EJEMPLOS DE MODO PROTEGIDO

Programación en Assembler

Mariano Cerdeiro
m.cerdeiro@soix.com.ar

1era Edición
Octubre de 2004

Buenos Aires
Argentina

http://www.soix.com.ar/links/mpbyexamples.html

Cerdeiro, Mariano
Ejemplos de modo protegido: programación en assembler. -1a ed. - Buenos Aires: el autor, 2004.
Internet; 91 p.

ISBN 987-43-8245-7

1. Programación-Computación I. Título
CDD 005

ISBN 987-43-8245-7

A mis profesores Lic. Mirta Bindstein, Ing. Alejandro Furfaro e Ing. Marcelo Doallo.

Agradezco a Manuela Cerdeiro, mi hermana, quien tradujo el documento al español.

```
; Introducción
;
; Este libro tiene como finalidad cubrir la gran brecha existente entre
; los libros y manuales de modo protegido y lograr correr un programa
; sin que el mismo bootee la PC por algún error. Para ello se explica la
; teoría necesaria para entender el código a continuación.
;
; Cabe aclarar que este libro NO es de teoría de modo protegido, por
; ello  se recomienda llevar el estudio teórico con algún otro libro del
; tema. Personalmente recomiendo los manuales de Intel, que el día de la
; fecha se encuentran en:
;
; http://developer.intel.com/design/Pentium4/documentation.htm#man
;
; De este link se pueden bajar los manuales del Pentium 4. Los
; conocimientos sobre modo protegido se pueden obtener de los capítulos
; 2, 3, 4, 5 y 6 del IA-32 Intel Architecture Software Developer's
; Manual Volume 3: System Programming Guide.
;
; Para programar en modo protegido se puede utilizar cualquier editor de
; texto, como vi, notepad, edit u otro. Para compilar se utiliza el nasm
; y para correr los programas, el bochs, con el freedos cargado en una
; imagen de disquete, o booteando la PC en modo real, ya sea mediante un
; disquete o booteando en modo real, si el sistema operativo lo permite.
;
; El nasm es un compilador de assembler, se puede bajar de:
;
; http://nasm.sourceforge.net/
;
; El bochs es una PC virtual, y es una herramienta IMPRESCINDIBLE para
; poder debugear el código de modo protegido.
;
; http://bochs.sourceforge.net/
;
; Y el Free DOS:
;
; http://www.freedos.org/
;
; La instalación de los programas dependerá del sistema operativo, del
; freedos sólo es necesario bajar la imagen e indicar en el archivo de
; configuración del bochs dónde encontrarla.
;
; Por último es recomendable tener a disposición un manual de
; interrupciones y de hard. Para tal fin existe la página de Ralf Brown:
;
; http://www.ctyme.com/rbrown.htm
;
; El libro está formado por nueve ejemplos de los siguientes temas:
;     - ej00.asm  tutorial de DOS
;     - ej01.asm  entrar y salir a modo protegido
;     - ej02.asm  utilizando un segmento de datos en modo protegido
;     - ej03.asm  interrupciones en modo protegido
;     - ej04.asm  excepciones en modo protegido
;     - ej05.asm  multitarea en modo protegido
;     - ej06.asm  salto de niveles de privilegio en modo protegido
;     - ej07.asm  modo protegido 32 bits y pila expand down
;     - ej08.asm  paginación en modo protegido
;
; Personalmente espero que el libro les sea útil, y que me hagan llegar
; cualquier comentario o duda. A su vez pido disculpas por cualquier
; error.
```

```
; ej00.asm - programa hola mundo
;
; Protected Mode by Examples - 1era Edición - Octubre 2004
;
; Autor:
; Mariano Cerdeiro
; <m.cerdeiro@soix.com.ar>
; http://www.soix.com.ar/links/mpbyexamples.html
;
; compilar: nasm ej00.asm -o ej00.com
;
; El primer ejercicio tiene por finalidad mostrar la estructura básica
; de un programa en nasm y el modo en que es cargado en memoria por el
; DOS. También se explican algunos conceptos básicos, como la
; segmentación en modo real y las diferencias entre un segmento de 16 y
; uno de 32 bits.
;
; En modo real (el modo en que se enciende la CPU y en el que se
; mantiene cuando corre el sistema operativo DOS) se accede a la memoria
; mediante segmentos y desplazamientos. El desplazamiento (u offset) es
; un valor de 16 bits que se suma a la base del segmento para conocer la
; memoria lineal y física (las cuales, en modo real, coinciden).
;
; Siempre que se direcciona memoria, se hace mediante un segmento y un
; offset. Para conocer cuál es la posición física de la memoria se deben
; sumar la base del segmento y el offset. En modo real, la base de un
; segmento es 16 veces su valor.
;
; Por ejemplo: el segmento 0040:0037, al escribirlo de esta forma, se
; hace referencia al segmento:offset (ambos en hexadecimal) que apunta a
; la dirección de memoria 00437h.
;
; Por lo cual, para acceder a cualquier posición de memoria, existen
; 4096 formas distintas de hacerlo ya que, por ejemplo, 4000:FFFF
; será físicamente la misma posición que 4001:FFEF (físicamente ambas
; direcciones serán 4FFFF, o sea, el byte número 327679 de la memoria).
;
;
; El DOS es quien carga los programas .com, que son los únicos que se
; utilizan en este libro. A diferencia de los archivos .exe, que poseen
; un header con información para el sistema, los archivos .com son
; directamente copiados a memoria y ejecutados, pero existen ciertas
; reglas a conocer:
;     - cuando se ejecuta un archivo .com, el mismo se carga en memoria
;       a partir del primer segmento libre de memoria que tenga
;       disponible el DOS.
;     - se le asigna toda la memoria hasta los 640 Kbytes.
;     - se carga el contenido del del archivo .com a partir del offset
;       100h, quedando así 256 bytes para el sistema, los cuales se
;       denominan PSP (Program Segment Prefix).
;
; EL PSP almacena información de la aplicación (en Internet se puede
; averiguar cómo está conformado). Sin embargo, lo más importante a
; saber es que en el offset 80h se encuentra un byte que indica la
; cantidad de caracteres que contiene la línea de comandos, y a partir
; del offset 81h se encuentra la línea de comandos.
