AMPLIACIÓN DE SISTEMAS OPERATIVOS

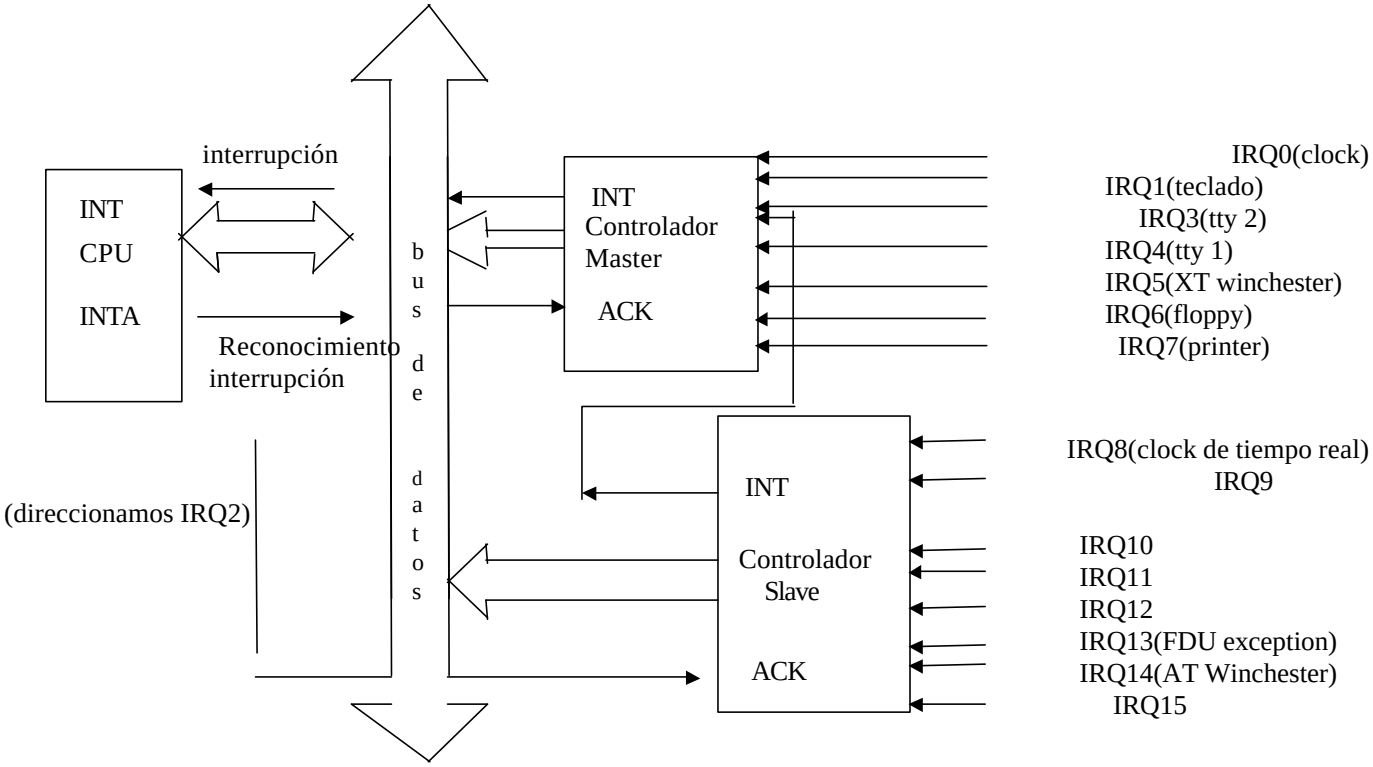
I8259.C
PROTECT.C
EXCEPTION.C

Javier Falcón Quintana
Manuel Robaina Mederos

M a n e j o d e i n t e r r u p c i o n e s e n M i n i x .

Las interrupciones generadas por los dispositivos hardware son señales el
primer lugar por un controlador de interrupciones (8259), un circuito integrado que puede captar un conjunto de tales señales, y por cada una genera un único patrón en el bus de datos. Esto es necesario porque el procesador solo tiene una entrada para recibir todos los dispositivos y por lo tanto no puede diferenciar que dispositivo necesita el servicio.

Los procesadores Intel de 32-bits son equipados con dos controladores, cada uno con 8 entradas, pero uno funciona como esclavo (Slave) el cual alimenta sus salidas a una de las entradas del controlador maestro, así distinguimos 15 dispositivos externos que pueden ser tratados como la combinación que vemos en la figura .



El pin INT de la CPU informa a esta que se ha producido una interrupción, a su vez la CPU da un reconocimiento de dicha interrupción y cuando esta llega al controlador citado, este introduce los datos al bus de datos especificando al procesador que rutina de servicio debe realizar. El controlador de interrupciones es programado durante la inicialización cuando el main llama int_init.

El programa determina la salida enviada a la CPU para una señal recibida sobre cada una de las líneas de entrada, también otros parámetros de operaciones de contro -bit, usado para indexar la tabla de vectores de interrupciones de 256 elementos. La tabla del MINIX tiene 56 elementos. Solo de los 56 utilizamos 35 y el resto son reservados para los futuros procesadores para Minix.

*****Mecanismo realizado a grosso modo en una interrupción*****

El mecanismo de cambio de contexto del procesador al recibir una interrupción es complejo, cambiar el contador de programa para ejecutar otra función constituye sólo una parte del mismo. Cuando la CPU recibe una interrupción mientras ejecuta un proceso, crea una nueva pila para usarla durante el servicio de interrupción. La localización de esta pila la determina una entrada en el Segmento de Estado de Tarea *(TSS).* El efecto es que la nueva pila creada por una interrupción siempre comienza al final de la estructura *stackframe (trama de pila)* dentro de la tabla de procesos correspondiente al proceso interrumpido. La CPU introduce varios registros en la nueva pila, incluyendo aquellos necesarios para la restauración de la pila del proceso interrumpido, así como su contador de programa. Cuando se empieza a ejecutar el manejador de la interrupción, utiliza esta zona en la tabla de procesos como su pila y ay se habrá almacenado mas información necesaria para retornar al proceso interrumpido.

El manejador de la interrupción, mete en la pila el contenido de ciertos registros adicionales, y cuando se llena el stackframe conmuta a una pila que proporciona el *Kernel.*

La terminación de la rutina de atención, se logra conmutando desde la pila del *Kernel* a un stackframe en la tabla de procesos (pero no necesariamente la misma que fue creada por la última interrupción) y ejecutando *iretd* (retorno de interrupción).

Iretd recupera el estado que existía antes de una interrupción, restaurando el valor de los registros que fueron introducidos por el hardware en la pila, y conmutando a una pila que se estaba usando antes de introducirse la interrupción.

La CPU desactiva todas las interrupciones cuando recibe una interrupción. Esto garantiza que nada pueda producir un desbordamiento del stackframe en la entrada de la tabla de proceso. Esto se hace automáticamente, pero también existen instrucciones a nivel ensamblador para habilitar y deshabilitar interrupciones. El manejador de interrupciones reactiva las interrupciones después de conmutar a la pila del Kernel que se encuentra fuera de la tabla de procesos. Se deben desactivar de nuevo todas las interrupciones antes de volver a conmutar a la pila dentro de la tabla de procesos, ya que puede producirse otra interrupción y ser procesada.

El nivel de privilegio controla los diferentes tipos de respuesta a interrupciones recibidas mientras un proceso y el código del Kernel (incluida la rutina de atención de la interrupción) se está ejecutando.

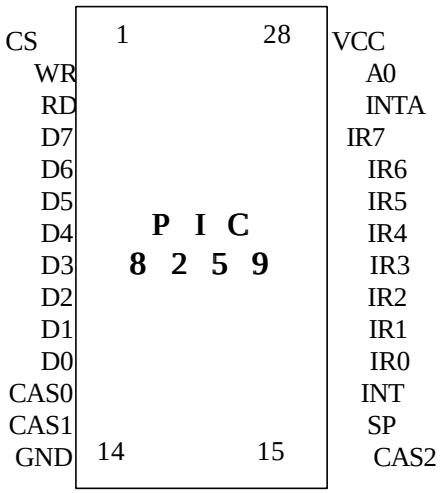
El controlador de interrupciones

Para aislar un poco más la CPU del hardware se intercaló un controlador de interrupciones entre está y las líneas de interrupciones de los diferentes dispositivos (8259A).

