Exposición en Clase

Controlador de Disco Duro a través del BIOS

Ampliación de Sistemas Operativos

Grupo:
Elia C. Rivero Díaz
Gabriel Dovalo Carril

1.- Introducción

El BIOS surge para facilitar las compatibilidades entre el sistema operativo y el hardware de la máquina. Es decir, el nivel de la BIOS funcionará como interfase entre ambos. El Sistema Operativo enviará la petición a la BIOS y éste se encargará de programar los chips de E/S del hardware para realizar la operación.

A un nivel muy abstracto los distintos niveles en que esto se realiza pueden verse en la

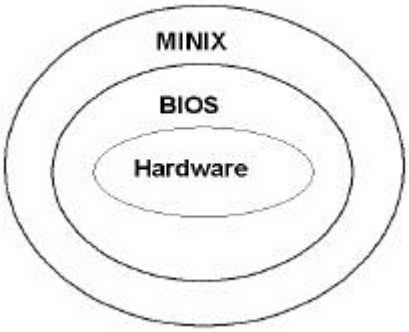


figura siguiente:

El fichero encargado de realizar la labor de comunicación entre el Sistema Operativo y el Hardware se denomina "bios_win.c". Dicho fichero contiene un controlador de disco duro que utiliza las funciones suministradas por la ROM BIOS.

Este controlador se encuentra constantemente a la espera de recibir peticiones de lectura o escritura sobre el disco. Recibida la petición contesta con una respuesta de aceptación y realiza la transferencia, si el dispositivo no es capaz de atender en ese momento la petición la encola.

Es un controlador que no funciona mediante interrupciones, y además tiene un bajo rendimiento.

La ventaja de este controlador es que trabaja en cualquier PC, XT, 80386, PS/2.

El disco genérico BOOT usa este tipo de controlador, ya que no hay otra posibilidad, puesto que en el arranque sólo disponemos de la BIOS para acceder a los dispositivos.

2.- Descripción Funcional

Podemos describir el comportamiento del fichero "bios_win.c" en una serie de fases, que pasamos a enumerar:

1. Inicialización, en esta fase se inicializan los parámetros del controlador de disco duro, esta fase tiene lugar durante el arranque del sistema.
2. Activación del controlador, bios_winchester_task, esta fase se basa en que el controlador se ponga a la espera de peticiones procedentes de procesos de niveles superiores. Es un procedimiento que simplemente hace una llamada a "driver_task", que como recordaremos de "Dispositivos por Bloque", es un procedimiento que espera una petición sobre un dispositivo y siempre está en ejecución, muriendo al mismo tiempo que muere el sistema operativo, es decir, cuando apagas el ordenador.
3. Recepción de una petición:
    1. Comprobación de que el dispositivo es válido, si no es válido debe darse un error y abortar el tratamiento de la petición
    2. Comprobación de que el dispositivo ya esté inicializado completamente, si no está inicializado debe configurarse, (apertura del dispositivo, OPEN)
    3. Planificación de la petición:
        1. Si la petición es para una zona de memoria consecutiva a la de la última petición:
            (11 Si se usa zona de almacenamiento intermedia, buffer, y no está completa se almacenará aquí la nueva petición
            (12 Si la zona de almacenamiento está completa entonces se produce la terminación de la atención a petición, y la petición actual pasa a una nueva planificación
            (13 Si no usa zona de almacenamiento intermedia, buffer, las peticiones se van encolando en un vector específico denominado "wtrans"
        2. Si la petición es para una zona de almacenamiento no consecutiva se produce la terminación de la atención a petición, y la petición actual pasa a una nueva planificación.
    4. Terminación de la atención a una petición.

3.- Estructuras de Datos del fichero "bios_win.c"

3.1.- wini

La principal estructura de datos que se utiliza en el manejador de disco duro es "wini". Esta estructura proporciona la información necesaria para llevar a cabo la operación sobre el dispositivo solicitado, así como las características que reúne ese dispositivo. Existen tantas estructuras como unidades.

