# Indice

## 1.INTRODUCCIÓN AL DISPOSITIVO DE DISCO DE RAM.

El objetivo de un dispositivo Disco RAM es acelerar los accesos que se hagan al trozo del file system que se asocia a este dispositivo, ya que al soportarse éste sobre la memoria principal, los accesos de R/W serán, lógicamente, más rápidos, respecto a cualquier otro dispositivo físico.

El disco de RAM emula todas las propiedades de los dispositivos de bloque en general. Un dispositivo de bloque es un medio de almacenamiento con dos operaciones: *escritura* y *lectura* de un bloque. Normalmente, estos bloques se almacenan en periféricos como discos duros o flexibles, pero en este caso se almacenarán en una porción de la memoria central previamente asignada, por lo que su manejo es mucho más simple.

**Ventajas y Desventajas.**

Ventajas:

Tiene acceso instantáneo, ya que no es necesario la localización de la pista y sector en la que se encuentra el dato. Esto lo hace adecuado para almacenar programas o datos que sean accedidos con frecuencia, como por ejemplo los del sistema.

El sistema operativo MINIX fue diseñado para trabajar en ordenadores con sólo un disco flexible, (por ejemplo: un 8086 sin disco duro y con sólo una disquetera), y el problema que se presenta al no colocar el dispositivo raíz en el disco Ram, es que no se puede retirar el disco flexible de la unidad (el dispositivo raíz no puede ser desmontado), ya que perdemos la posibilidad de hacer referencia a todo el file system al no disponer de la tabla de directorio que estaba en el diskette. Además, al tener el dispositivo raíz en el disco de RAM el sistema se vuelve altamente flexible porque cualquier combinación de discos duros o flexibles puede montarse en él.

Desventajas:

Al destinar una parte de la memoria principal para el dispositivo, disminuye la cantidad de ésta que queda disponible para programas de usuario.

Como consecuencia de la disminución de la memoria disponible hay un mayor tráfico de información con la memoria secundaria, debido a que se producirán más fallos de página. Esto provoca que se degrade el rendimiento de la máquina.

## 2.SISTEMAS MONTADOS VS NO MONTADOS.

Señalaremos brevemente la diferencia que existe entre los sistemas que tienen sistemas de archivo montados y aquellos que no. Con los sistemas de archivo montados, el dispositivo raíz siempre está presente y en una localización fija, y los sistemas removibles (discos) pueden montarse en el árbol de archivos para formar un sistema de archivo integrado, es decir, el sistema de archivos del disco sería un directorio más del raíz por lo que el usuario no necesita preocuparse en absoluto por el dispositivo en el cual se encuentra el archivo.

Por el hecho de considerar a todos los dispositivos del sistema (red, floppy...) parte del sistema de archivos, estructurado como árbol, podemos usar los mismos comandos para las operaciones de escritura y de lectura, aunque el file system internamente llame a unas subrutinas específicas para acceder a cada dispositivo.

Un ejemplo que aclara esto es el hecho de que en el MINIX se pueda leer de floppy con el comando *dosread* sin montarlo en un subdirectorio del árbol del sistema de archivos principal, mientras que si se pretende utilizar el comando cp, usado a nivel de usuario para la copia de cualquier fichero, es necesario montar el floppy ensamblándolo al sistema de archivos.

En contraste tenemos a los sistemas no montados como MS-DOS, donde el usuario debe especificar la localidad de cada archivo, ya sea en forma explícita como en B:FICHERO o mediante ciertas omisiones (paths por defecto). Con sólo uno o dos discos flexibles esta tarea es abordable, pero en sistemas de computación grande con docenas de discos, teniendo que llevar el control de los dispositivos todo el tiempo sería muy pesado.

## 3.LA MEMORIA Y LA ORGANIZACIÓN DEL DISCO RAM.

La idea de como es un disco RAM se muestra en la siguiente figura.


El disco RAM se divide en *n* bloques, según la cantidad de memoria que se le haya asignado por parte del sistema de archivos. Cada bloque es del mismo tamaño que el tamaño de bloque que se utiliza en los discos reales. Cuando el manejador recibe un mensaje para leer o escribir un bloque, simplemente determina el sitio en la memoria del disco de RAM donde se encuentra el bloque y lee o escribe en él, y no en un disco duro o flexible. Normalmente la transferencia se hará llamando a un procedimiento en ensamblador **(phys_copy)** que copia en o del programa de usuario a la máxima velocidad que el hardware sea capaz.