El controlador de interrupciones tiene la misión de recibir las solicitudes de interrupciones provenientes del hardware y transmitirlas a la CPU con lo que ésta activa el correspondiente administrador de interrupciones en la memoria RAM. No solo tiene que ver con las interrupciones del hardware sino con el teclado, disco duro, impresora, reloj estas quieren ser atendidas al mismo tiempo si bien la CPU sólo puede en cada instante atender

Las ocho interrupciones que puede atender un PIC (Programmable Interrupt Controller 8259A), están numeradas del 0 al 7. IRQ0 tiene la máxima prioridad y IRQ7 tiene la mínima.

Los terminales del PIC.



Si el procesador está preparando para asumir una interrupción de hardware y consiguientemente para llamar al correspondiente administrador de interrupciones, lo indica a través de un impulso por el terminal INTA del PIC. El controlador reacciona enviando el número de interrupción que debe ejecutarse, por medio de la línea D0 a D7.

La CPU llama a continuación al administrador de interrupciones de la correspondiente interrupción para lo cual lee su dirección en el vector de interrupciones. El número de interrupció
esa tabla.

El proceso se describe paso a paso del siguiente modo:

- Las líneas Irx están conectadas a cada una de las fuentes de interrupción.
- Cuando se va a dar paso a una interrupción se activa la INT, la cual está conectada a INTR en el procesador
- A través de INTA el procesador indica que está preparado y tras el segundo impulso en dicha pata puede recibir el número de interrupción.

- Por las líneas Dx se envía se envía al procesador el número de interrupción, éste sirve de índice para buscar la dirección de la rutina que lo trata en el vector de interrupciones.
- Por último pueden comentarse otros terminales. Mediante CS, WR y RD la CPU puede escribir y leer registros del PIC a través del bus de datos, el registro se indica e
  la comunicación del PIC maestro con el esclavo se hace mediante CAS0, CAS1 y CAS2.

Los registros internos y la estructura de prioridades

Para saber en cada instante qué interrupción se está ejecutando y cuáles están aún esperando, el PIC contiene tres registros internos con los nombres *Interrupt Request Register (IRR), In Service Register (ISR) y Interrupt Mask Register (IMR).*

IRR está directamente relacionado a las ocho líneas de interrupciones IR0-7; si una fuente de interrupción pone su línea IR activa, el correspondiente bit en IRR se pone a 1. De esta manera reconoce PIC que se ha

IMR por medio de este registro se pueden suprimir normalmente las interrupciones (bit a 1).

ISR es donde están anotadas las interrupciones que están ejecutándose. Cuando pasa a ISR se borra de IRR para no tratar de iniciarla otra vez.

Si se va a ejecutar una interrupción con menor prioridad, el PIC guarda la nueva interrupción hasta que se hayan atendido todas las interrupciones con mayor prioridad. También puede suceder que una interrupción con mayor prioridad suspenda la ejecución de otra interrupción si ésta tiene una prioridad menor, lo normal es que no se dé ninguna suspensión de una interrupción con una prioridad baja puesto que ante una llamada a una interrupción la CPU desactiva las interrupciones. El comando IRET activa las interrupciones cuando se finaliza

Disposición en cascada

Podemos poner en cascada varios manejadores de interrupciones, considerando a uno de ellos cono maestro y el resto como esclavo. Si un esclavo recibe una señal por una línea Irx para provocar una interrupción, primero comprueba, basándose en sus registros internos como ya se ha explicado, si se va a producir el requerimiento de la interrupción. Si es así, pone su línea INT activa recibiendo la señal el manejador maestro, al comprobar el maestro en la línea IR2 una solicitud de interrupción la trata exactamente igual que si viniera del hardware.

Cuando se envía la solicitud a la CPU, el maestro pone el número del esclavo que a realizado la solicitud en las líneas CASx que se conectan a las líneas del mismo nombre en el esclavo. Cuando recibe una señal en INTA lanzada por el maestro por INT, envía por el bus de datos la interr
hacen las modificaciones necesarias, y habituales, en los registros.

Los PIC saben si son maestro o esclavo, porque el maestro es el que recibe tensión por EN mientras el resto la tiene conectada a la masa de la placa.

Inicialización del manejador (PIC) vía software

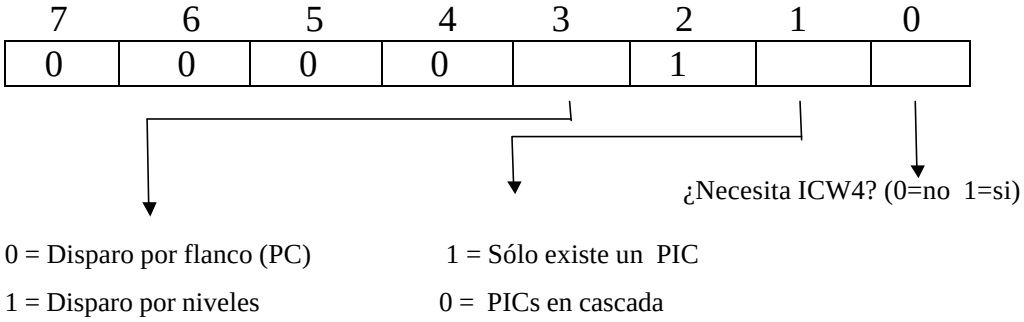
El PIC reconoce diversos comandos para su inicialización y control, divididos en comandos *ICW* y *OCW*.

ICW (Inicilization Command Words) Se usan para configurar el PIC y tienen que ser enviados al manejador de interrupciones en un orden concreto y estricto pues depende unos de otros.

OCW (Operational Command Words) no hay dependencia por lo que se puede utilizar en el orden que se prefiera.

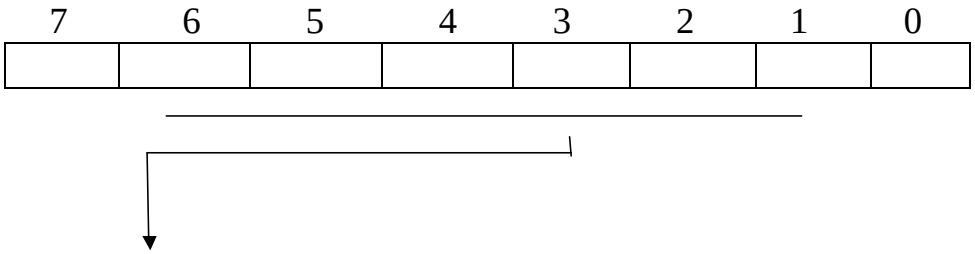
Las comunicaciones se establecen básicamente por dos puertos de direcciones diferentes. Para el maestro en los PIC, los puertos son el 20h y 21h y para el esclavo A0h y A1h. Normalmente la BIOS asume la inicialización del PIC y sólo en situaciones excepcionales se debe modificar la configuración establecida por la BIOS. La inicialización comienza siempre con el envío del ICW1 al puerto 20h o A0h, dependiendo si se inicializa un maestro o un esclavo.