Los campos que se componen la estructura son:

```
PRIVATE struct wini {
        /* estructura principal de unidad, posee una entrada por cada unidad */
        unsigned cylinders;             /* numero de cilindros */
        unsigned heads;         /* numero de cabezas */
        unsigned sectors;               /* numero de sectores por pista */
        unsigned open_ct;               /* contador de uso */
        struct device part[DEV_PER_DRIVE];              /* particiones primarias: hd[0-4] */
        struct device subpart[SUB_PER_DRIVE];   /* subparticiones: hd[1-4][a-d] */
        } wini[MAX_DRIVES], *w_wn;
```

La siguiente estructura de gran importancia es la llamada "trans", que es la que contiene la información necesaria para realizar la transferencia, y cuyos campos son:

3.2.- wtrans

```
PRIVATE struct trans {
        struct iorequest_s *iop;      /* pertenece a una petición de E/S */
        unsigned long block;          /* primer sector a transferir */
        unsigned count;              /* contador de bytes */
        phys_bytes phys;              /* dirección física del usuario */
        phys_bytes dma;               /* dirección física del DMA */
        } wtrans[NR_IOREQS];
```

3.3.- Otras macros y declaraciones.

Además el código realiza la siguiente serie de inicializaciones o macros:

```
/* Parámetro de tipo lógico según tamaño de buffer DMA:
```

Verdadero, si el tamaño del buffer asignado al DMA es suficientemente grande, en cuyo caso sera el usado siempre. ***Falso***, si el tamaño del buffer asignado al DMA es demasiado pequeño con lo que se utilizará el vector "wtrans" */

```
#define USE_BUF     (DMA_BUF_SIZE > BLOCK_SIZE)

/* Códigos de error */
#define ERR         (-1) /* Define un error de tipo general */

/* Parámetros para las unidades de disco.*/
#define MAX_DRIVES  4               /* Este driver soporta 4 unidades (hd0 - hd19) */
#define MAX_SECS        255         /* El BIOS puede transferir 255 sectores */
#define NR_DEVICES  (MAX_DRIVES * DEV_PER_DRIVE)
#define SUB_PER_DRIVE    (NR_PARTITIONS * NR_PARTITIONS)
#define NR_SUBDEVS  (MAX_DRIVES * SUB_PER_DRIVE)
```

```
/* Parámetros de la BIOS */
#define BIOS_ASK    0x08    /* Cód. de operación para pedir parámetros al BIOS */
#define BIOS_RESET 0x00   /* Cód. de operación para resetear el disco mediante el BIOS */
#define BIOS_READ 0x02    /* Cód. para realizar lecturas mediante BIOS */
#define BIOS_WRITE 0x03   /* Cód. para realizar escrituras mediante BIOS */
#define HD_CODE     0x80    /* Cód. para la unidad de disco duro 0 */
```

Tras la definición de los parámetros, se declaran variables de tipo global a todas las funciones implementadas en el fichero "bios_win.c". Las variables son las siguientes:

```
PRIVATE int nr_drives = MAX_DRIVES;    /* Número de unidades */
PRIVATE struct trans *w_tp; /* Añade peticiones al vector que se encarga de las transferencias
                                    */
PRIVATE unsigned w_count;           /* Número de bytes a transferir */
PRIVATE unsigned long w_nextblock;          /* Próximo bloque de disco para transferir */
PRIVATE int w_opcode;               /* DEV_READ o DEV_WRITE */
PRIVATE int w_drive;                /* Unidad seleccionada */
PRIVATE struct device *w_dv;        /* Tamaño y base del dispositivo */
extern unsigned Ax, Bx, Cx, Dx, Es;  /* Mantiene los registros para las llamadas al BIOS */
```

Prototipado de funciones típico de C, estas funciones son las que permiten que las capas superiores sean independientes del hardware:

```
FORWARD _PROTOTYPE ( struct device *w_prepare, (int device) );
FORWARD _PROTOTYPE ( char *w_name, (void) );
FORWARD _PROTOTYPE ( int w_schedule, (int proc_nr,struct iorequest_s *iop) );
FORWARD _PROTOTYPE ( int w_finish, (void) );
FORWARD _PROTOTYPE ( int w_do_open, (struct driver *dp,message *m_ptr) );
FORWARD _PROTOTYPE ( int w_do_close, (struct driver *dp,message *m_ptr) );
FORWARD _PROTOTYPE ( void w_init, (void) );
FORWARD _PROTOTYPE ( void enable_vectors, (void) );
FORWARD _PROTOTYPE ( void w_geometry, (struct partition *entry));
```