## 4.MANEJADOR DEL DISCO RAM.

El manejador que vamos a describir se encarga de la gestión de cuatro dispositivos menores que están muy relacionados en cuanto a su tratamiento se refiere. Los dispositivos menores son los siguientes:

*/dev/ram 1: /dev/mem 2: /dev/kmem 3: /dev/null*

**/dev/ram** Es el disco de RAM verdadero es decir, que en él podemos crear un file system. Ni su tamaño ni su origen están integrados en el manejador, ya que estos son determinados por el sistema de archivos al cargar MINIX. Esta estrategia hace posible incrementar o reducir la cantidad del disco de RAM que está presente sin tener que recompilar el sistema operativo. Todo lo que uno necesita hacer es utilizar un disquete del sistema de fichero raíz diferente.

**/dev/mem** Se utiliza para leer y escribir la memoria física en su totalidad. Comienza en la posición 0 absoluto de la memoria, incluyendo al vector de interrupciones, que de esta manera puede ser alterado. Por este motivo, este dispositivo está protegido, pudiendo únicamente ser utilizado por el superusuario.

**/dev/kmem** Se utiliza para leer y escribir en la memoria del kernel. El byte 0 de este fichero es el byte 0 de la memoria de datos del kernel, una localización cuya dirección absoluta varía dependiendo del tamaño del código del kernel de Minix. Por ejemplo, en el Minix 1.5 y según la memoria realizada en el 96, es la posición 1536 de la memoria física. Tanto este dispositivo como el anterior se utilizan para depuración y programas muy especiales. Obsérvese que las áreas del disco de RAM cubiertas por estos dos dispositivos menores se superponen. La razón de ser del */dev/kmem* radica en que si en futuras versiones de Minix el desplazamiento de la memoria del kernel ya no es 1536 (para el Minix 1.5), los programas que accedían a través de */dev/kmen* seguirán funcionando, mientras que aquellos que accedían a través de */dev/mem* indicando el desplazamiento fallarán. Este dispositivo también se encuentra protegido.

**/dev/null** Es un fichero especial que acepta datos y los desecha. El controlador del disco de RAM lo trata como si fuera de tamaño cero, de manera que

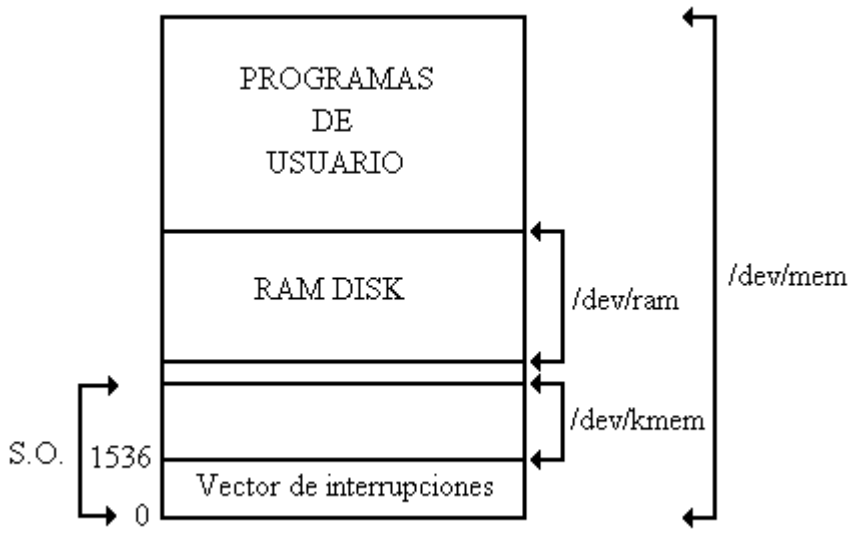
nunca se copien datos en o de él. Se utiliza comúnmente en comandos del shell cuando el programa solicitado genera una salida que no se necesita. Por ejemplo:

```
a.out >/dev/null
```

ejecuta el programa **a.out**, pero desprecia su salida.