;
; Al cargar el .com se inicializan los registros ax,bx,cx,dx,bp,si,di en
; 0, el sp en FFFCh y cs,ds,es,ss en el segmento de memoria libre, y el
; ip en 100. Por lo que se puede observar que el código y los datos se
; superponen en el mismo segmento y éstos con la PILA. Por eso se debe
```

```
; tener especial CUIDADO cuando se utiliza mucha pila, ya que una
; excesiva cantidad de pushs podría sobrescribir el código o los datos
; del programa.
;
; Lo primero que se escribe al comenzar un programa es la directiva
; use16 o use32. Ésta indica para qué tipo de segmento debe ser
; compilado el programa (para un segmento de 16 o de 32 bits).
;
use16
;
;
; Segmentos de 16 y de 32 bits.
;
; Que un segmento sea de 16 o de 32 bits, indica qué tipo de registros y
; direccionamiento se utilizan en forma predeterminada, de modo que las
; instrucciones que utilizan registros y direccionamientos
; predeterminados ocupan menos espacio. Veamos un ejemplo:
;
; 89 C8                 mov ax,cx
;
; Se puede  observar que la instrucción se codifica como 89 C8. Se trata
; de una instrucción que ocupa 2 bytes. Haciendo lo mismo con eax y ecx
; se obtiene:
;
; 66 89 C8              mov eax,ecx
;
; Aquí, al utilizar eax y ecx, se ocupa un byte más, el 66. Éste
funciona
; como prefijo, y su función es indicar al procesador que la siguiente
; instrucción se debe decodificar utilizando el tipo de registros no
; predeterminado. De esto que se deduce correctamente que las
; instrucciones anteriores han sido compiladas para 16 bits. Si, por el
; contrario, se compilan estas dos mismas instrucciones en código 32
; bits, se obtiene:
;
; 89 C8                 mov eax,ecx
; 66 89 C8              mov ax,cx
;
; Se puede ver que la instrucción más corta es la de 32 bits. 66h no es
; el único prefijo que existe, también existe el 67h, que cumple una
; función similar, pero no con el tamaño del registro, sino con el tipo
; de direccionamiento (si es de 16 o de 32 bits). También existen otros
; prefijos, pero no están relacionados con el tipo de segmento.
;
; Por lo que una misma instrucción se puede codificar de dos formas
; distintas, según se trate de un segmento de 16 o de uno de 32 bits (se
; le debe indicar al procesador de qué tipo de segmento de código se
; trata). Para tal fin existe un bit en la definición del segmento. Sin
; embargo, en modo real, los segmentos pueden ser únicamente de 16 bits,
; por lo que se dejarán para más adelante los segmentos de 32 bits y se
; utilizarán, hasta el ejemplo 6 inclusive, segmentos de 16 bits.
;
org 100h
;
; Debido a que el compilador desconoce que los archivos .com son
; cargados y ejecutados a partir del offset 100h, se debe indicar el
; offset inicial. De esta forma, al referirse a la cadena mensaje (leer
; líneas más delante), mediante el puntero, el compilador sabrá que
; el mismo tiene valor de 102h (suponiendo 2 bytes que ocupa el jmp
; inicio).
;
```

```
jmp inicio
;
; En los archivos .com se cuenta con un solo segmento para código y
; datos, como se mencionó. Por eso se debe hacer un jmp para saltar los
; datos, o bien colocar los datos al final. En todos los ejemplos se
; colocarán los datos al comienzo.
;
mensaje     db    'Hola Mundo$'
;
; mensaje es la única variable de este ejercicio y es un array de db
; (data bytes), donde el primer byte es H, el segundo O y sucesivamente.
; El último caracter es un símbolo $, que es interpretado por el
; servicio DOS como fin del mensaje.
;
inicio:
;
; Es el label donde comienza el código.
;
      mov ah,09h
      mov dx,mensaje
      int 21h
;
; En estas tres líneas se carga en ah el valor 9, en dx el offset que
; corresponde al mensaje. Al llamar a la interrupción 21h interrupción
; de DOS con ah en 09 se imprime en pantalla lo apuntado por ds:dx
; hasta encontrar un caracter $.
;
      mov ah,4ch
      int 21h
;
; Estas dos líneas llaman nuevamente al DOS pero al servicio 4C, el cual
; termina el programa volviendo al DOS.
;
```

```
; ej01.asm - entrar y salir a modo protegido
;
; Protected Mode by Examples - 1era Edición - Octubre 2004
;
; Autor:
; Mariano Cerdeiro
; <m.cerdeiro@soix.com.ar>
; http://www.soix.com.ar/links/mpbyexamples.html
;
; compilar: nasm ej01.asm -o ej01.com
;
; Este programa entra y sale de modo protegido. Se mantiene en modo
; protegido hasta que se presione la tecla escape para entonces retornar
; a modo real.
;
; Entrar a modo protegido implica setear el bit 0 del registro cr0 PE
; (Protection Enable). Para retornar se debe hacer lo inverso, resetear
; nuevamente el bit PE.
;
; Como su nombre lo indica, modo protegido es un modo en el que el
; procesador brinda al SO herramientas para poder proteger tanto a una
; aplicación de otra, como al SO de todas las aplicaciones. De esta
; forma, si alguna genera un error, bastará con cerrarla. De modo que,
; si, estando ya en modo protegido (por ejemplo en una ventana de
; Windows), se intenta setear el bit PE (el cual ya estará seteado), el
; programa será cerrado por el sistema operativo, indicando que no se
; puede correr la aplicación.
;
use16
org 100h
;
; Al tratarse de un archivo .com se coloca el org correspondiente, y
; como se parte de modo real se debe usar segmento de 16 bits.
;
jmp inicio
;
; Se saltean las variables por costumbre, a pesar de que en este caso no
; hay.
;
inicio:
;
;        mov eax,1
;break:  cmp eax,1
;        je break
;
; IMPORTANTE:
; Las anteriores líneas se colocan aquí para poder comenzar a utilizar
; el bochs como herramienta de depuración. Lo que se debe hacer es
; quitar los comentarios de las líneas y compilar el programa. Al correr
; el bochsdbg, pulsar ctrl-c en la consola de debug, una vez ejecutado
; el programa ej01.com. Del manual del bochs y del help de la línea de
; comandos se puede obtener ayuda para entender su funcionamiento.
;
; Con set $eax=0 <ENTER> se hace cierta la condición para salir del
; bucle, para luego, con s <ENTER>, ejecutar paso a paso el programa.