Estructura de tipo driver que almacenará los nombes de las funciones que utilizará el controlador, es la
estructura que será pasada a "driver_task" para determinar que rutina se atenderá en cada petición a este dispositivo:

3.4.- w_dtab

```
/* Puntos de entrada para esta unidad */
PRIVATE struct driver w_dtab = {
        w_name,         /* Nombre de la unidad actual */
        w_do_open,      /*Petición de apertura o montaje de la unidad, inicializa el dispositivo*/
        w_do_close,     /* Libera el dispositivo */
        do_diocntl,     /* Obtiene o establece la geometría de una partición */
```

```
        w_prepare,      /* Prepara un dispositivo menor para operaciones de E/S */
        w_schedule,     /* Calcula los cilindros, cabezas, sectores, etc. */
        w_finish,       /* Realizacion de la E/S */
        nop_cleanup,    /* No se necesita operación de limpieza, (recalibrado) */
        w_geometry      /* Indica la geometría del disco */
};
```

4.- Funciones de Abios_win.c@

A partir de aquí empiezan a codificarse las funciones propias del fichero "bios_win.c":

4.1.- Función bios_winchester_task:

Es la entrada principal cuando arranca el sistema operativo, llama a "driver_task", pasándole la estructura vista anteriormente con los nombres de las funciones a utilizar en caso de que haya que atender peticiones para este dispositivo.

```
PUBLIC void bios_winchester_task()
        {
        driver_task(&w_dtab);
        }
```

4.2.- Función w_prepare:

Inicializa el dispositivo indicado para permitir E/S.

```
PRIVATE struct device *w_prepare(device)
        int device;    /* dispositivo que debe ser inicializado */
        {
        /* Al inicializan aún no hay nada que transferir, con lo que el contador de bytes a transferir
        debe ser puesto a 0  */
        w_count = 0;

        if (device < NR_DEVICES)
                {    /* hd0, hd1, ... */
       /* si el dispositivo a iniciar es válido, se debe obtener el número de unidad, sus
       características y la partición a la que está asociado */
       w_drive = device / DEV_PER_DRIVE;   /*almacena número de drive*/
       w_wn = &wini[w_drive];
       w_dv = &w_wn->part[device % DEV_PER_DRIVE];
       }
    else
       if ((unsigned) (device -= MINOR_hd1a) < NR_SUBDEVS)
                {    /* hd1a, hd1b, ... */
                /* el dispositivo puede pertenecer a los de tipo "menor" */
                /* con lo cual debe obtenerse el número de unidad "menor", sus características y la
```

```
                subpartición a la que está asociado */
                w_drive = device / SUB_PER_DRIVE;
                w_wn = &wini[w_drive];
                w_dv = &w_wn->subpart[device % SUB_PER_DRIVE];
                }
        else /* el dispositivo no es válido y se devuelve un error */
                { return (NIL_DEV); }
    /* si el número de unidad esta en el rango de unidades totales se devuelve las características
de la partición a la que esté asociado, en caso contrario se devuelve un error */
return (w_drive < nr_drives ? w_dv : NIL_DEV);
}
```

4.3.- Función w_name:

Retorna un nombre para la unidad actual.

```
PRIVATE char *w_name()
        {
        static char name[] = "bios-hd5";
        name[7] = '0' + w_drive * DEV_PER_DRIVE;
        return name;
        }
```

4.4.- Función w_schedule:

Agrupa las peticiones de E/S de bloques consecutivos, ya que éstas serán escritas o leídas en un comando emitido por el BIOS si se usa buffer. Chequea y agrupa todas las peticiones e intenta terminarlas lo antes posible en caso de que no haya buffer.

```
PRIVATE int w_schedule(proc_nr, iop)
int proc_nr;                    /* Identificador del proceso que realiza la petición */
struct iorequest_s *iop;        /* Puntero a la estructura de donde se va a leer o a escribir */
{
        int r, opcode;
        unsigned long pos;
        unsigned nbytes, count, dma_count;
        unsigned long block;
        phys_bytes user_phys, dma_phys;

        /* Cantidad de bytes que se desean leer o escribir */
        nbytes = iop->io_nbytes;
        if ((nbytes & SECTOR_MASK) != 0) return (iop->io_nbytes = EINVAL);

        /* Desde o hacia esta posición sobre el dispositivo */
        pos = iop->io_position;
        if ((pos & SECTOR_MASK) != 0) return (iop->io_nbytes = EINVAL);
```

```
/* Hacia o desde esta dirección de usuario */
/* numap nos devuelve la dirección de proceso de usuario que realizó la petición */
user_phys = numap (proc_nr, (vir_bytes) iop->io_buf, nbytes);
if (user_phys == 0) return(iop->io_nbytes = EINVAL);

/* Operación deseada: lectura o escritura */
opcode = iop->io_request & ~OPTIONAL_IO;

/* Que bloque del disco y como cerrar */
/* si la posición a leer o escribir es mayor que el tamaño del dispositivo se debe retornar
sin hacer nada */
if (pos >= w_dv->dv_size) return(OK);        /* Final de Dispositivo (EOF) */

/* si lo que tenemos que transferir (pos+nbytes) es mayor que el tamaño del dispositivo,
simplemente acotamos la cantidad de datos a transferir al tamaño de dispositivo que nos
queda*/
if (pos + nbytes > w_dv->dv_size) nbytes = w_dv->dv_size - pos;

/* determina el número de bloque que se quiere transferir */
block = (w_dv->dv_base + pos) >> SECTOR_SHIFT;

/* si usamos buffer de DMA, (USE_BUF), el número de bloques a transferir es mayor que
cero, pero el bloque seleccionado no es consecutivo al último que se transfirió, con lo cual
comienza el proceso de terminación de atención a una petición, w_finish */
if  (USE_BUF && w_count > 0 && block != w_nextblock)
    {
    if ((r = w_finish()) != OK) return (r);
    }

/* Si el siguiente bloque a transferir es consecutivo y se usa buffer de DMA, entonces
calculamos cual es el bloque que esperamos transferir la próxima vez, sumandole al
bloque actual, el número de bloques que se vieron afectados en la actual petición */
if (USE_BUF) w_nextblock = block + (nbytes >> SECTOR_SHIFT);

/* Mientras queden  bytes a transferir de la petición hacemos... */
do
    { /* tomamos el número de bytes a transferir */
    count = nbytes;

    if (USE_BUF)
        { /* estamos usando buffer de DMA */
        if (w_count == DMA_BUF_SIZE)
            {/* el buffer de DMA está lleno, con lo que no se puede atender la petición
actual, puesto que no se puede transferir más del buffer permitido para el DMA */
            if ((r = w_finish()) != OK) return(r);
```

```
            }
        if (w_count + count > DMA_BUF_SIZE)
            /* hay buffer libre en DMA, pero lo que queda por transmitir es mayor de
            lo que este buffer puede contener, con lo que se acorta la cantidad de datos
            que se enviarán al buffer */
            count = DMA_BUF_SIZE - w_count;
        }
    else
        { /* no estamos usando buffer de DMA */
        if (w_tp == wtrans + NR_IOREQS)
            {
            /* no hay espacio en el vector de transferencia "wtrans" para encolar la
            nueva petición con lo que debe esperar */
            if ((r = w_finish()) != OK) return(r);
            }
    }
    if (w_count == 0)
        { /* en la primera petición aún no hay nada encolado para transferir, pero debe
inicializarse los parámetros de Cód. de operación y de Posición dentro del vector de
transferencia */
        w_opcode = opcode;
        w_tp = wtrans;
        }
    if (USE_BUF)
        {
        /* Si el tamaño del buffer del DMA es mayor que el buffer a transferir */
        dma_phys = tmp_phys + w_count;
        }
    else
        {
        /* Se envía parte de lo que se va a transferir al DMA */
        dma_phys = user_phys;
        dma_count = dma_bytes_left(dma_phys);
        if (dma_phys >= (1L << 20))
            {
            /* La BIOS sólo puede direccionar el primer megabyte. */
            count = SECTOR_SIZE;
            dma_phys = tmp_phys;
            }
        else
            if (dma_count < count)
                {
                        /* Acercando 64K a un límite */
                if (dma_count >= SECTOR_SIZE)
                    {
```

```
                        /* Puede leer algunos sectores antes de llegar al límite */
                        count = dma_count & ~SECTOR_MASK;
                        }
                    else
                        {
                        /* Debe usar el buffer especial para esto */
                        count = SECTOR_SIZE;
                        dma_phys = tmp_phys;
                        }
                    }
                }
            /* Almacena parámetros de E/S */
            w_tp->iop = iop;
            w_tp->block = block;
            w_tp->count = count;
            w_tp->phys = user_phys;  /* Dirección de almacenamiento del área de usuario */
            w_tp->dma = dma_phys;   /* Dirección física del DMA */
            /* Actualizar los contadores  */
            w_tp++;
            w_count += count;
            block += count >> SECTOR_SHIFT;
            user_phys += count;
            nbytes -= count;
            }
        while (nbytes > 0);
        return (OK);
        }
```