En la versión Minix 1.5 se gestinaba un dispositivo menor adicional, llamado /dev/port, que trataba las entradas/salidas por puerto. Sólo se utilizaba en el caso de que el procesador fuera Intel. Esto es debido a que en el 68000 no hay diferencia entre acceso a E/S y memoria (se hace con las mismas instrucciones), mientras que en el 8086 para realizar la E/S se accede a puerto a través de las instrucciones *in* y *out.* Esta es la razón por la que había que añadir un dispositivo más para el 8086, mientras que con el 68000 el problema quedaba resuelto con los anteriores dispositivos. En la versión 2.0 del Minix, no se toma como tarea de memory.c la gestión de este dispositivo.

En la siguiente figura podemos encontrar la distribución de los dispositivos antes mencionados para la versión 1.5 del Minix (en otras versiones /dev/kmem no tiene porqué comenzar en la posición 1536):



## 5. ESTRUCTURAS DE DATOS QUE SE UTILIZAN.

**m_geom:** Es un vector de elementos de tipo device. El tipo device es una estructura cuyos campos son base y tamaño (del dispositivo). Por tanto, en este vector se almacenarán la base y el tamaño de los 4 dispositivos tratados en memory.c .

**m_dtab:** Es una estructura de tipo driver (declarada en driver.c) cuyos campos son los procedimientos de memory.c que deberá usar driver_task para gestionar las peticiones referidas a los dispositivos de memoria.

**m_device:** variable que indica el dispositivo actual.

**Estructura iorequest_s**: El procedimiento m_schedule recibe un parámetro, llamado iop, de este tipo. Es una estructura similar a la estructura message, pero con menos campos:

- io_request: Contiene el tipo de solicitud. Se corresponde con el campo m_type de la estructura message.

-io_buf: Contiene la Dirección virtual del buffer del espacio de datos del proceso que solicita la operación. Se corresponde con el campo address de la estructura message.

-io_position: Se corresponde con el campo position de message. Contiene el desplazamiento dentro del dispositivo menor, es decir, la posición del primer byte de dispositivo de memoria a transferir.

*message:* Una variable de este tipo es usada por driver.c, que la pasa como parámetro a algunos procedimientos de memory.c para que este compruebe algunos de sus campos. Memory.c sólo usa mensajes de petición, ya que el que se encarga de recibir mensajes y enviar las respuestas es el driver.c. Los campos de un mensaje de tipo petición son:

int m_source: Quién envió el mensaje.

int m_type: Tipo de mensaje u operación solicitada.

DISK_READ
DISK_WRITE
DISK_IOCTL
SCATTERED_IO

int DEVICE: Número de dispositivo menor.

int PROC_NR: Número de proceso que solicita una E/S.

int COUNT:

R/W -> Número de bytes a transferir.
IOCTL -> Número de bloques del dispositivo menor.
SCATTERED_IO -> Número de peticiones.

long POSITION:

R/W -> Offset dentro del dispositivo menor.
IOCTL -> Origen del disco RAM.

char *ADDRESS: Dirección virtual del buffer del espacio de datos del proceso que solicita la operación.

## 6.IMPLEMENTACIÓN DEL MANEJADOR DEL DISCO RAM.

A continuación vamos a presentar los procedimientos que implementan el manejador de disco RAM.

### *mem_task()*

Es el programa principal,que no tiene parámetros de entrada ni de salida. Lo Primero que hace es llamar al procedimiento m_init (incluido en memory.c), que hace la inicialización de los dispositivos /dev/mem y /dev/kmem.

Después se llama al procedimiento driver_task (incluido en driver.c), que se mete en un bucle infinito en el que va recibiendo mensajes y llamando a las funciones correspondientes según el tipo de mensaje para, a continuación enviar el mensaje de respuesta al proceso que había mandado el mensaje de solicitud. Se le pasa como parámetro una estructura (de tipo driver, que está declarada en driver.c) que contiene los

nombres de procedimientos concretos a los que ha de llamar para nuestro caso (memory.c). Estos procedimientos son los que a continuación se explican:

**- m_init**: Es el procedimiento que efectua la inicialización, en la que se obtiene el origen y el límite del dispositivo */dev/kmem* y el límite del dispositivo */dev/mem.* El tamaño de este último dispositivo (y por tanto su límite) lo pondrá a 1Mb, 16Mb ó 4Gb-1 dependiendo si se está corriendo en modo 8088, 80286, 80386. Estos tamaños son los máximos soportados por Minix y no tienen nada que ver con cuanta RAM se instala en la máquina.

