DEV != DEV_RAM) return;

  /* Copy the blocks one at a time from the image to the RAM disk. */
  printf("Loading RAM disk.\33[23CLoaded:    0K ");

  inode[0].i_mode = I_BLOCK_SPECIAL;      /* temp inode for rahead() */
  inode[0].i_size = LONG_MAX;
  inode[0].i_dev = IMAGE_DEV;
  inode[0].i_zone[0] = IMAGE_DEV;

  for (b = 0; b < (block_t) lcount; b++) {
      bp = rahead(&inode[0], b, (off_t)BLOCK_SIZE * b, BLOCK_SIZE);
      bp1 = get_block(ROOT_DEV, b, NO_READ);
      memcpy(bp1->b_data, bp->b_data, (size_t) BLOCK_SIZE);
      bp1->b_dirt = DIRTY;
      put_block(bp, FULL_DATA_BLOCK);
      put_block(bp1, FULL_DATA_BLOCK);
      k_loaded = ( (long) b * BLOCK_SIZE)/1024L;      /* K loaded so far */
      if (k_loaded % 5 == 0) printf("\b\b\b\b\b\b%5ldK ", k_loaded);
  }

  printf("\rRAM disk loaded.\33[K\n\n");

  /* Close and invalidate image device. */
  dev_mess.m_type = DEV_CLOSE;
  dev_mess.DEVICE = IMAGE_DEV;
  (*dmap[major].dmap_close)(task, &dev_mess);
  invalidate(IMAGE_DEV);

  /* Resize the RAM disk root file system. */
  bp = get_block(ROOT_DEV, SUPER_BLOCK, NORMAL);
  dsp = (struct super_block *) bp->b_data;
  zones = ram_size >> sp->s_log_zone_size;
  dsp->s_nzones = conv2(sp->s_native, (u16_t) zones);
  dsp->s_zones = conv4(sp->s_native, zones);
  bp->b_dirt = DIRTY;
  put_block(bp, ZUPER_BLOCK);
}
```

LOAD_SUPER

Como hemos visto en el procedimiento fs_init, tras la llamada a la rutina load_ram(), se realizaba la llamada a la rutina load_super() pasandole el número del dispositivo donde se encontraba el sistema de ficheros raiz. Las tareas que realiza son las siguientes:

ALGORITMO

1.- Inicializa la tabla de superbloque.

2.- Lee el superbloque del sistema de ficheros raíz.

4.- Comprueba la consistencia del sistema de ficheros raíz.

5.- Termina de cargar configurar el superbloque en memoria del root file system Toma el nodo-i del directorio raíz (get_inode)..

Parámetros de entrada: Nº dispositivo donde se encuentra el superbloque del sistema de archivos.

Parámetros de salida: Ninguno.

Procedimientos que utiliza :

- **read_super (sp):** Realiza la lectura de un superbloque.
- **get_inode (dev, numb):** (externo, fs\inode.c). Halla una entrada en la tabla de nodo-i, carga el nodo-i especificado en ella y devuelve un puntero a esa entrada. En caso que el 'dev' sea NO_DEV, simplemente devuelve la entrada libre.
- **dup_inode (ip):** (externo, fs\inode.c). Duplica el nodo-i que se le pasa como argumento.

```
/*===========================================================================*
 *                        load_super                                         *
 ===========================================================================*/
PRIVATE void load_super(super_dev)
dev_t super_dev;              /* place to get superblock from */
{
  int bad;
  register struct super_block *sp;
  register struct inode *rip;

  /* Initialize the super_block table. */
  for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++)
      sp->s_dev = NO_DEV;

  /* Read in super_block for the root file system. */
  sp = &super_block[0];
  sp->s_dev = super_dev;

  /* Check super_block for consistency (is it the right diskette?). */
  bad = (read_super(sp) != OK);
  if (!bad) {
      rip = get_inode(super_dev, ROOT_INODE);   /* inode for root dir */
      if ( (rip->i_mode & I_TYPE) != I_DIRECTORY || rip->i_nlinks < 3)
bad++;
  }
  if (bad)panic("Invalid root file system.  Possibly wrong
diskette.",NO_NUM);

  sp->s_imount = rip;
  dup_inode(rip);
  sp->s_isup = rip;
  sp->s_rd_only = 0;
  return;
}
```

GET_BOOT_PARAMETERS

Este procedimiento crea un mensaje del tipo SYS_GBOOT para enviarlo al manejador de tareas del sistema, solicitando una copia de los parámetros del boot. Estos se cargarán en la estructura boot_parameters.

Parámetros de entrada y/o salida : Ninguno .

Procedimientos que utiliza :

- **Sendrec(destino, &m):** (Externo) Envía un mensaje a la tarea del sistema y espera la respuesta.

Toma el nodo-i del directorio raíz (get_inode). Toma el nodo-i del directorio raíz (get_inode).

```
/*===========================================================================*
 *                        get_boot_parameters                                *
 ============================================================================*/
PUBLIC struct bparam_s boot_parameters;

PRIVATE void get_boot_parameters()
{
/* Ask kernel for boot parameters. */

  m1.m_type = SYS_GBOOT;
  m1.PROC1 = FS_PROC_NR;
  m1.MEM_PTR = (char *) &boot_parameters;
  (void) sendrec(SYSTASK, &m1);
}
```

CUESTIONES

1. ¿Cuál es la misión de la función MAIN?
2. ¿Para que sirve la tabla DMAP?
3. ¿Para que sirve la tabla CALL_VECTOR?
4. ¿Para que sirve la tabla FPROC?