; Este método de depuración se puede copiar en cualquier parte del
; programa. Es conveniente colocarlo después de la parte del programa
; que no falla, para hacer el proceso de depuración lo más simple
; posible. También se lo puede utilizar para saber qué está sucediendo
; exactamente al ejecutar estos u otros ejercicios.
;
```

```
; Se debe RECORDAR eliminar estas líneas cuando se desee correr sobre la
; la PC real, ya que el programa se bloqueará indefinidamente hasta una
; interrupción o, en el caso en que estén inhabilitadas, hasta que se
; reinicie la PC.
;
     cli
;
; En modo protegido la tabla de interrupciones se maneja de forma
; distinta e incompatible con la de modo real, por lo que se deben
; inhabilitar las interrupciones hasta conocer y entender el mecanismo
; de interrupciones (ver ej03.asm).
;
     mov eax,cr0             ;lee el registro de control 0.
;
; El bit PE es el bit 0 del cr0. Para mantener el resto de los bits
; inalterados, antes de modificar el registro, se lee su contenido y
; se setea únicamente el bit deseado. Las formas de acceder al bit PE
; son:
;
; mov cr0,exx y mov exx,cr0
; lmsw reg16 y smsw reg16
;
; A diferencia del mov al registro cr0, el segundo grupo de
; instrucciones sólo accede a la parte baja del cr0 (el cual en el
; 80286 se denomina Machine Status Word). Por otro lado, las
; instrucciones mov del cr0 no se pueden correr estando en modo
; protegido en los niveles NO privilegiados, en los que se corre, por
; ejemplo, el programa en una ventana bajo Windows. Por lo tanto, para
; saber si el procesador se encuentra ya en modo protegido, se debe
; utilizar smsw. En este ejercicio se lee directamente del cr0, por lo
; que el programa se cerrará si el procesador se encuentra en modo
; protegido. En ejemplos posteriores se utilizará la instrucción smsw
; para indicar en la pantalla, mediante un mensaje, si el procesador se
; encuentra en modo protegido.
;
     or al,1                 ;setea el bit PE Protection Enable.
;
; Se coloca el bit correspondiente en 1.
;
     mov cr0,eax             ;pasa a modo protegido.
;
; Se pasa a modo protegido.
;
     jmp short $+2           ;vacía la cola de ejecución.
;
; Intel recomienda vaciar la cola de ejecución al entrar y salir de modo
; protegido. Si bien es posible que el programa funcione a pesar de no
; hacerlo, Intel no lo puede asegurar.
;
          mp:in al,60h       ;lee el scan-code del teclado.
;
; Se lee el contenido del puerto 60h, o sea, el scan-code del teclado.
;
          dec al             ;compara con esc.
;
; Se decrementa el scan-code para compararlo con 0, por ser esto más
; simple que comparar con uno.
;
          jnz mp             ;si es esc, se retorna a modo real.
;
; Si el scan-code leído del puerto 60h es distinto de 1, significa que
```

```
; no se pulsó la tecla escape, de modo que se sigue dentro del bucle.
;
      mov eax,cr0
      and al,0feh             ;resetea el PE.
;
; Se lee nuevamente el valor de cr0 y se coloca en cero el bit de PE.
;
      mov cr0,eax             ;retorna a modo real.
      jmp short $+2           ;vacía la cola de ejecución.
;
; Se carga el cr0, con lo que se pasa nuevamente a modo real. Luego se
; realiza nuevamente un salto corto para vaciar la cola de ejecución.
;
      sti                     ;habilita las interrupciones.
      mov ah,4ch              ;retorna al DOS mediante el
      int 21h                 ;servicio 4ch de la int 21h.
;
; Se habilitan las interrupciones nuevamente y se retorna al sistema.
;
```

```
; ej02.asm - programa que guarda el BIOS en un archivo
;
; Protected Mode by Examples - 1era Edición - Octubre 2004
;
; Autor:
; Mariano Cerdeiro
; <m.cerdeiro@soix.com.ar>
; http://www.soix.com.ar/links/mpbyexamples.html
;
; compilar: nasm ej02.asm -o ej02.com
;
; Este ejercicio guarda en el archivo bios.dat la parte de BIOS del
; start-up. Como es sabido, los procesadores, a partir de la 80386
; comienzan a ejecutar el código que encuentran en memoria en el
; segmento F000 y offset FFF0. Pero este segmento, a pesar de
; encontrarse en modo real, no tiene base igual a F0000h, como se podría
; suponer, sino que su base es FFFF0000. A pesar de iniciarse en
; modo real, el comportamiento del segmento de código no es el normal de
; modo real. La primera vez que se modifique el cs, su base será cargada
; normalmente (o sea, la base será 16 veces el valor del cs), en tanto
; la base del segmento de código permanecerá en FFFF0000. Ya que el
; procesador comienza ejecutando esta sección de la memoria, es de
; suponer que en la misma debe haber memoria ROM cuyo código inicialice
; a la PC, o al menos los bytes suficientes para llevar a cabo un salto.
; Es por esto que las PCs tienen 64 Kbytes de ROM a partir de la
; posición de memoria física FFFF0000 y hasta FFFFFFFF.
;
; Para copiar el BIOS a un archivo es necesario entrar a modo protegido,
; donde se tendrá acceso a los 4Gb de memoria, y solamente al primer
; mega. De esta forma se tendrá acceso a la ROM del BIOS. Por otro lado,
; se debe guardar esta información en un archivo. Lamentablemente, el
; modo protegido también tiene un defecto y es que no hay rutinas de
; BIOS ni DOS para guardar un archivo en disco, por lo cual se puede
; hacer un driver de disco para modo protegido (tema que excede al
; libro), o bien copiar el BIOS por debajo del mega de memoria,
; retornar a modo real y, una vez en modo real, grabar el archivo
; mediante las llamadas al sistema convencionales de DOS.
;
; Ahora lo que se necesita conocer es cómo se direcciona la memoria en
; modo protegido, para así poder leer el BIOS que se encuentra al final.
; En modo protegido, al igual que en modo real, los segmentos tienen una
; base y un límite, pero además tienen otros atributos, como permisos de
; acceso y otros indicadores. A diferencia del modo real, donde la base
; de un segmento es el valor del selector (cs,ds,es,fs,gs,ss)
; multiplicado por 16, en modo protegido la misma puede ser cualquier
; valor dentro de los 4Gb. Asimismo el límite, que en modo real es de
; 65536 bytes, puede ser de cualquier valor entre 0 y 4 Gb. Claramente
; se puede intuir que toda esta información no puede ser almacenada en
; 16 bits que poseen cs, ds, es, fs, gs y ss.