ICW1



En la figura anterior tenemos la estructura del ICW1. Posteriormente debe seguir la ICW2, por el segundo puerto, a través de la cual se define la dirección base para la primera interrupción (IRQ0), la segunda (IRQ1) ejecuta esta interrupción +1, la (IRQ2) la interrupci

ICW2

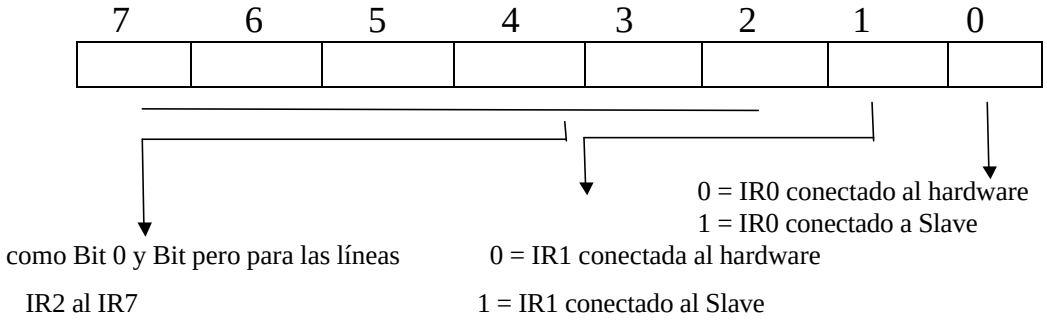


Número de interrupción que se dispara con la IRQ0

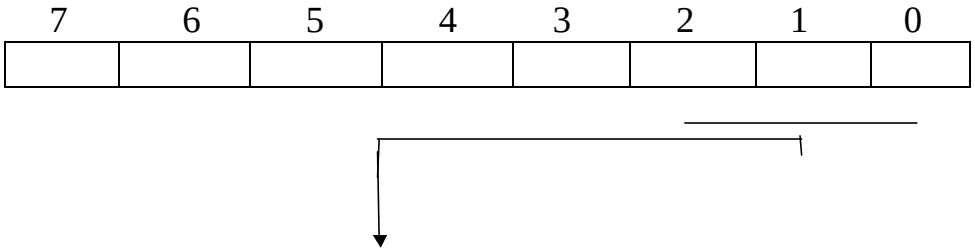
Para el maestro, el BIOS utiliza el valor 08h en ICW2 y para el esclavo, el valor 70h.

La ICW3 sólo debe enviarse si tenemos mas de un manejador de interrupciones, es decir si esta en cascada.

ICW3 para inicializar PIC como maestro

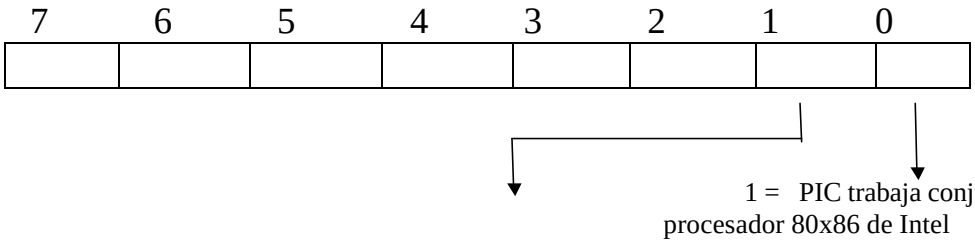


ICW3 para inicializar PIC como esclavo



Número de líneas de interrupciones a través de conectado al maestro.

Para el ICW4 se informa al PIC de si trabaja en un entorno Intel y de si debe registrar automáticamente la finalización de una interrupción o debe dejarse ayudar por el software para ello.



1 = PIC trabaja conjuntamente con un procesador 80x86 de Intel

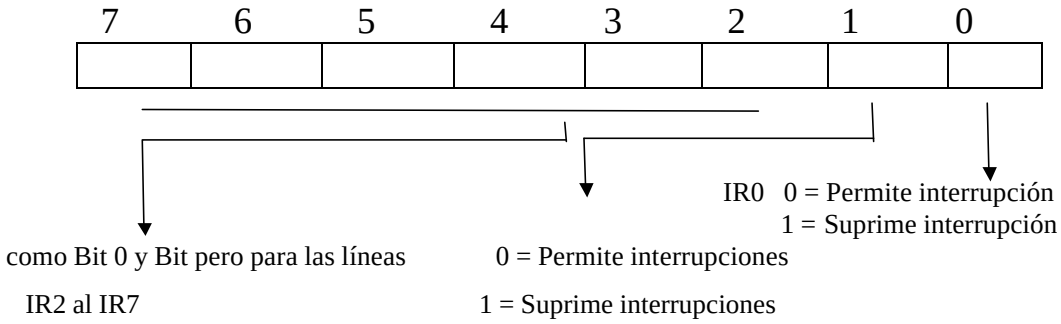
0 = Terminación manual de interrupción mediante EOI

1 = Terminación automática de interrupción

Control y consulta del PIC a través de los OCW

Mientras que los ICW sólo se envían una vez al PIC y de ello se encarga el BIOS, en los programas se pueden utilizar los OCW para configurar el PIC o para simplemente obtener informaciones de estado. Se distingue entre las distintas OCW mediante su formato y el puerto de entrada.

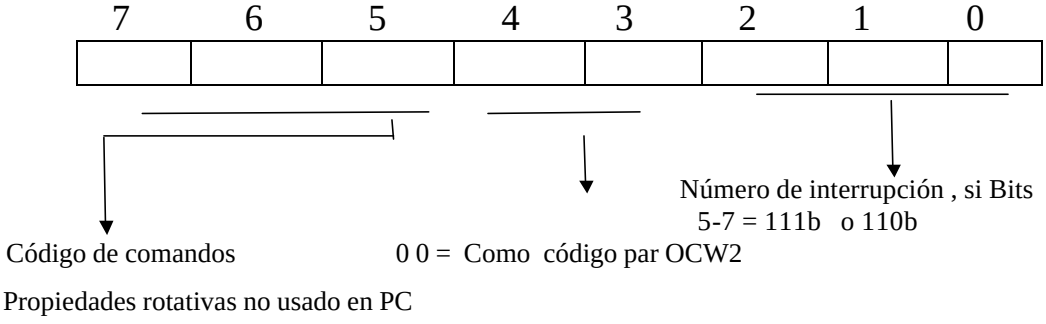
OCW1



La OCW1 permite el acceso directo al registro de enmascaramiento y ofrece la posibilidad de suprimir momentáneamente o volver a liberar de manera individual las interrupciones del hardware.

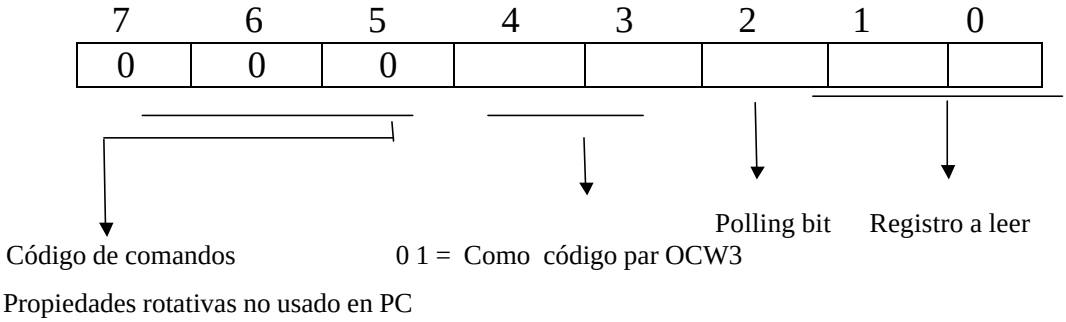
La OCW2 se utiliza sobre todo para indicar el final de una interrupción
serie de posibilidades de configuración referidas a las prioridades y el modo EOI automático.

OCW2



Por último tenemos OCW3 ofrece la posibilidad de poder leer desde un programa los registros internos IRR e ISR.

OCW3



La opción de polling es aquella en que el procesador no está directamente conectado con el PIC, sino que el sistema operativo debe encargarse de que el procesador consulte a intervalos regulares el estado de las interrupciones.