**- m_prepare**: Este procedimiento prepara al dispositivo para ser usado. Lo que hace es comprobar que el numero de dispositivo menor que ha sido requerido es correcto (está entre 0 y 3). Luego pone como dispositivo actual (m_device) el dispositivo a preparar. El procedimiento retorna la dirección de la estructura que guarda la dirección base y el tamaño del área de RAM requerida (es decir la posición correspondiente a ese dispositivo del vector *m_geom[ ]*).

**- m_schedule*:***

Es el procedimiento encargado de la lectura y escritura de bloques. Tiene dos parámetros de entrada, el número de proceso que ha hecho la petición (proc_nr) y un puntero a la petición de lectura o escritura (iop). Este último parámetro es de tipo iorequest_s y es una estructura similar a la de los mensajes pero con menos campos. Los pasos que sigue son los siguientes:

Determina el número de dispositivo menor accediendo a la variable m_device, que es el número de dispositivo menor actual.

Calcula, mediante la función numap, la dirección física de memoria del usuario (user_phys) donde serán escritos los datos o de donde serán leídos.

A continuación comprueba si el dispositivo es /dev/null, en cuyo caso no realiza ninguna operación.

Si el dispositivo no era */dev/null,* calcula el comienzo y límite de la memoria que interviene en la transferencia, teniendo en cuenta que no sobrepasa los límites de la

memoria del dispositivo correspondiente. Si la posición de lo que se quiere leer o escribir está en el rango, pero el número de bytes (a leer o escribir) lo sobrepasa, sólo permite transferir el número de bytes entre la posición y el límite superior del dispositivo de memoria correspondiente.

Determina la dirección física (mem_phys) en donde se escribirán los datos o se leerán, a partir de la dirección virtual.

Si la operación es de lectura, copia los datos de la dirección *mem_phys* a la dirección *user_phys*. Si es de escritura, al revés. Esto lo hace mediante la función *phys_copy*.

El número de bytes que fueron leídos lo devuelve en el parámetro iop (concretamente en iop->io_nbytes).

**- m_do_open**: Este procedimiento recibe dos parámetros. El primero (*dp*) es una estructura de tipo driver, y no es usado para nada por este procedimiento. ¿Por qué está ahí entonces? Suponemos que por que habrá otros dispositivos que si lo usen, y hay que guardar las formas con respecto a driver.c (recordemos que la estructura driver viene declarada en driver.c, y tiene como campos procedimientos con una serie de parámetros dados). El segundo parámetro es *m_ptr*, que es de tipo mensaje.

Lo primero que hace es comprobar que el número de dispositivo menor requerido (es uno de los campos de *m_ptr*) es correcto mediante una llamada a m_prepare. Si el chip es Intel comprueba además si se requiere el /dev/mem o el /dev/kmem. Si es así llama al procedimiento *enable_iop* ,pasando como parámetro la dirección de la ranura de la tabla de procesos correspondiente al proceso que hizo la solicitud (es otro campo de *m_ptr*). Esta función activa el bit correspondiente de esa ranura que permite a ese proceso acceder a los puertos de E/S.

**- m_ioctl**: Recibe como parámetros una estructura driver que al igual que *m_schedule*, no usa. Además recibe un parámetro de tipo mensaje llamado *m_ptr*.

En el mensaje hay un campo (*m_ptr->REQUEST*) que puede tomar los siguientes valores:

- MIOCRAMSIZE: Selecciona el tamaño de disco RAM (dispositivo /dev/ram). Esta operación sólo puede ser realizada por el File System. En esta operación, lo primero que hace es comprobar que el proceso que ha hecho la petición es el File System. El campo POSITION de m_ptr contendrá el número de bloques que deberá tener el disco RAM. Con ello calculará su tamaño en bytes. A continuación buscará un hueco en memoria lo suficientemente grande para albergar el disco RAM. Luego actualiza la base y el tamaño (*m_geom[ ]*) del dispostivo RAM.