;
;
; Segmentación en modo protegido
;
; En modo protegido los selectores cs, ds, es, fs, gs y ss son un
; puntero a una entrada de una tabla, la cual almacena información sobre
; cómo es el segmento: base, límite y derechos de acceso. El formato de
; los selectores en modo protegido es el siguiente:
;
;  15          3  2  1  0
; +-------------+---+----+
; |    Índice   |TI | PL |
```

```
; +-------------+---+----+
;
;     - ÍNDICE: Indica qué entrada de la tabla se debe utilizar.
;     - TI (Table Indicator): Indica qué tabla se debe utilizar, la
;       GDT o la LDT.
;     - PL (Privilege Level): Indica qué nivel de privilegio tiene este
;       selector.
;
; Los selectores pueden apuntar a una de dos tablas de descriptores, a
; la GDT (Global Descriptor Table) o a la LDT (Local Descriptor Table).
; Estas tablas, como el nombre indica, están formadas por descriptores.
; Cada descriptor ocupa 8 bytes y consta de:
;
;  31                    16 15                     0
; +----------+-+-+-+-+-----+-+-+-+-+----+-----------+
; |   Base   | | | | |Limit|P|D P|#|Tipo|   Base    | 4
; |   31:23  | | | | |19:16| | L |S|    |   22:16   |
; +----------+-+-+-+-+-----+-+-+-+-+----+-----------+
;
; +------------------------+------------------------+
; |       Base 15:0        |      Limit 15:0        | 0
; |                        |                        |
; +------------------------+------------------------+
;  31                    16 15                     0
;
;     - Base 31:0: Base LINEAL del segmento, puede ser cualquier
;       posición de memoria LINEAL dentro de los 4Gb lineales
;       direccionables.
;     - D/B: Su función es variable. Si es un segmento de código indica
;       si el segmento es de 16 o 32 bits. Si es un segmento de datos de
;       pila, indica el tamaño de registros de pila, por ejemplo, si
;       está seteado, a pesar de que se pushee ax, decrementará el
;       puntero de la pila en 4 para mantenerla alineada en 4 bytes.
;       Si es un segmento de datos expand down, también indica si el
;       límite superior es 0FFFFh ó 0FFFFFFFFh.
;     - AVL: A disposición del programador del sistema.
;     - G: Granularidad. Como se puede ver, hay solamente 20 bits para
;       indicar el largo del segmento. Este bit cambia las unidades del
;       límite de bytes a 4096 bytes. Entonces, al estar G seteado y el
;       límite al máximo, se obtendrá un límite de 4Gb.
;     - Límite 19:0: Límite del segmento en cantidad de bytes o 4096
;       bytes (ver G).
;     - P: Bit de presencia. Indica si el segmento está presente en
;       memoria o si está en memoria virtual. La finalidad de este bit
;       es poder administrar la memoria virtual en segmentación. Hoy en
;       día se suele utilizar la paginación para la memoria virtual,
;       pero existe la posibilidad de utilizar la segmentación. Para
;       eso, al faltar memoria, se quita el segmento resetanado el bit
;       de presencia y se podrán utilizar todos los campos salvo el 5
;       byte para indicar en qué parte del disco se ha almacenado este
;       segmento. De esta forma se podrá usar la sección de memoria que
;       el segmento ocupaba. En el caso de que alguna aplicación o el
;       mismo sistema desee utilizar el segmento no presente, se
;       generará una excepción (ver ejercicio 4) que podrá cargar el
;       segmento donde sea correcto y continuar la ejecución normal del
;       programa.
;     - DPL: Nivel de privilegio del descriptor (Descriptor Privilege
;       Level).
;     - #S: Indica si se trata de un segmento de sistema o no (0
;       significa que es de sistema).
;     - TIPO: Si #S es 1, se trata de un segmento, entonces el campo
```

```
;       tipo puede ser alguno de los siguientes:
;
;       Campo TIPO con #S en 1
;
;               Segmentos de datos (C/#D en 0)
;
;       C/#D    E       W       A       Descripción
;       0       0       0       0       sólo lectura
;       0       0       0       1       sólo lectura, accedido
;       0       0       1       0       lectura/escritura
;       0       0       1       1       lectura/escritura, accedido
;       0       1       0       0       sólo lectura, expand down
;       0       1       0       1       sólo lectura, expand down,
;                                       accedido
;       0       1       1       0       lectura/escritura, expand down
;       0       1       1       1       lectura/escritura, expand donw,
;                                       accedido
;
;               Segmentos de código (C/#D en 1)
;
;       C/#D    C       R       A
;       1       0       0       0       ejecución
;       1       0       0       1       ejecución, accedido
;       1       0       1       0       ejecución/lectura
;       1       0       1       1       ejecución/lectura, accedido
;       1       1       0       0       ejecución, conforming
;       1       1       0       1       ejecución, conforming, accedido
;       1       1       1       0       ejecución/lectura, conforming
;       1       1       1       1       ejecución/lectura, conforming,
;                                       accedido
;
; NOTA: El bit denominado aquí como C/#D, en el manual de Intel no tiene
; denominación. La misma se realiza aquí sólo para su mejor comprensión.
; El mismo indica si se trata de un segmento de datos, en caso de estar
; reseteado, o de un segmento de código, en caso de estar seteado.
;
;
;               Continuando con el campo tipo, el mismo, en caso de ser
;               el campo S=0, se tratará de un descriptor de sistema.