/* i 8 2 5 9 . c */

/* Este fichero contiene rutinas para inicializar el controlador de interrupciones.:
* **get_irq_handler**: Direcciona y maneja una interrupción dada, nos da la
* dirección de la interrupción a tratar en la tabla de interrupciones.
* **put_irq_handler**: Registra un manejador de interrupciones, dando la tarea a * realizar reemplazándola en la variable spurious-irq..
* **intr_init:** Inicializa el controlador de interrupciones y escribe datos a los
* puertos locales según sea 20h - 21h maestro y A0h -A1h esclavo.
*/

```
#include "kernel.h"

#define ICW1_AT         0x11    /* Disparo por flancos, cascada, necesita ICW4 */
#define ICW1_PC         0x13    /* Disparo por flancos , no cascada, necesita ICW4 */
#define ICW1_PS         0x19    /* Disparo por flancos, cascada, necesita ICW4 */
#define ICW4_AT         0x01    /* not SFNM, not buffered, normal EOI, Intel 8086 */
#define ICW4_PC         0x09    /* not SFNM, buffered, normal EOI, Intel 8086 */

FORWARD _PROTOTYPE( int spurious_irq, (int irq) );

#if _WORD_SIZE == 2
typedef _PROTOTYPE( void (*vecaddr_t), (void) );

FORWARD _PROTOTYPE( void set_vec, (int vec_nr, vecaddr_t addr) );

PRIVATE vecaddr_t int_vec[] = {
  int00, int01, int02, int03, int04, int05, int06, int07,
};

PRIVATE vecaddr_t irq_vec[] = {
  hwint00, hwint01, hwint02, hwint03, hwint04, hwint05, hwint06, hwint07,
  hwint08, hwint09, hwint10, hwint11, hwint12, hwint13, hwint14, hwint15,
};
#else
#define set_vec(nr, addr)       ((void)0)
#endif
```

*** i n t r _ i n i t ***

```
PUBLIC void intr_init(mine)
int mine;
{
```

/* Inicializa la 8259 , finalizando desactivando todas las interrupciones. Esto es solamente realizado en modo protegido,en modo real no tocamos el 8259, usamos la BIOS. El flag “mine” es activado si el 8259 es programado por Minix, o es reseteado por la BIOS. “mine” selecciona el modo a usar.
*/

```
  int i;
```

Modo protegido

```
  lock();
  if (protected_mode) {
        /* Los AT y los nuevos PS/2 tienen dos controladores de interrupciones , uno maestro, uno esclavo en  IRQ 2.  (No
tenemos que tratar con el PC que tiene un controlador ,porque este debe correr en modo real.) */

        out_byte(INT_CTL, ps_mca ? ICW1_PS : ICW1_AT);  /*ICW1*/
        out_byte(INT_CTLMASK, mine ? IRQ0_VECTOR : BIOS_IRQ0_VEC);      /* ICW2 Para maestro */
        out_byte(INT_CTLMASK, (1 << CASCADE_IRQ));      /* ICW3 Dice al esclavo */
        out_byte(INT_CTLMASK, ICW4_AT);                 /*ICW4*/
        out_byte(INT_CTLMASK, ~(1 << CASCADE_IRQ));   /* Deshabilita todas las irq menos la 2 OCW1 */
        out_byte(INT2_CTL, ps_mca ? ICW1_PS : ICW1_AT);
        out_byte(INT2_CTLMASK, mine ? IRQ8_VECTOR : BIOS_IRQ8_VEC);         /* ICW2 para esclavo */
        out_byte(INT2_CTLMASK, CASCADE_IRQ);            /* ICW3 es esclavo nr */
        out_byte(INT2_CTLMASK, ICW4_AT);
        out_byte(INT2_CTLMASK, ~0);                     /* OCW1 */
  } else {

        /* Usamos el vector de interrupciones de la BIOS en modo real. Solamente reprogramamos las excepciones, el
vector de interrupciones son reprogramadas  a solicitud. Solamente reprogramamos el  SYS_VECTOR es el sistema de
llamadas para el paso de mensaje en Minix .
         */
        for (i = 0; i < 8; i++) set_vec(i, int_vec[i]);
        set_vec(SYS_VECTOR, s_call);  /* Llamadas al vector de interrupciones */
  }

  /* Inicializamos la tabla de manejador de interrupciones. */
  for (i = 0; i < NR_IRQ_VECTORS; i++) irq_table[i] = spurious_irq;
}



*                                   spurious_irq                                    *

PRIVATE int spurious_irq(irq)
int irq;
{
/* Manejador de  interrupciones por defecto. */

  if (irq < 0 || irq >= NR_IRQ_VECTORS)         /*Interrupción no váilida*/
        panic("invalid call to spurious_irq", irq);

  printf("spurious irq %d\n", irq);

  return 1;         /* Reactivamos interrupciones */
}
```

```
*                    p u t _ i r q _ h a n d l e r                    *

PUBLIC void put_irq_handler(irq, handler)
int irq;
irq_handler_t handler;
{
/* Registra un manejador de interrupciones . */

  if (irq < 0 || irq >= NR_IRQ_VECTORS)          /*IRQ no válida*/
        panic("invalid call to put_irq_handler", irq);

  if (irq_table[irq] == handler)                 /*Ya estaba inicializado*/
        return;         /* extra initialization */

  if (irq_table[irq] != spurious_irq)            /*Se inicializa por segunda vez con un manejador distinto*/
        panic("attempt to register second irq handler for irq", irq);
  disable_irq(irq);
  if (!protected_mode)  set_vec(BIOS_VECTOR(irq), irq_vec[irq]);
  irq_table[irq]= handler;
  irq_use |= 1 << irq;    /*bitmap de todas las irq´s en uso, activa mediante un or bit a bit la irq actual*/
}


#if _WORD_SIZE == 2

*                    s e t _ v e c                    *

PRIVATE void set_vec(vec_nr, addr)
int vec_nr;                         /* Que  vector */
vecaddr_t addr;                 /* Dirección de comienzo */
{
/* se coloca en modo real  */

  u16_t vec[2];                     /*Se declara un vector de dos posiciones*/

  /*Construye un vector en el array 'vec'. */
  vec[0] = (u16_t) addr;            /*Le asigna una dirección de comienzo*/
  vec[1] = (u16_t) physb_to_hclick(code_base);  /*Dirección base del código del Kernel*/

  /* Copia el vector en el array creado */
  phys_copy(vir2phys(vec), vec_nr * 4L, 4L);  /*Copia un bloque de datos entre dos posiciones cualquiera de memoria*/
}
#endif /* _WORD_SIZE == 2 */
```

M o d o p r o t e g i d o

Vemos una breve explicación de como funciona el modo protegido del 80386 y y486. Trabajamos con registros de 32 bits de anchura.

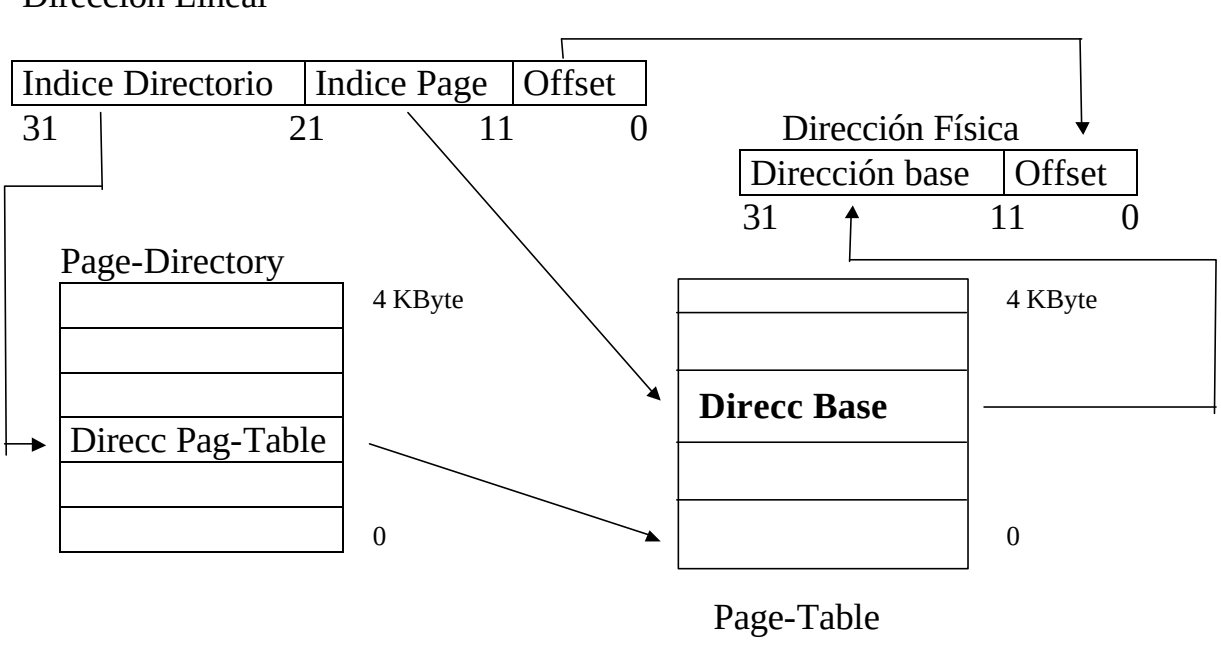
****Paginado y la gestión virtual de memoria ****

El 80386 y sus sucesores implementan un mecanismo de paginado con bloques de memoria de 4 Kbytes de tamaño. Mientras este mecanismo está desactivado, las direcciones lineales que se crean de la resolución de los sectores de segmento y las direcciones de offset representan, al igual que en el 80286, direcciones físicas a las que el procesador realmente accede.