- MIOCSPSINFO: esta opción es usada por el File System y por el manejador de memoria para poner las direcciones de sus entradas correspondientes de la tabla de procesos en la tabla de información del *ps* (psinfo), donde este programa (*ps*) puede recuperarlas usando la operación MIOCGPSINFO.

**- m_geometry**: Los dispositivos de memoria no tienen geometría, es decir, carecen de elementos como cilindros, sectores y pistas. Sin embargo, este procedimiento se añade en memory.c para guardar una concordancia con el resto de dispositivos.

## 7.PROCEDIMIENTOS QUE SE UTILIZAN.

*numap (int proc_nr, vir_bytes vir_address, vir_bytes bytes)*

Calcula la dirección de memoria física para una dirección virtual dada.

*phys_copy (phys_bytes dir_phys1, phys_bytes dir_phys2, int count)*

Procedimiento ensamblador que copia un bloque de memoria física. *dir_phys1* es la localidad física de la cual vendrá el bloque a copiar y di*r_phys2* es la localidad física a la cual se dirigirá el bloque.

*vir2phys( vir)*

Procedimiento que recibe un entero, *vir*, al que le suma la dirección base de la parte del kernel en memoria(*data_base*). Es usada por *m_init* para inicializar la base y el tamaño del dispositivo /dev/kmem.