;
;       0       0       0       0       Reservado
;       0       0       0       1       16-Bit TSS (no ocupada)
;       0       0       1       0       LDT
;       0       0       1       1       16-Bit TSS (ocupada)
;       0       1       0       0       16-Bit Call Gate
;       0       1       0       1       Task Gate
;       0       1       1       0       16-Bit Interrupt Gate
;       0       1       1       1       16-Bit Interrupt Gate
;       1       0       0       0       Reservado
;       1       0       0       1       32-Bit TSS (no ocupada)
;       1       0       1       0       Reservado
;       1       0       1       1       32-Bit TSS (ocupada)
;       1       1       0       0       32-Bit Call Gate
;       1       1       0       1       Reservado
;       1       1       1       0       32-Bit Interrupt Gate
;       1       1       1       1       32-Bit Trap Gate
;
; Hasta aquí se trataron muchos nuevos conocimientos, que no serán tan
; sencillos ni fáciles de asimilar. Por eso parece razonable hacer un
; repaso. Para empezar, del 286 en adelante los registros CS,DS,ES,FS,GS
; y SS ya no se denominan registros de segmento sino selectores. Los
```

```
; mismos no apuntan ya a la base física dividido 16, sino que tienen un
; campo índice y uno tabla, que hacen referencia a una de dos tablas,
; que pueden ser la GDT o la LDT, ambas de descriptores. Los
; descritpores pueden ser de segmentos del sistema, según el campo S. Si
; son descriptores de segmentos pueden ser apuntados por CS,DS,ES, FS,
; GS y SS; en caso contrario tendrán otras finalidades que veremos más
; adelante.
;
; Siendo el objetivo de este problema leer el BIOS, que se encuentra en
; los últimos 64 Kbytes de memoria, se debe armar un segmento que apunte
; a ese sector de memoria física. Para eso se debe armar un descriptor
; que apunte a ese sector de memoria y cargar el selector que direcciona
; al descriptor. De esta forma se podrá leer el BIOS y copiarlo debajo
; del mega de memoria para así grabarlo en un archivo.
;
; Para grabar un archivo bajo DOS se utilizan las funciones de DOS (int
; 21h).
;
; Crear o truncar un archivo Int 21 ah=3C.
; CX = file attributes, 0 es un archivo normal.
;     DS:DX -> ASCIZ filename
;     Return ax=handler
;
; Cerrar un archivo Int 21 ah=3E.
; bx=handler
;
; Escribir en un archivo Int 21 ah=40.
; bx=handler
; cx=cantidad de bytes
; ds:dx -> datos a escribir
;
org 100h
comienzo:jmp inicio
;
archivo         db      'bios.dat',0
;
; Aquí se define un nombre para el archivo, que debe terminar en NULL,
; porque así interpreta el DOS el fin del nombre (no se lo debe
; confundir con cómo interpreta el fin de línea para imprimir un
; string).
;
gdtr    dw 8*3-1
        dd 0
;
; Hasta aquí se trataron la GDT y la LDT, que tienen descriptores (que
; son de 8 bytes) y que los selectores deben hacer referencia a algún
; descriptor. Sin embargo no se mencionó en qué posición de memoria se
; encuentran estas tablas y el procesador no lo puede adivinar. Para
; ello existe un registro llamado GDTR (Global Descriptor Table
; Register) formado por 6 bytes, que indica, en 1 word, el largo menos 1
; de la tabla y, en un double word, la base lineal de la tabla (para la
; LDT se utiliza otro método que no se estudiará aquí). Se coloca el
; largo, se indica 8 por el largo de cada descriptor, 3 porque se quiere
; utilizar 3 de ellos y -1 por la definición de límite.
;
; Al colocar la base pueden surgir dudas. ¿Dónde esta la base de la GDT?
; Se sabe dónde comienza: en gdt: (ver abajo). Pero, ¿dónde quedará
; esa posición cuando el DOS o algún loader cargue el programa? Es
; imposible saber esto de antemano, por lo cual se debe cargar la base
; en tiempo de ejecución. Como el programa comienza a ejecutarse en modo
; real, el CS es la base sobre 16 y el offset es gdt:. Para conocer la
```

```
; base se calcula CSx16+gdt:.
;
gdt:     resb 8
;
; A partir de aquí comienza la GDT. Se puede observar que la primera
; entrada se deja en blanco. La razón es que el micro NO UTILIZA el
; primer descriptor de la GDT, el cual se denomina descriptor NULO y
; puede ser referenciado por cualquier selector, con el fin de colocar
; los selectores no utilizados en algún valor.
;
        ;--- 08h Segmento de datos flat
        dw 0ffffh       ;límite 15.00
        dw 00000h       ;base 15.00
        db 000h         ;base 23.16
        db 10010010b    ;Presente Segmento Datos Read Write
        db 10001111b    ;G y limite 0Fh
        db 00h          ;base 31.24
;
; Para inicializar el segmento se debe conocer la base y el límite que
; dependen de lo que se quiere hacer. En este caso se coloca la base en
; 0 y el límite de 4Gb, el BIOS también se podría leer colocando la base
; en FFFF0000h y el límite en 0FFFFh. Un segmento que tiene base en 0 y
; límite 4Gb se denomina FLAT.
;
; Como el límite es mayor que el Mbyte, se debe setear la granularidad,
; entonces se expresa en unidades de 4 Kbytes: 4 Gb/4 Kbytes-1, o sea,
; 0FFFFFh. Donde justamente los 20 bits del límite están seteados. Es
; por esto que en el primer word los bits 15:0 del límite están
seteados.
; La base es 0. Luego se indica el tipo de segmento (de datos, de
; lectura o de escritura). Es indistinto como se setea el bit accedido.
; Este es automáticamente colocado en 1 por hard cada vez que se utiliza
; el descriptor. Así el SO puede llevar un control de cuánto se utiliza
; cada descriptor. Colocando en cero este bit periódicamente sabrá
; aproximadamente cada cuánto tiempo es accedido. El objetivo final es
; saber cuál es el segmento más indicado para enviar a memoria virtual.
; En el último byte se deben colocar la granularidad en 1 y el resto del
; límite también. El 8° byte, se coloca cero, que representa la parte
; alta de la base del segmento.
;
        ;--- 10h Segmento de datos de 64 Kb
        dw 0ffffh
        dw 0
        db 0
        db 10010010b
        db 0
        db 0
;
; El segundo descriptor se utiliza para mantener compatibilidad con
; el modo real. En modo real el límite es de 65535 bytes, por lo que es
; recomendable que, antes de retornar a modo real, todos los límites
; estén en este valor.
;
; En este descriptor el límite es de 0FFFFh (64 Kbytes). La base, al no
; ser conocida (ya que depende de dónde el DOS cargue el programa), se
; deja en blanco, para completarla en tiempo de ejecución con 16 veces
; el valor indicado por CS más lo que ocupe el código.