Pero si se conecta el paginado, las direcciones lineales ya no son idénticas con las direcciones físicas, sino que se proyectan a las direcciones de memoria físicas mediante la llamada Page-Table.

La gestión de memoria virtual emplea bloque de memoria constante en nuestro caso 4 Kbyte, sólo es necesario cargar las páginas de cuatro Kbyte que realmente se necesitan, y no el segmento completo con su tamaño de hasta 1 Gbyte. Mediante la alineación a límites de 4 Kbyte, se puede tomar directamente los 12 bits inferiores de una dirección lineal a la dirección física. Así que representa una especie de offset a la página actual. Los 20 bits superiores de la dirección lineal indican por el contrario el número de página en la que se encuentra la posición de memoria direccionada. Todas las entradas en la Page-Table tiene una anchura de 32 bits . Sin embargo sólo se necesitan 20 bits para la dirección base , ya que ha de comenzar en una posición de memoria divisible entre 4 Kbyte, y los 12 bits bajos serán cero en cualquier caso, o lo utilizamos como banderas que puede indicar a la gestión virtual de la memoria si la página se encuentra actualmente en memoria.

Sin embargo, el asunto es algo mas confuso, ya que la Page-Table con sus 2^20 entradas de 4 Kbyte ocuparía cuatro Mbyte completos, esto significa regalar varios Mbytes de valiosa memoria para solventar esto creamos otra tabla llamada Page-Directory.



Las diferentes entradas representan a punteros a las direcciones de las diferentes Page-Table en la zona de direcciones física. El procesador obtiene sólo de ellas la dirección física de una página. El número de la entrada que se emplea en la Page-Table correspondiente para la conversión de la dirección resulta de los bits 12 ineal, de modo que esta se divide en dos trozos prácticamente iguales: un índice de 10 bits para el Page-Directory y otro índice de 10 bits para la Page-Table correspondiente.

*/ p r o t e c t . c */

Contiene rutinas relativas a la operación en modo protegido de los procesadores Intel. La Tabla de descriptores Globales (GDT), Tabla de descriptores locales (LTDs) y Tabla de descriptores de interrupciones(IDT) se encuentran en memoria, y su acceso está restringido, proprocionando acceso restringido a los recursos del sistema. La GDT e IDT son indexadas por registros especiales entro de la CPU. La GDT está disponible para todos los procesos y mantiene descriptores de segmento para regiones de memoria usadas por el sistema operativo. Hay normalmente una LDT por cada proceso, manteniendo el descriptores de segmento de las regiones de memoria usadas por los procesos. Los descriptores son estructuras de 8 bytes con un cierto número de componentes, pero los campos más importantes son los que describen, la dirección
región de memoria. La IDT contiene la dirección del código a ejecutar cuando la correspondiente interrupción es activada.

s r c / k e r n e l / p r o t e c t . c /* en que directorio de Minix lo encontramos */

```
07700  /* Este fichero contiene el código para inicializar el modo protegido ,para
07701   *iniciar los descriptores de segmento de código y datos,asi como inicializar
07702   *los descriptores globales , locales en la tabla de procesos.
07703   */

07704
07705  #include "kernel.h"
07706  #include "proc.h"
07707  #include "protect.h"
07708
07709  #define INT_GATE_TYPE   (INT_286_GATE | DESC_386_BIT)
07710  #define TSS_TYPE        (AVL_286_TSS  | DESC_386_BIT)
07711
07712  struct desctableptr_s {
07713    char limit[sizeof(u16_t)];
07714    char base[sizeof(u32_t)];           /* really u24_t + pad for 286 */
07715  };
07716
07717  struct gatedesc_s {
07718    u16_t offset_low;
07719    u16_t selector;
07720    u8_t pad;                   /* |000|XXXXX| ig & trpg, |XXXXXXXX| task g */
07721    u8_t p_dpl_type;            /* |P|DL|0|TYPE| */
07722    u16_t offset_high;
07723  };
07724
07725  struct tss_s {
07726    reg_t backlink;
07727    reg_t sp0;                  /* puntero de  pila para usar durante la interrupción */
07728    reg_t ss0;                  /*   "   segment  " "    "        "     */
07729    reg_t sp1;
07730    reg_t ss1;
07731    reg_t sp2;
07732    reg_t ss2;
07733    reg_t cr3;
```

```
07734     reg_t ip;
07735     reg_t flags;
07736     reg_t ax;
07737     reg_t cx;
07738     reg_t dx;
07739     reg_t bx;
07740     reg_t sp;
07741     reg_t bp;
07742     reg_t si;
07743     reg_t di;
07744     reg_t es;
07745     reg_t cs;
07746     reg_t ss;
07747     reg_t ds;
07748     reg_t fs;
07749     reg_t gs;
07750     reg_t ldt;
07751     u16_t trap;
07752     u16_t iobase;
07753   };
07754
07755   PUBLIC struct segdesc_s gdt[GDT_SIZE];
07756   PRIVATE struct gatedesc_s idt[IDT_SIZE];
07757   PUBLIC struct tss_s tss;
07758
07759   FORWARD _PROTOTYPE( void int_gate, (unsigned vec_nr, phys_bytes base,
07760                 unsigned dpl_type) );
07761   FORWARD _PROTOTYPE( void sdesc, (struct segdesc_s *segdp, phys_bytes base,
07762                 phys_bytes size) );
07763
07764
07765   *                    p r o t_i n i t                    *
```