4.5.- Función w_finish:

Realiza las peticiones de E/S recogidas en el vector "wtrans" que fueron generadas por w_schedule.

```
PRIVATE int w_finish()
{
        struct trans *tp = wtrans, *tp2;
        unsigned count, cylinder, sector, head, hicyl, locyl;
        unsigned secspcyl = w_wn->heads * w_wn->sectors;
        int many = USE_BUF;

        /* si no hay nada que transferir retornamos */
        if (w_count == 0) return (OK);

        do
            {
            if (w_opcode == DEV_WRITE)
```

```
        { /* se va a realizar una escritura */
                        tp2 = tp; /* tomamos la dirección de wtrans */
        count = 0;
        do
            { /* envía todo lo que haya que escribir */
            if (USE_BUF || tp2->dma == tmp_phys)
                {
                phys_copy(tp2->phys, tp2->dma, (phys_bytes) tp2->count);
                }
            count += tp2->count;
            tp2++;
            }
        while (many && count < w_count);
        }
    else
        {
        count = many ? w_count : tp->count;
        }

    /* realiza la transferencia */
    cylinder = tp->block / secspcyl; /* el cilindro afectado en la transferencia */
    sector = (tp->block % w_wn->sectors) + 1; /* el sector afectado en la transferencia
    head = (tp->block % secspcyl) / w_wn->sectors; /* la cabeza afectada en la
transferencia */
    /* si la operación es una escritura en Ax almacena en código apropiado, y si es una
lecturahacer lo mismo, inicializa los registros */
    Ax = w_opcode == DEV_WRITE ? BIOS_WRITE : BIOS_READ;
    Ax = (Ax << 8) | (count >> SECTOR_SHIFT);    /* cód. de operación y bytes
                            a transferi */
    Bx = (unsigned) tp->dma % HCLICK_SIZE;       /* 4 bits menos significativos */
    Es = physb_to_hclick(tp->dma);          /* 16 bits más significativos */
    hicyl = (cylinder & 0x300) >> 8;   /* 2 bits más significativos */
    locyl = (cylinder & 0xFF);         /* 8 bits menos significativos */
    Cx = sector | (hicyl << 6) | (locyl << 8);
    Dx = (HD_CODE + w_drive) | (head << 8);
    level0 (bios13); /* llama a la interrupción 13 del BIOS */
    if ((Ax & 0xFF00) != 0)
        {
        /* ha ocurrido un error, pero aún así debemos volver a intentar la transferencia
        if (!many) return(tp->iop->io_nbytes = EIO);
        many = 0;
        continue;
        }

    w_count -= count;
```

```
        do
            { /* se va a realizar una lectura */
            if (w_opcode == DEV_READ)
                {
                if (USE_BUF || tp->dma == tmp_phys)
                    {
                    phys_copy(tp->dma, tp->phys, (phys_bytes) tp->count);
                    }
                }
            tp->iop->io_nbytes -= tp->count;
            count -= tp->count;
            tp++;
            }
        while (count > 0);
        }
    while (w_count > 0);

    return (OK);
    }
```

4.6.- Función w_do_open:

Dispositivo abierto: Inicializa el controlador y lee la tabla de particiones.

```
PRIVATE int w_do_open(dp, m_ptr)
struct driver *dp; /* driver del dispositivo */
message *m_ptr; /* mensaje a transferir */
{
    static int init_done = FALSE;

    if (!init_done)
        { /* el dispositivo no ha sido inicializado, con lo cual se llama a w_init */
        w_init();
        init_done = TRUE;
        }
    /* comprobamos que el dispositivo es válido */
    if (w_prepare(m_ptr->DEVICE) == NIL_DEV) return (ENXIO);

    /* incrementamos el contador de uso sobre el dispositivo y leemos la tabla de particiones
    */
    if (w_wn->open_ct++ == 0)
        {
        /* Partición del disco. */
        partition(&w_dtab, w_drive * DEV_PER_DRIVE, P_PRIMARY);
        }
```

```
return (OK);
}
```