;
inicio:
        mov al,0d1h
        out 64h,al
```

```
        mov al,0dfh
        out 60h,al      ;Eanble A20
;
; A20 hace referencia a una AND que funciona como habilitación de la
; línea 20 del bus de direcciones. Cuando el A20 está en 0, la línea 20
; del bus de direcciones es cero siempre; cuando A20 está en 1, la línea
; A20 del bus de direcciones es la que corresponde según la referencia a
; memoria física que se haga. Si se intenta acceder a la dirección
; 3FFFFFh con la A20 en cero, se accederá a la dirección 2FFFFFh, o sea
; que el bit 20 de las direcciones es colocado en cero por la AND.
;
; La AND en el Address Bus 20 fue colocada por compatibilidad con la
; 8086. En el 8086 el micro, al direccionar la posición de memorias
; superiores al mega (por ejemplo haciendo al segmento igual a 0FFFFh y
; el offset mayor a 0Fh), se direcciona nuevamente; la parte baja de
; memoria. Por ejemplo para direccionar el primer byte físico de la RAM
; se puede hacer con DS en cero y offset 0, o bien con DS en FFFFh y
; offset 10h. En los procesadores posteriores al 8086, estando en modo
; real y colocando DS en 0FFFFh y el offset en 10h, se accede al byte
; físico 100000h y no al 0h. Esto implica una incompatibilidad, por la
; cual se colocó la AND.
;
; La existencia de esta AND hace colocar en 1 la entrada de control,
; para poder acceder a los megas impares de memoria (todos los que
tengan
; en 1 el bit 20 del bus de direcciones). Para evitar complicaciones, al
; utilizar memoria superior al mega, se habilita
; la A20.
;
        ;mov al,0D1h
        ;out 64h,al
        ;mov al,0ddh
        ;out 60h,al          ;Disable A20
;
; Éste es el procedimiento para inhabilitar la A20. A pesar de que no
; se use en este ejemplo, se lo deja planteado. Se debe tener en cuenta
; que a veces programas residentes de modo real utilizan la memoria
; entre 0FFFFh:10h y 0FFFFh:0FFFFh, por eso es recomendable no modificar
; esta sección de memoria.
;
        xor eax,eax         ;pone en cero eax.
        mov ax,cs           ;carga el cs en eax.
        shl eax,4           ;en eax=base lineal de cs.
        mov ecx,eax         ;almacena en ecx la base lineal de cs.
;
; Como se indicó anteriormente, para colocar en el GDTR y en el
; descriptor 10h se debe conocer la base lineal en la que el DOS cargó
; el código. Para eso, estando todavía en modo real, se lee el CS y se
; lo multiplica por 16 (que es lo mismo que hacer 4 veces un shift a la
; izquierda). Este valor debe ser almacenado en ecx y eax.
;
; Luego, con el valor de ecx, que es la base del segmento de código, se
; puede calcular la base de la GDT para colocar en el GDTR.
;
        add ecx,gdt         ;ecx=base lineal gdt
        mov [gdtr+2],ecx    ;carga la base de GDTR
        lgdt [gdtr]         ;carga el GDTR
;
; Con las primeras dos instrucciones se completó el campo de base del
; GDTR. Una vez realizado esto se debe cargar estos 6 bytes dentro del
; registro de la PC, para lo que existe la instrucción lgdt. También
```

```
; existe su función sgtr que lee el contenido, y cuya sintaxis es
; idéntica a la de lgdt.
;
; Luego se debe colocar la base al segmento de datos, que se encuentra
; debajo del mega. Para eso se necesita memoria libre que, por ser un
; .com, se encuentra desde el final del código del programa hasta los
; 640 Kbytes.
;
; IMPORTANTE: La pila se encuentra en el segmento de código en el
; desplazamiento sp=0FFFEh, posiciones de memorias que en este ejemplo
; serán utilizadas para almacenar el BIOS, lo cual es sólo posible
porque
; no se utiliza la pila. Es importante inhabilitar las interrupciones,
; evitando así el uso involuntario.
;
; Por otro lado se debe buscar un segmento de datos vacío por debajo del
; mega de memoria. Para eso se utiliza el label fin: que indica el
; final del código.
;
        xor ebx,ebx
        mov bx,fin+15
        add eax,ebx     ;eax=base lineal fin+15
;
; En eax se tiene la base lineal del segmento de código. Se suma el
; offset de fin: y 15. Como se busca un segmento, es recomendable que
; éste se encuentre alineado en 16 bytes.
;
        shr eax,4
        mov es,ax       ;es segmento vacío de 64 Kb.
;
; Se divide por 16 y se lo carga en es.
;
        cli
        mov eax,cr0
        or al,1
        mov cr0,eax
        jmp $+2
;
; Se pasa a modo protegido.
;
        mov ax,8h
        mov ds,ax
;
; Se carga en ds el selector que apunta al descriptor de un segmento de
; datos de FLAT, o sea que tiene la base en 0 y el límite en 4Gb.
;
        mov di,0
        mov esi,0FFFF0000h
        mov cx,0ffffh
;
; Se inicializan las variables para copiar de ds:esi a es:di. No se
; utiliza la instrucción rep movsd, ya que el origen es esi (32 bits) y
; el destino es di (16 bits).
;
        repmovsb:
                mov al,[ds:esi]
                mov [es:di],al
                inc esi
                inc di
                dec cx
        jnz repmovsb
```

```
;
; Se realiza el loop, copiando así 65536 bytes.
;
        mov ax,10h
        mov ds,ax             ;se carga nuevamente un segmento de 64 Kb.
                              ;de límite.
;
; Se carga en el selector que se utilizó un descriptor de límite de
; 64Kbytes por compatibilidad con modo real.
;
        mov eax,cr0
        and al,0feh
        mov cr0,eax           ;se pasa a modo real.
        jmp $+2
;
; Se pasa a modo real y se realiza el jmp para vaciar la cola de
; instrucciones.
;
        mov ax,cs
        mov ds,ax             ;se recupera el segmento de modo real.
;
; El selector ds apuntaba a un descriptor de segmento de datos de base 0
; y límite de 64 Kbytes. Sin embargo su valor era de 10h, por lo cual,
; en modo real, se podría interpretar que su base es de 100h. Por eso es
; que se carga nuevamente con un valor de modo real, así se acomoda la
; base correctamente.
;
        mov ah,3ch
        xor cx,cx
        mov dx,archivo
        int 21h                 ;se crea el archivo.
        mov bx,ax               ;se guarda el handler.