Esta rutina la llama start.c para inicializar la GDT. La BIOS requiere que se ordene de una manera concreta.Los índices asociados se definen en *protect.h* .En la tabla de procesos se distribuye espacio para una LDT por cada proceso. Cada LDT contiene dos descriptores, para los segmentos de código y datos,.

```
07766
07767   PUBLIC void prot_init()
07768   {
07769   /* Crea  tablas necesarias para el modo protegido.
07770    * Todas las entradas de la GDT  son  almacenadas en tiempo de compilación.
07771    */
07772
07773     phys_bytes code_bytes;
07774     phys_bytes data_bytes;
07775     struct gate_table_s *gtp;
07776     struct desctableptr_s *dtp;
07777     unsigned ldt_selector;
07778     register struct proc *rp;
07779
07780     static struct gate_table_s {
07781           _PROTOTYPE( void (*gate), (void) );
07782           unsigned char vec_nr;
07783           unsigned char privilege;
07784     }
07785     gate_table[] = {
07786           divide_error, DIVIDE_VECTOR, INTR_PRIVILEGE,
07787           single_step_exception, DEBUG_VECTOR, INTR_PRIVILEGE,
```

```
07788         nmi, NMI_VECTOR, INTR_PRIVILEGE,
07789         breakpoint_exception, BREAKPOINT_VECTOR, USER_PRIVILEGE,
07790         overflow, OVERFLOW_VECTOR, USER_PRIVILEGE,
07791         bounds_check, BOUNDS_VECTOR, INTR_PRIVILEGE,
07792         inval_opcode, INVAL_OP_VECTOR, INTR_PRIVILEGE,
07793         copr_not_available, COPROC_NOT_VECTOR, INTR_PRIVILEGE,
07794         double_fault, DOUBLE_FAULT_VECTOR, INTR_PRIVILEGE,
07795         copr_seg_overrun, COPROC_SEG_VECTOR, INTR_PRIVILEGE,
07796         inval_tss, INVAL_TSS_VECTOR, INTR_PRIVILEGE,
07797         segment_not_present, SEG_NOT_VECTOR, INTR_PRIVILEGE,
07798         stack_exception, STACK_FAULT_VECTOR, INTR_PRIVILEGE,
07799         general_protection, PROTECTION_VECTOR, INTR_PRIVILEGE,
07800         page_fault, PAGE_FAULT_VECTOR, INTR_PRIVILEGE,
07801         copr_error, COPROC_ERR_VECTOR, INTR_PRIVILEGE,
07802         { hwint00, VECTOR( 0), INTR_PRIVILEGE },
07803         { hwint01, VECTOR( 1), INTR_PRIVILEGE },
07804         { hwint02, VECTOR( 2), INTR_PRIVILEGE },
07805         { hwint03, VECTOR( 3), INTR_PRIVILEGE },
07806         { hwint04, VECTOR( 4), INTR_PRIVILEGE },
07807         { hwint05, VECTOR( 5), INTR_PRIVILEGE },
07808         { hwint06, VECTOR( 6), INTR_PRIVILEGE },
07809         { hwint07, VECTOR( 7), INTR_PRIVILEGE },
07810         { hwint08, VECTOR( 8), INTR_PRIVILEGE },
07811         { hwint09, VECTOR( 9), INTR_PRIVILEGE },
07812         { hwint10, VECTOR(10), INTR_PRIVILEGE },
07813         { hwint11, VECTOR(11), INTR_PRIVILEGE },
07814         { hwint12, VECTOR(12), INTR_PRIVILEGE },
07815         { hwint13, VECTOR(13), INTR_PRIVILEGE },
07816         { hwint14, VECTOR(14), INTR_PRIVILEGE },
07817         { hwint15, VECTOR(15), INTR_PRIVILEGE },
07818 };
07819
07820 /* Se llama al comienzo, en este punto no puede utilizar todavía las tablas  creadas por main(). */
07821 data_bytes = (phys_bytes) sizes[1] << CLICK_SHIFT;
07822 if (sizes[0] == 0)
07823         code_bytes = data_bytes;        /* comun  I&D */
07824 else
07825         code_bytes = (phys_bytes) sizes[0] << CLICK_SHIFT;
07826
07827 /* Construye punteros a GDT e IDT en GDT, donde la  BIOS espera que se encuentren. */
07828 dtp= (struct desctableptr_s *) &gdt[GDT_INDEX];
07829 * (u16_t *) dtp->limit = (sizeof gdt) - 1;
07830 * (u32_t *) dtp->base = vir2phys(gdt);
07831
07832 dtp= (struct desctableptr_s *) &gdt[IDT_INDEX];
07833 * (u16_t *) dtp->limit = (sizeof idt) - 1;
07834 * (u32_t *) dtp->base = vir2phys(idt);
07835
07836 /* Construye descriptores de segmento para manejador de tareas e  interrupciones. */
07837 init_codeseg(&gdt[CS_INDEX], code_base, code_bytes, INTR_PRIVILEGE);
07838 init_dataseg(&gdt[DS_INDEX], data_base, data_bytes, INTR_PRIVILEGE);
07839 init_dataseg(&gdt[ES_INDEX], 0L, 0L, TASK_PRIVILEGE);
07840
07841 /* Construye resto de descriptores para funciones de klib88. */
07842 init_dataseg(&gdt[DS_286_INDEX], (phys_bytes) 0,
07843               (phys_bytes) MAX_286_SEG_SIZE, TASK_PRIVILEGE);
07844 init_dataseg(&gdt[ES_286_INDEX], (phys_bytes) 0,
```

```
07845                 (phys_bytes) MAX_286_SEG_SIZE, TASK_PRIVILEGE);
07846
07847     /* Construye descriptores locales en GDT para cada LDT en la tabla de procesos.
07848      * Los LDT's  son asignados en tiempo de compilación en la tabla de procesos
7849       * e inicializados siempre que el mapa de un proceso se inicializa o cambia */
7850
07851     for (rp = BEG_PROC_ADDR, ldt_selector = FIRST_LDT_INDEX * DESC_SIZE;
07852         rp < END_PROC_ADDR; ++rp, ldt_selector += DESC_SIZE) {
07853         init_dataseg(&gdt[ldt_selector / DESC_SIZE], vir2phys(rp->p_ldt),
07854                 (phys_bytes) sizeof rp->p_ldt, INTR_PRIVILEGE);
07855         gdt[ldt_selector / DESC_SIZE].access = PRESENT | LDT;
07856         rp->p_ldt_sel = ldt_selector;
07857     }
07858
07859     /* Construcción del segmento de estado de tareas (TSS).
7860       * De hecho sólo se accede al campo de puntero de pila, para almacenar el valor que tiene
07861        * y usarlo después después de una interrupción
07862       * Se crea el puntero para que una interrupción automáticamente grabe
07863       * los registros de procesos concurrentes ip:cs:f:sp:ss en la correspondiente
07864       * pista de la tabla de proceso.
07865       */
07866     tss.ss0 = DS_SELECTOR;
        /*
7867     init_dataseg(&gdt[TSS_INDEX], vir2phys(&tss), (phys_bytes) sizeof tss,
07868                                             INTR_PRIVILEGE);
07869     gdt[TSS_INDEX].access = PRESENT | (INTR_PRIVILEGE << DPL_SHIFT) | TSS_TYPE;
07870     tss.iobase = sizeof tss;       /* mapa de permisos  i/o vacias */
07871
07872     /* Construye descriptores  para puertas de interrupciones de entrada en IDT. */
07873     for (gtp = &gate_table[0];
07874         gtp < &gate_table[sizeof gate_table / sizeof gate_table[0]]; ++gtp) {
07875         int_gate(gtp->vec_nr, (phys_bytes) (vir_bytes) gtp->gate,
07876                 PRESENT | INT_GATE_TYPE | (gtp->privilege << DPL_SHIFT));
07877     }
07878     int_gate(SYS_VECTOR, (phys_bytes) (vir_bytes) p_s_call,
07879             PRESENT | (USER_PRIVILEGE << DPL_SHIFT) | INT_GATE_TYPE);
07880     int_gate(LEVEL0_VECTOR, (phys_bytes) (vir_bytes) level0_call,
07881             PRESENT | (TASK_PRIVILEGE << DPL_SHIFT) | INT_GATE_TYPE);
07882     int_gate(SYS386_VECTOR, (phys_bytes) (vir_bytes) s_call,
07883             PRESENT | (USER_PRIVILEGE << DPL_SHIFT) | INT_GATE_TYPE);
07884 }

07886
07887  *                    i n i t _ c o d e s e g                    *
07888 /* Construye los descritores  donde encontramos el segmento de código */
07889  PUBLIC void init_codeseg(segdp, base, size, privilege)
07890  register struct segdesc_s *segdp;
07891  phys_bytes base;
07892  phys_bytes size;
07893  int privilege;
07894  {
07895
07896
07897     sdesc(segdp, base, size);
07898     segdp->access = (privilege << DPL_SHIFT)
07899                 | (PRESENT | SEGMENT | EXECUTABLE | READABLE);
07900                 /* CONFORMING = 0, ACCESSED = 0 */
```

```
07901  }
07903
07904   *                    i n i t _ d a t a s e g                    *
07905
07906  PUBLIC void init_dataseg(segdp, base, size, privilege)
07907  register struct segdesc_s *segdp;
07908  phys_bytes base;
07909  phys_bytes size;
07910  int privilege;
07911  {
07912  /*Construye el descriptor para un segmento de dato. */
07913
07914    sdesc(segdp, base, size);
07915    segdp->access = (privilege << DPL_SHIFT) | (PRESENT | SEGMENT | WRITEABLE);
07916                  /* EXECUTABLE = 0, EXPAND_DOWN = 0, ACCESSED = 0 */
07917  }
07919
07920   *                    s d e s c                    *
7 9 2 1         /*Completa el trabajo de init_dataseg e init-codeseg. El modo de actuación de ambas es similar, se
        convierten      los parametros pasados en descriptores de segmento*/
07922  PRIVATE void sdesc(segdp, base, size)
07923  register struct segdesc_s *segdp;
07924  phys_bytes base;
07925  phys_bytes size;
07926  {
07927  /* Rellena los campos de tamaño(base, límite y granularidad) de un descriptor. */
07928
07929    segdp->base_low = base;
07930    segdp->base_middle = base >> BASE_MIDDLE_SHIFT;
07931    segdp->base_high = base >> BASE_HIGH_SHIFT;
07932    --size;                     /* lo convierte a un limite, 0 tamaño significa 4G */
07933    if (size > BYTE_GRAN_MAX) {
07934          segdp->limit_low = size >> PAGE_GRAN_SHIFT;
07935          segdp->granularity = GRANULAR | (size >>
07936                                    (PAGE_GRAN_SHIFT + GRANULARITY_SHIFT));
07937    } else {
07938          segdp->limit_low = size;
07939          segdp->granularity = size >> GRANULARITY_SHIFT;
07940    }
07941    segdp->granularity |= DEFAULT;        /* significa BIG para un segmento de datos */
07942  }

        /* Realiza una operación inversa a sdesc, extrayendo la dirección base de un descriptor de segmento */
07944
07945   *                    s e g 2 p h y s                    *
07946
07947  PUBLIC phys_bytes seg2phys(seg)
07948  U16_t seg;
07949  {
07950  /* Retorna la dirección base de un segmento, pudiendo ser seg o bien un registro de
07951   * segmento 8086 o bien un selector de segmento 286/386.
```

```
07952   */
07953    phys_bytes base;
07954    struct segdesc_s *segdp;
07955
07956    if (!protected_mode) {
07957          base = hclick_to_physb(seg);
07958    } else {
07959          segdp = &gdt[seg >> 3];
07960          base = segdp->base_low | ((u32_t) segdp->base_middle << 16);
07961          base |= ((u32_t) segdp->base_high << 24);
07962    }
07963    return base;
07964  }
07965
07966
07967   *                    i n t _ g a t e                    *
07968
07969  PRIVATE void int_gate(vec_nr, base, dpl_type)
07970  unsigned vec_nr;
07971  phys_bytes base;
07972  unsigned dpl_type;
07973  {
07974  /* Se construye un descriptor para puerta de interrupción. */
07975
07976    register struct gatedesc_s *idp;
07977
07978    idp = &idt[vec_nr];
07979    idp->offset_low = base;
07980    idp->selector = CS_SELECTOR;
07981    idp->p_dpl_type = dpl_type;
07982    idp->offset_high = base >> OFFSET_HIGH_SHIFT;
07983  }

07985
07986   *                        enable_iop                        *
07987
07988  PUBLIC void enable_iop(pp)
07989  struct proc *pp;
07990  {
07991  /* Permite a un proceso de usuario usar instrucciones de E/S.Cambia los bits de
07992    * nivel de permiso(CPL) de E/S en el campo psw. Estos especifican un permiso
          * actual con el menor privilegio permitido para ejecutar instrucciones de E/S.
07993    * Ususarios y servidores tienen un CPL 3.
07994    * Este es el menor privilegio posible.El Kernel tiene CPL 0,
07995    * las tareas tienen CPL 1. */
07996    pp->p_reg.psw |= 0x3000;
07997  }
```