# 8.CÓDIGO FUENTE DE MEMORY.C:

```
/* This file contains the device dependent part of the drivers for the
 * following special files:
 *     /dev/null          - null device (data sink)
 *     /dev/mem                   - absolute memory
 *     /dev/kmem          - kernel virtual memory
 *     /dev/ram           - RAM disk
 *
 * El fichero tiene un punto de entrada:
 *          mem_task:    entrada principal cuando el sistema es arrancado.
 */

#include "kernel.h"
#include "driver.h"
#include <sys/ioctl.h>

#define NR_RAMS          4  /* número de dispositivos de memoria*/

PRIVATE struct device m_geom[NR_RAMS];    /*Base y tamaño de cada disco RAM*/
PRIVATE int m_device;                  /* dispositivo actual */

FORWARD _PROTOTYPE( struct device *m_prepare, (int device) );
FORWARD _PROTOTYPE( int m_schedule, (int proc_nr, struct iorequest_s *iop) );
FORWARD _PROTOTYPE( int m_do_open, (struct driver *dp, message *m_ptr) );
FORWARD _PROTOTYPE( void m_init, (void) );
FORWARD _PROTOTYPE( int m_ioctl, (struct driver *dp, message *m_ptr) );
FORWARD _PROTOTYPE( void m_geometry, (struct partition *entry) );


/* Puntos de entrada a este dispositivo. */
PRIVATE struct driver m_dtab = {
  no_name,      /* nombre del dispositivo actual */
  m_do_open,  /* para abrir el dispositivo */
  do_nop,       /* al cerrar no hace nada */
  m_ioctl,      /* Especifica el tamaño del disco RAM */
  m_prepare,    /* prepara un dispositivo menor para la E/S */
  m_schedule,  /* hace la E/S */
  nop_finish,    /* no hay que hacer nada al finalizar */
  nop_cleanup, /* nada está sucio */
  m_geometry,  /* “geometría” del dispositivo de memoria */
};




/*===========================================================*
 *                                mem_task                                                *
*==== ==========================================================*/
PUBLIC void mem_task()
```

```
{
  m_init();
  driver_task(&m_dtab);
}


/*===========================================================*
 *                          m_prepare                        *
 *===========================================================*/
PRIVATE struct device *m_prepare(device)
int device;
{
/* Prepara el dispositivo para E/S. */

  if (device < 0 || device >= NR_RAMS) return(NIL_DEV);/*Comprueba que el
dispositivo
                                                        es correcto*/
  m_device = device;

  return(&m_geom[device]);
}


/*===========================================================*
 *                          m_schedule                       *
 *===========================================================*/
PRIVATE int m_schedule(proc_nr, iop)
int proc_nr;                    /* proceso que hizo la petición */
struct iorequest_s *iop;        /* puntero a la petición de lectura o escritura */
{
/* Lee o escribe /dev/null, /dev/mem, /dev/kmem, ó /dev/ram. */

  int device, count, opcode;
  phys_bytes mem_phys, user_phys;
  struct device *dv;

  /* Tipo de petición (lectura o escritura)*/
  opcode = iop->io_request & ~OPTIONAL_IO;

  /* Obtiene el número de dispositivo menor. */
  device = m_device;
  dv = &m_geom[device];

  /* Determina la dirección donde los datos van o de donde vendrán*/
  user_phys = numap(proc_nr, (vir_bytes) iop->io_buf,
                                          (vir_bytes) iop->io_nbytes);
  if (user_phys == 0) return(iop->io_nbytes = EINVAL);

/* Comprueba si el dispositivo es /dev/null */
  if (device == NULL_DEV) {
```

```
        if (opcode == DEV_WRITE) iop->io_nbytes = 0;
        count = 0;
  } else {
        /* /dev/mem, /dev/kmem, ó /dev/ram: Comprueba si sobrepasa los límites del */
        /* dispositivo de memoria correspondiente*/
        if (iop->io_position >= dv->dv_size) return(OK);
        count = iop->io_nbytes;
        if (iop->io_position + count > dv->dv_size)
                count = dv->dv_size - iop->io_position;/*Sólo permite transferir hasta el
límite                                                  del dispositivo de memoria*/
  }

  /* Calcula 'mem_phys' para  /dev/mem, /dev/kmem, ó /dev/ram */
  mem_phys = dv->dv_base + iop->io_position;

  /* Almacena el número de bytes que quedarán por transferir. */
  iop->io_nbytes -= count;

  if (count == 0) return(OK);

  /* Copia los datos. */
  if (opcode == DEV_READ)
        phys_copy(mem_phys, user_phys, (phys_bytes) count);
  else
        phys_copy(user_phys, mem_phys, (phys_bytes) count);

  return(OK);
}
```

```
/*===========================================================*
 *                          m_do_open                        *
 *===========================================================*/
PRIVATE int m_do_open(dp, m_ptr)
struct driver *dp;
message *m_ptr;
{
/* Comprueba el número de dispositivo a abrir. Otorga privilegios de E/S a los procesos
que abren  /dev/mem ó /dev/kmem. */

 if (m_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
/*m_prepare devuelve NIL_DEV si el dispositivo no está en rango*/

#if (CHIP == INTEL)
  if (m_device == MEM_DEV || m_device == KMEM_DEV)
        enable_iop(proc_addr(m_ptr->PROC_NR));