;
; Se crea un archivo utilizando la función de BIOS. Al handler del mismo
; se lo deja en bx.
;
        mov ax,es
        mov ds,ax
        mov ah,40h
        mov cx,0ffffh
        xor dx,dx
        int 21h                 ;se graba el archivo.
;
; Se graba el archivo. Cómo el segmento que se estaba utilizando estaba
; en ds, se lo pasa a ds.
;
        mov ah,3eh              ;se cierra el archivo.
        int 21h
;
; Se cierra el archivo.
;
        sti
        mov ah,4ch
        int 21h
;
; Se habilitan las interrupciones y se vuelve al DOS.
;
fin:
;
; Éste es label que se utiliza para conocer la ubicación del final del
; código.
```

```
; ej03.asm - programa con interrupciones
;
; Protected Mode by Examples revision 0.1 - Octubre 2004
;
; Autor:
; Mariano Cerdeiro
; <m.cerdeiro@soix.com.ar>
; http://www.soix.com.ar/links/mpbyexamples.html
;
; Compilar: nasm ej03.asm -o ej03.com
;
; Este es un programa muy simple que muestra el uso de interrupciones en
; modo protegido, también muestra cómo verificar si el microprocesador
; ya se encuentra en este modo, evitando que, si lo está, el sistema
; operativo cierre el programa indicando un error. El programa
; intercepta la interrupción 8, es decir la IRQ0, la cual pertenece al
; timer. La rutina de interrupción graba, en una variable global, el
; scan-code actualmente presente en el puerto 60h (el del teclado),
; mientras el programa principal espera hasta que esta variable scan-
; code sea igual a 1 (tecla escape). Antes de comenzar a programar se
; debe conocer el funcionamiento de las interrupciones en modo
; protegido.
;
;
; Interrupciones
;
; En modo real se sabe que cuando sucede una interrupción el
; microprocesador busca, en una tabla ubicada a partir de la posición
; física 0, el vector de interrupción; cada vector ocupa 4 bytes.
; Para conocer dónde se encuentra el offset y el segmento a ejecutar en
; el caso de una interrupción se busca el vector lejano en la posición
; interrupción x 4, a partir del comienzo de la memoria, ya que cada
; vector de interrupción ocupa 4 bytes.
;
; En modo protegido el procedimiento es similar, existe una tabla
; llamada IDT (Interrupt Descriptor Table). Ésta no guarda vectores de
; interrupción, sino que, como su nombre lo indica, almacena
; descriptores. Sin embargo, a diferencia de las anteriores dos tablas
; (GDT y LDT que aún no se vieron), esta tabla tiene únicamente 256
; entradas, ya que es ésta la cantidad de interrupciones del procesador.
;
; Las entradas de la IDT pueden ser de tres tipos y únicamente del
; sistema, no se puede definir un segmento de datos en la IDT. Los
; tipos son:
;
;     - Interrupt Gate
;     - Trap Gate
;     - Task Gate
;
; La primera es una interrupción normal en la cual, para ingresar, se
; inhabilitan las interrupciones. Ésta es la única característica que
; la diferencia de la Trap Gate. Por otro lado, la Task Gate es un
; puntero a una tarea. No se han visto tareas todavía, de modo que este
; tema quedará para más adelante. Sin embargo, es bueno saber que al
; llegar una interrupción se puede ejecutar directamente otra tarea.
;
; Al igual que la GDT, la IDT tiene un registro que indica tanto su base
; como su tamaño; se denomina IDTR (Interrupt Descriptor Table
; Register) al cual se accede, como al GDTR, mediante dos instrucciones:
;     - LIDT fword ptr
;     - SIDT fword ptr
```

```
;
; La primera de ellas carga en IDTR el valor del puntero, mientras
; que SIDT lee el contenido del IDTR y lo coloca en la dirección del
; puntero.
;
use16
org 100h
comienzo:jmp inicio

modo_virtual_msg  db    'El procesador se encuentra en modo virtual, no'
                        'se puede correr la aplicación.$'
;
; El texto anterior servirá de mensaje de error cuando se ejecute el
; programa estando en modo virtual.
;
mask_real_pic1          db 0
;
; Se almacenará el valor de la máscara de interrupciones del PIC
; (Programable Interrupt Controler) en modo real, para poder restaurar
; al regresar. Se debe tener en cuenta que el programa no utilizará
; todas las interrupciones, y una forma prolija para habilitar algunas y
; otras no, es mediante el PIC.
;
real_idtr         resb 6
;
; Estos 6 bytes son para almacenar el valor del IDTR de modo real. En
; los x86 no existía este registro, sin embargo, a partir del x286, el
; mismo indica dónde se encuentra la IDT, tanto en modo protegido como
; en MODO REAL. Por eso es importante, antes de modificar el contenido
; del IDTR, guardar una copia para poder recuperar al volver a modo
; real.
;
scan_code         db     0
;
; Ésta será la variable global que almacenará el último scan-code.
;
gdtr:             dw 8*5-1
                  dd 0
;
; Puntero utilizado para cargar el GDTR. La base será calculada en
; tiempo de ejecución, ya que depende de donde cargue el SO el
; programa, mientras que el límite puede ya ser impuesto en 5
; descriptores (4, ya que el primero es el NULO y no se utiliza).
;
idtr              dw 8*10-1
                  dd 0
;
; Este puntero será el utilizado para cargar el IDTR. Al igual que con
; el GDTR la base será cargada en tiempo de ejecución, mientras que el
; límite será de 10 descriptores, ya que se necesita manejar la
; interrupción 9.
;
gdt               resb 8
;
; Se deja la primera entrada libre, ya que se trata del descriptor NULO.
;
      ;--- 08h Segmento de código de 64 Kb sobre el de modo real ---
seg_cod:dw 0ffffh       ;Límite 15:0
      dw 0              ;Base 15:0
      db 0              ;Base 23:16
      db 10011010b      ;P PL #S Tipo
```

```
      db 00000000b      ;G AVL  Límite 19:16
      db 0              ;Base 31:24
;
; Como segunda entrada de la GDT se coloca el segmento de código, que
; está superpuesto con el de modo real. El límite es de 65536. La
; base se coloca en tiempo de ejecución, de tal forma que se superponga
; con el de modo real, que es donde está el código. El segmento está
; presente, es de nivel de privilegio 0, se trata de un segmento, es de
; código y legible. La granularidad es cero.