*** p r o t e c t . h */**

```
/* Constantes para modo protegido. */

/* Tamaño de Tablas. */
#define GDT_SIZE (FIRST_LDT_INDEX + NR_TASKS + NR_PROCS) /* spec. and LDT's */
#define IDT_SIZE (IRQ8_VECTOR + 8)          /* Vector mas alto */
#define LDT_SIZE         2        /* contiene CS y DS  */
```

```
/* Table de descriptores globales.  1 a 7 son  prescritos por la BIOS. */
#define GDT_INDEX        1     /* descriptor GDT  */
#define IDT_INDEX        2     /* descriptor IDT  */
#define DS_INDEX         3     /* kernel DS */
#define ES_INDEX         4     /* kernel ES (386: flag 4 Gb) */
#define SS_INDEX         5     /* kernel SS (386: monitor SS) */
#define CS_INDEX         6     /* kernel CS */
#define MON_CS_INDEX     7 /* temp for BIOS (386: monitor CS ) */
#define TSS_INDEX        8     /* kernel TSS */
#define DS_286_INDEX     9   /* segmento fuente de 16 bits*/
#define ES_286_INDEX    10   /* segmento destino de 16 bits*/
#define VIDEO_INDEX     11   /* segmento de memoria de video */
#define DP_ETH0_INDEX       12        /* Western Digital Etherplus buffer */
#define DP_ETH1_INDEX       13        /* Western Digital Etherplus buffer */
#define FIRST_LDT_INDEX 14            /* resto de descriptors son LDT's */

#define GDT_SELECTOR      0x08      /* (GDT_INDEX * DESC_SIZE) bad for asld */
#define IDT_SELECTOR      0x10      /* (IDT_INDEX * DESC_SIZE) */
#define DS_SELECTOR       0x18      /* (DS_INDEX * DESC_SIZE) */
#define ES_SELECTOR       0x20      /* (ES_INDEX * DESC_SIZE) */
#define FLAT_DS_SELECTOR  0x21   /* menos privilegio ES */
#define SS_SELECTOR       0x28      /* (SS_INDEX * DESC_SIZE) */
#define CS_SELECTOR       0x30      /* (CS_INDEX * DESC_SIZE) */
#define MON_CS_SELECTOR   0x38   /* (MON_CS_INDEX * DESC_SIZE) */
#define TSS_SELECTOR      0x40      /* (TSS_INDEX * DESC_SIZE) */
#define DS_286_SELECTOR   0x49     /* (DS_286_INDEX * DESC_SIZE + 1) */
#define ES_286_SELECTOR   0x51     /* (ES_286_INDEX * DESC_SIZE + 1) */
#define VIDEO_SELECTOR    0x59     /* (VIDEO_INDEX * DESC_SIZE + 1) */
#define DP_ETH0_SELECTOR  0x61   /* (DP_ETH0_INDEX * DESC_SIZE) */
#define DP_ETH1_SELECTOR  0x69   /* (DP_ETH1_INDEX * DESC_SIZE) */

/* Tabla de descritores locales. */
#define CS_LDT_INDEX     0  /* process CS */
#define DS_LDT_INDEX     1  /* process DS=ES=FS=GS=SS */

/* Privileges. */
#define INTR_PRIVILEGE   0  /* kernel y manejador de interrupciones */
#define TASK_PRIVILEGE   1
#define USER_PRIVILEGE   3

/* 286 hardware constantes. */

/*Vector de Exception . */
#define BOUNDS_VECTOR       5       /* bounds check failed */
#define INVAL_OP_VECTOR     6       /* código de operación no válido */
#define COPROC_NOT_VECTOR   7  /*coprocesador no disponible */
#define DOUBLE_FAULT_VECTOR 8
#define COPROC_SEG_VECTOR   9   /* overrun en segmento de coprocesador*/
#define INVAL_TSS_VECTOR   10      /* TSS inválido*/
#define SEG_NOT_VECTOR     11       /* segmento no presente */
#define STACK_FAULT_VECTOR 12  /* excepción de pila */
#define PROTECTION_VECTOR  13   /* error de protección general*/

/* Selector bits. */
#define TI           0x04 /* indicador de Tabla a la que indexa(GDT o LDT) */
#define RPL           0x03          /* nivel de privilegio requerido */
```

```
/* Descriptor structure offsets. */
#define DESC_BASE        2      /* se utiliza como base_low */
#define DESC_BASE_MIDDLE 4      /* se utiliza como base_middle */
#define DESC_ACCESS      5      /* se utiliza como access byte */
#define DESC_SIZE        8      /* tamaño de la estructura segdesc_s */

/* Tamaños de segmento. */
#define MAX_286_SEG_SIZE 0x10000L

/* Base, y tamaño  y desplazamiento del límite. */
#define BASE_MIDDLE_SHIFT   16    /* desplazamiento para base --> base_middle */

/* Access-byte and type-byte bits. */
#define PRESENT       0x80      /* set for descriptor present */
#define DPL           0x60      /* descriptor privilege level mask */
#define DPL_SHIFT        5
#define SEGMENT       0x10      /* set for segment-type descriptors */

/* Access-byte bits. */
#define EXECUTABLE    0x08 /* se activa para segmento ejecutable*/
#define CONFORMING    0x04/* se activa para "conforming segment" si tiene acceso ejecutable */
#define EXPAND_DOWN   0x04      /* set for expand-down segment if !executable*/
#define READABLE      0x02    /* se activa para segmentos con permisos de lectura si tienen acceso ejecutable */
#define WRITEABLE     0x02    /* se activa para segmentos con permisos de escritura si tienen acceso ejecutable */
#define TSS_BUSY      0x02    /* se activa si el descriptor TSS está ocupado */
#define ACCESSED      0x01    /* se activa si el segmento se ha accedido */

/* Special descriptor types. */
#define AVL_286_TSS      1     /* disponible 286 TSS */
#define LDT              2     /* tabla de descriptores locales */
#define BUSY_286_TSS     3    /* establece transparencia de cara al software*/
#define CALL_286_GATE    4  /* no usado */
#define TASK_GATE        5     /* solo usado por el debugger */
#define INT_286_GATE     6    /* puerta de interrupció̧m, usado por todos los vectores */
#define TRAP_286_GATE    7  /* no usado */

/* Constantes extra del hardware del 386 . */

/* Números de vector de excepción. */
#define PAGE_FAULT_VECTOR   14
#define COPROC_ERR_VECTOR   16 /* error del coprocessador */

/* Desplazamiento de la estructura descriptor. */
#define DESC_GRANULARITY     6    /* to granularity byte */
#define DESC_BASE_HIGH       7       /* to base_high */

/* Base, y tamaño y desplazamiento del límite.. */
#define BASE_HIGH_SHIFT     24       /* desplazamiento para base --> base_high */
#define BYTE_GRAN_MAX   0xFFFFFL   /* tamaño máximo para byte granular segment */
#define GRANULARITY_SHIFT   16   /* Desplazamiento para límite--> granularity */
#define OFFSET_HIGH_SHIFT   16     /* Desplazamiento para  (gate) offset --> offset_high */
#define PAGE_GRAN_SHIFT     12      /* Desplazamiento extra para page granular limits */

/* Type-byte bits. */
#define DESC_386_BIT      0x08          /* Los tipos del 386 se obtienen por ORing*/
                                  /* Las LDT´s y puertas de tarea no lo necesitan*/
```

```
/* Granularity byte. */
#define GRANULAR          0x80          /* Se activa para  granularidad 4K*/
#define DEFAULT           0x40 /* Se activa para  32-bit defaults (segmento ejecutable) */
#define BIG               0x40       /* Se activa para  "BIG" (expand-down seg) */
#define AVL               0x10       /* 0 para representar disponible */
#define LIMIT_HIGH        0x0F          /* máscara para bits más representativos del límite. */
```