4.7.- Función w_do_close:
Cierra el dispositivo.

```
PRIVATE int w_do_close(dp, m_ptr)
struct driver *dp;
message *m_ptr;
{
  if (w_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
  w_wn->open_ct--;
  return(OK);
}
```

4.8.- Función w_init:
Esta rutina es llamada para empezar a inicializar los parámetros del driver.

```
PRIVATE void w_init()
{
int drive;
struct wini *wn;

/* Habilitar el modo real de los vectores de la BIOS. */
enable_vectors();

/* Establece la geometria de las unidades */
for (drive = 0; drive < nr_drives; drive++)
    {
    (void) w_prepare(drive * DEV_PER_DRIVE);
    wn = w_wn;
    Dx = drive + HD_CODE;
    Ax = (BIOS_ASK << 8);
    level0 (bios13);
    nr_drives = (Ax & 0xFF00) == 0 ? (Dx & 0xFF) : drive;
    if (nr_drives > MAX_DRIVES) nr_drives = MAX_DRIVES;
    if (drive >= nr_drives) break; /* si el número de unidad no es válido, puesto que es
superior al máximo permitido */
    wn->heads = (Dx >> 8) + 1;
    wn->sectors = (Cx & 0x3F);
    wn->cylinders = ((Cx << 2) & 0x0300) + ((Cx >> 8) & 0x00FF) + 1;
    wn->part[0].dv_size = ((unsigned long) wn->cylinders
            * wn->heads * wn->sectors) << SECTOR_SHIFT;
    /* muestra información referente a la configuracion de cada unidad */
    printf("%s: %d cylinders, %d heads, %d sectors per track\n",
```

```
            w_name(), wn->cylinders, wn->heads, wn->sectors);
        }
    }
```

4.9.- Función enable_vectors:

El modo protegido de Minix se ha reprogramado para poder utilizar diferentes vectores de interrupciones del bios, por lo que ello hace que los vectores de la bios tengan que ser copiados a un área de Minix para poder ser utilizados en modo protegido. Con este funcionamiento en modo protegido hace posible que se recuperen los posibles ticks perdidos mientras el disco esté activo al hacer una llamada a Int 13.

```
PRIVATE void enable_vectors()
    {
    int vec, irq;
    static u8_t clock_handler[] =
        {
        0x50, 0xB0, 0x20, 0xE6, 0x20, 0xEB, 0x06, 0xE4,
        0x61, 0x0C, 0x80, 0xE6, 0x61, 0x58, 0x53, 0x1E,
        0xE8, 0x00, 0x00, 0x5B, 0x2E, 0xC5, 0x5F, 0x0A,
        0xFF, 0x07, 0x1F, 0x5B, 0xCF,
        0x00, 0x00, 0x00, 0x00,
        };
    /* Manejador de reloj */
    u16_t vector[2];
    /* si no estamos en modo protegido debemos salir, porque los vectores de la BIOS sólo se
    pueden habilitar en modo
    protegido */
    if (!protected_mode) return;

    for (irq = 0; irq < NR_IRQ_VECTORS; irq++)
        {
        phys_copy(BIOS_VECTOR(irq)*4L, VECTOR(irq)*4L, 4L);
        }
    /* Instala un manejador de la interrupción de reloj para acumular los ciclos de reloj
    cuando el driver de disco de la BIOS está activo. Se usa para ello un simple bit del codigo
    del 8086 que incrementa "lost ticks" */
    vector[0] = vir2phys(clock_handler) % HCLICK_SIZE;
    vector[1] = vir2phys(clock_handler) / HCLICK_SIZE;
    phys_copy(vir2phys(vector), VECTOR(CLOCK_IRQ)*4L, 4L);

    /* Almacena la dirección de lost_ticks del manejador */
    vector[0] = vir2phys(&lost_ticks) % HCLICK_SIZE;
    vector[1] = vir2phys(&lost_ticks) / HCLICK_SIZE;
    memcpy(clock_handler + 0x1D, vector, 4);
```

```
if (ps_mca) clock_handler[6]= 0;   /* (PS/2 port B clock ack) */
}
```

4.10.- Función w_geometry:

Da las especificaciones del disco cilindros, cabezas y cilindros.

```
PRIVATE void w_geometry(entry)
struct partition *entry;
        {
        entry->cylinders = w_wn->cylinders;
        entry->heads = w_wn->heads;
        entry->sectors = w_wn->sectors;
        }
```

5.- Preguntas

A) Por qué se considera útil el controlador bios_win.c si su funcionamiento es muy lento y además su utilización no es recomendada?