        /*Habilita al proceso para usar instrucciones de direccionamiento de puertos de
E/S*/
#endif

  return(OK);
}


/*===========================================================*
 *                          m_init                           *
 *===========================================================*/
PRIVATE void m_init()
{
  extern int _end;

  m_geom[KMEM_DEV].dv_base = vir2phys(0);
  m_geom[KMEM_DEV].dv_size = vir2phys(&_end);

#if (CHIP == INTEL)
  if (!protected_mode) {
        m_geom[MEM_DEV].dv_size =   0x100000;          /* 1M para sistemas 8086 */
  } else {
#if _WORD_SIZE == 2
        m_geom[MEM_DEV].dv_size =  0x1000000;          /* 16M para sistemas 286*/
#else
        m_geom[MEM_DEV].dv_size = 0xFFFFFFFF;          /* 4G-1 para sistemas 386*/
#endif
  }
#else /* !(CHIP == INTEL) */
#if (CHIP == M68000)
  m_geom[MEM_DEV].dv_size = MEM_BYTES;
#else /* !(CHIP == M68000) */
```

```
#error /*límites de memoria no han sido inicializados (el procesador no es Intel ni
M68000  */
#endif /* !(CHIP == M68000) */
#endif /* !(CHIP == INTEL) */
}


/*===========================================================*
 *                                m_ioctl                                    *
 *===========================================================*/
PRIVATE int m_ioctl(dp, m_ptr)
struct driver *dp;
message *m_ptr;                         /* puntero al mensaje. */
{
  unsigned long bytesize;
  unsigned base, size;
  struct memory *memp;
  static struct psinfo psinfo = { NR_TASKS, NR_PROCS, (vir_bytes) proc, 0, 0 };
  phys_bytes psinfo_phys;

  switch (m_ptr->REQUEST) {
case MIOCRAMSIZE:
        /* FS selecciona el tamaño del disco RAM. */
        if (m_ptr->PROC_NR != FS_PROC_NR) return(EPERM);

        /*POSITION es el nº de bloques que habrá de tener el disco RAM*/
        bytesize = m_ptr->POSITION * BLOCK_SIZE;
        size = (bytesize + CLICK_SHIFT-1) >> CLICK_SHIFT;

        /* Busca un hueco de memoria lo suficientemente grande para el disco RAM. */
        memp= &mem[NR_MEMS];
        while ((--memp)->size < size) {
                if (memp == mem) panic("RAM disk is too big", NO_NUM);
                /*Si se cumple el if anterior es que no encontró huecos*/
        }
        /*Lo siguiente actualiza el vector mem[ ]*/
        base = memp->base;
        memp->base += size;
        memp->size -= size;

        /*Actualiza la base y el tamaño del dispositivo RAM*/
        m_geom[RAM_DEV].dv_base = (unsigned long) base << CLICK_SHIFT;
        m_geom[RAM_DEV].dv_size = bytesize;
        break;

  case MIOCSPSINFO:
        /*MM ó FS pone la dirección de su tabla de procesos en la tabla de informacion
de      ps (psinfo)*/
        if (m_ptr->PROC_NR == MM_PROC_NR) {
                psinfo.mproc = (vir_bytes) m_ptr->ADDRESS;
```

```
        } else
        if (m_ptr->PROC_NR == FS_PROC_NR) {
                psinfo.fproc = (vir_bytes) m_ptr->ADDRESS;
        } else {
                return(EPERM);
        }
        break;
  case MIOCGPSINFO:
        /* El programa ps requiere las direcciones de la tabla de procesos. */
        psinfo_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS,
                                                        sizeof(psinfo));
        if (psinfo_phys == 0) return(EFAULT);
        phys_copy(vir2phys(&psinfo), psinfo_phys, (phys_bytes) sizeof(psinfo));
        break;
  default: /*Si no es ninguna de las operaciones anteriores se llama al proceso
equivalente a  m_ioctl en driver.c, que implementa las operaciones  comunes a todos los
        dispositivos*/
        return(do_diocntl(&m_dtab, m_ptr));
  }
  return(OK);
}


/*============================================================*
 *                                m_geometry                  *
 *============================================================*/
PRIVATE void m_geometry(entry)
struct partition *entry;
{
  /* Los dispositivos de memoria no tienen geometría, pero el mundo exterior insiste. */
  entry->cylinders = (m_geom[m_device].dv_size >> SECTOR_SHIFT) / (64 * 32);
  entry->heads = 64;
  entry->sectors = 32;
}
```

# 9.PREGUNTAS.

**1.Ventajas y desventajas de usar disco RAM.**

Le remitimos a la página 2.

**2.¿De qué dispositivos menores se encarga el manejador de disco RAM?**

Véase la página 6.

**3.¿Qué sentido tiene introducir el dispositivo menor /dev/kmem si con el /dev/mem también podemos acceder?**

La razón de ser del */dev/kmem* radica en que si en futuras versiones de Minix el desplazamiento de la memoria del kernel ya no es 1536 (para el Minix 1.5), los programas que accedían a través de */dev/kmen* seguirán funcionando, mientras que aquellos que accedían a través de */dev/mem* indicando el desplazamiento fallarán.

**4.¿Cómo se establece el tamaño del disco RAM?**

El File Sistem lo establece mediante un mensaje de tipo DEV_IOCTL, que recoje driver_task, el cual se encarga de llamar al procedimiento de memory.c m_ioctl. Éste procedimiento se encargará de buscar un hueco en memoria donde quepa y actualizar la base y tamaño del dispositivo RAM

**5.¿Qué hace la función m_schedule?**

Esto está en la página 11.