EXCEPTION.C

Contiene el manejador de excepciones, el cual se llama desde la parte en ensamblador del código de manejo de excepciones (mpx386.s). Las excepciones originadas por los procesos de usuario se convierten en *SIGNALS*, pero las excepciones originadas por el sistema indican errores graves y es causa de una alarma.

El array ***ex_data[*** determina el mensaje de error a mostrar en caso de excepción del S.O. o la señal que se enviara al proceso de usuario que ha causado la excepción.

Debido a que los primeros procesadores Intel no generaban todas las excepciones, el tercer campo indica el mínimo modelo de Intel capaz de generar cada una de ellas.

mpx386.s

Las transiciones hacia el kernel pasan a través de este fichero. Las transiciones pueden ser causadas por mensajes o por varias de las interrupciones. La primera se produce bien por una llamada al sistema, bien por excepciones, o bien por interrupciones, el resto solo por interrupciones hardware. El número de reentradas se guarda en *k_reenter*.

panic(mensaje,numero)

Esta función muestra un mensaje de error y termina la ejecución. Si la *MM* o *FS*. Si hay mensaje lo muestra y el número, y llama a la función *wreboot*, que lo que hace es esperar por una tecla y mostrar información de debug.

wreboot(x)

Muestra “Hit ESC to reboot, F-keys for debug dumps” y espera por que se pulse una tecla. Si se pulsa ‘ESC’ muestra “Rebooting...” y reinicializa los controladores de interrupción con información de la BIOS y resetea. Si por el contrario se pulsa alguna de las teclas de función, los que se muestran son distintos volcados de memoria, tablas, etc.

PUBLIC void exception(vec_nr)
{
ex_s : estructura que contiene el mensaje a mostrar, la señal a enviar y un código que identifica el procesador mínimo que

ex_s ex_data[] : array que contiene todas las posibles excepciones que se pueden dar.

Se guarda el estado del proceso con propósitos de depurar el código.
Se obtiene del array de excepciones el elemento que corresponde al número de excepción *vec_nr*
Se comprueba si se trata de una interrupción no enmascarable.
Si lo es, se muestra un mensaje y se retorna.

Se comprueba si es la primera reentrada del kernel y si el proceso que ha causado la excepción es de usuario.
Si es así, se le manda al proceso la señal correspondiente y se retorna.

Se comprueba que es una excepción soportada por el procesador en el que esta corriendo MINIX.
Si no lo es se muestra un mensaje y se retorna.
Si lo es quiere decir que el proceso que la originó fue el kernel, por lo que hay que generar una alarma y resetear el sistema si procede.

/* This file contains a simple exception handler. Exceptions in user

```
 * processes are converted to signals.  Exceptions in the kernel, MM and
 * FS cause a panic.
 */

#include "kernel.h"
#include <signal.h>
#include "proc.h"

PUBLIC void exception(vec_nr)
unsigned vec_nr;
{
/* Ha ocurrido una excepción o interrupción inesperada. */
  struct ex_s {
        char *msg;
        int signum;
        int minprocessor;
  };
  static struct ex_s ex_data[] = {
        "Divide error", SIGFPE, 86,
        "Debug exception", SIGTRAP, 86,
        "Nonmaskable interrupt", SIGBUS, 86,
        "Breakpoint", SIGEMT, 86,
        "Overflow", SIGFPE, 86,
        "Bounds check", SIGFPE, 186,
        "Invalid opcode", SIGILL, 186,
        "Coprocessor not available", SIGFPE, 186,
        "Double fault", SIGBUS, 286,
        "Copressor segment overrun", SIGSEGV, 286,
        "Invalid TSS", SIGSEGV, 286,
        "Segment not present", SIGSEGV, 286,
        "Stack exception", SIGSEGV, 286,        /* STACK_FAULT ya usada */
        "General protection", SIGSEGV, 286,
        "Page fault", SIGSEGV, 386,             /* no consecutivo */
        NIL_PTR, SIGILL, 0,                     /* probablemente es un caso de "software trap" */
        "Coprocessor error", SIGFPE, 386,
  };
  register struct ex_s *ep;
  struct proc *saved_proc;

  saved_proc= proc_ptr; /* Almacena proc_ptr, porque pudiera ser cambiado por
                         * sentencias del debugger.
                         */

  ep = &ex_data[vec_nr];

  if (vec_nr == 2) {                    /*  NMI espúreo en algunas máquinas */
        printf("got spurious NMI\n");
        return;
  }

  if (k_reenter == 0 && isuserp(saved_proc)) {
        unlock();                       /* Esto está protegido, como sys_call() */
        cause_sig(proc_number(saved_proc), ep->signum);
        return;
  }

  /* Esto no se espera que ocurra. */
```

```
  if (ep->msg == NIL_PTR || processor < ep->minprocessor)
        printf("\nIntel-reserved exception %d\n", vec_nr);
  else
        printf("\n%s\n", ep->msg);
  printf("process number %d, pc = 0x%04x:0x%08x\n",
        proc_number(saved_proc),
        (unsigned) saved_proc->p_reg.cs,
        (unsigned) saved_proc->p_reg.pc);
  panic("exception in system code", NO_NUM);
}
```

C U E S T I O N E S

**** C o m o s o l u c i o n a m o s e l a p r o v e c h a m i e n t o d e m e m o r i a s i s o l o d i r e c c i o n a m o s 2 ^ 2 0 e n t r a d a s d e 4 B y t e s , e n u n a P a g - T a b l e .**

**** E x p l i c a r l o s c o m a n d o s d e i n t e r r u p c i o n e s v í a s o f t w a r e .**

**** P r o c e s o q u e s e r e a l i z a c u a n d o s e p r o d u c e u n a i n t e r r u p c i ó n e n M i n i x .**

**** P o r q u e d e s a c t i v a m o s l a i n t e r r u p c i o n e s c u a n d o u n a d e e l l a e s t á e n e j e c u c i ó n .**

**** D e s c r i b i r c o m o s e t r a t a n l a s e x c e p c i o n e s e n M I N I X .**