Porque en casos de última necesidad, cuando el resto de controladores fallan, se podría seguir funcionando con éste, aunque el rendimiento fuese malo.

Además, a los discos de arranque no les queda más remedio que utilizar este controlador, ya que al iniciar el ordenador lo que tienen disponible es la bios.

A) Cómo trata las múltiples peticiones de lectura y/o escritura?

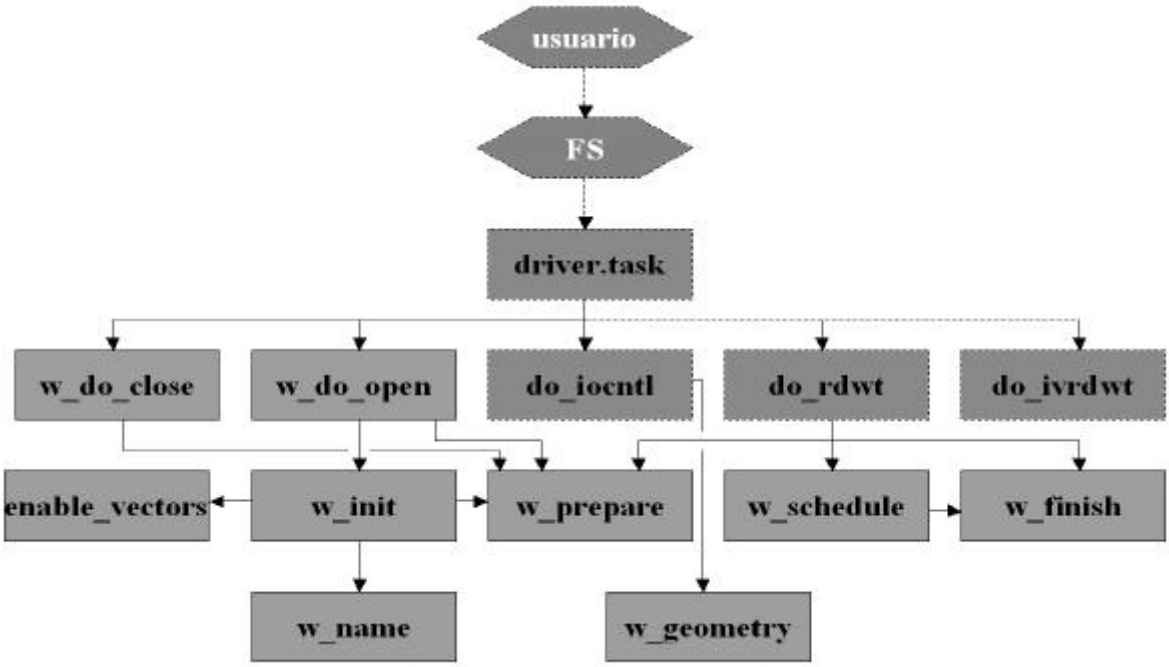
Se planifican en un vector de peticiones (wtrans), de forma que si disponemos de buffer se intentarán enviar varias peticiones de una sola vez a la bios. Ello sólo es posible si dichas peticiones acceden a zonas contiguas.

A) Para qué y por qué se utiliza el procedimiento enable_vectors ?

Se utiliza para copiar los vectores de interrupciones del bios a una zona de minix, de forma que cuando se produzca una interrupción, ésta se pueda tratar en modo protegido, habilitando las interrupciones que el minix no utilice y enmascarando las que se vean afectadas por la llamada a bios13().

Anexo1:

Relacion entre las distintas funciones del controlador de disco, tanto del nivel inferior como del nivel superior.


usuario
FS
driver.task
w_do_close
w_do_open
do_iocntl
do_rdwt
do_ivrdwt
enable_vectors
w_init
w_prepare
w_schedule
w_finish
w_name
w_geometry