;
; Ya se ha explicado explicado antes, pero siempre es bueno refrescar.
; Tal vez sea llamativo, éste es el PRIMER programa en que se utiliza
; segmento de código en modo protegido. ¿POR QUÉ? Al ejecutarse una
; interrupción, y al retornar de ella, se modifica tanto el CS, como el
; IP. En los ejemplos anteriores no se utilizaba CS de modo protegido,
; ya que se mantenía el de modo real. Ahora se modificará el CS
; implícitamente cuando ocurra una interrupción, por eso se debe
; utilizar un selector de modo protegido válido.
;
; Al suceder una interrupción se cargará el nuevo segmento de código y
; esto se hará de forma correcta, en la pila donde se almacena la
; dirección de retorno se guarda el CS de modo real. Cuando se ejecute
; un iret al finalizar la interrupción, se intentará cargar el CS con el
; valor del selector almacenado en la pila, que era el de modo real, sin
; embargo se cargará estando en modo protegido. Esto generará una
; excepción, ya que ese selector de la GDT o LDT no existirá.
;
      ;--- 10h Segmento de datos de 64 Kb sobre el de modo real.
seg_datos:dw 0ffffh
      dw 0
      db 0
      db 10010010b
      db 0b
      db 0
;
; En modo protegido no se puede escribir en el CS.
; O sea, la instrucción:
;
; mov [cs:si],al
;
; No se puede ejecutar. Entonces, si se desea modificar una variable
; sobre el code segment, se debe inicializar un segmento de datos
; superpuesto con el de código. Para ello se inicializa el segmento con
; todos sus campos menos la base, ya que la misma estará superpuesta con
; la del segmento de código.
       ;--- 18h Segmento de datos flat.
       dw 0ffffh       ;límite 15.00
       dw 00000h       ;base 15.00
       db 000h         ;base 23.16
       db 10010010b    ;Presente Segmento Datos Read Write.
       db 10001111b    ;G y límite 0Fh
       db 00h          ;base 31.24
;
; Este segmento flat se utiliza para agregar al programa un detalle.
; Incrementar el primer byte de la pantalla sirve únicamente para poder
; asegurar que el programa se está ejecutando correctamente.
;
       ;--- 20h Segmento de datos de 64 Kb.
       dw 0ffffh
       dw 0
       db 0
```

```
        db 10010010b
        db 0
        db 0
;
; Por último se inicializa un segmento para mantener la compatibilidad
; al retornar a modo real, como recomienda Intel.
;
idt:    resb 8*8
;
; En la tabla de interrupciones se dejan las primeras 8 interrupciones
; sin inicializar, por lo cual se deben tomar las suficientes
; precauciones para que éstas no sucedan.
;
irq0:  dw      0
       dw    08h
       db    0
       db    10000110b
       db    0
       db    0
;
; La IRQ 0 es la del timer-tick y la que se necesita para este
; ejemplo, el offset será colocado en tiempo de ejecución, ya que no
; se sabe dónde el SO colocará el código. Si se puede indicar el
; selector de código en 8. El segmento está presente, con nivel 0, se
; trata de un descriptor de sistema del tipo interrupt gate de 16 bits.
;
irq0_han:
;
; Ésta será la rutina de antención al timer-tick, la cual incrementará
el
; primer byte de la pantalla y almacenará en la variable scan code el
; valor leído desde el teclado.
;
       push ds
       pushad
;
; Lo primero que se debe hacer en el servicio de interrupción es guardar
; en la pila los registros que van a ser modificados, en este caso DS y
; los registros de propósitos generales. En realidad en este ejemplo no
; es necesario, ya que el programa principal no los utiliza, pero es una
; práctica recomendable.
;
       mov ax,18h
       mov ds,ax
       mov eax,0b8000h
       inc byte [eax]
;
; En estas 4 líneas se carga el selector del segmento FLAT en ds, se
; apunta a la memoria de video en modo texto, ubicada en 0b8000h en
; todos los monitores VGA color y se incrementa el primer byte.
;
       mov ax,10h
       mov ds,ax
;
; Luego se carga el selector del segmento imagen del de código.
;
       in al,60h
         mov [ds:scan_code],al
;
; Se lee el scan code y se lo almacena en la variable global.
;
```

```
      mov al,20h
      out 20h,al
;
; IMPORTANTE: Este paso es un error típico y genera muchos dolores de
; cabeza. El PIC nunca ejecuta una interrupción mientras esté sucediendo
; otra de mayor prioridad. Por eso es que por soft se le debe informar
; cuándo la rutina de interrupción a finalizado. Para ello se debe
; enviar un valor 20h al puerto 20h, de lo contrario nunca más se
; ejecutará una irq0 (hasta resetear).
;
      popad
      pop ds
      iret
;
; Se recuperan los registros y se retorna de la interrupción.
;
inicio:
;
; Aquí comienza el programa principal.
;
      smsw ax
;
; Lo primero es verificar si se está en modo protegido, para lo que se
; lee el bit 0 del CR0. Pero leer el CR0 en nivel 3 estando en modo
; protegido genera una excepción, por eso se utiliza la instrucción smsw
; (store machine status word), que almacena los 16 bits bajos en el
; registro destino. A modo informativo: también existe una instrucción
; lmsw que cumple la función inversa, o sea, carga los 16 bits bajos de
; CR0 con el contenido del registro origen.
;
      test al,1
;
; Se verifica el bit 0, que indica si se está en modo protegido o no.
;
      jz en_modo_real
            mov ah,09h
                mov dx,modo_virtual_msg
            int 21h
            mov ah,4ch
            int 21h
;
; Si se está en modo real, se imprime el mensaje de error y se retorna
; al DOS.
;
      en_modo_real:
      ;--- Guardo el cs de modo real
        mov [real_cs],cs
;
; En este ejemplo se utilizó un segmento de código de modo protegido.
; Por eso, al regresar a modo real, se necesitará el valor del selector
; de modo real, que se puede recuperar de varias formas. La más sencilla
; es directamente guardarlo en una variable.
;
      ;--- Se acomoda el CS
      xor eax,eax     ;eax=0
      mov ax,cs       ;almacena CS en eax.
      shl eax,4       ;en eax=base lineal de CS.
      mov ebx,eax     ;se guarda en ebx.
;
; Como se indica en la definición de los descriptores de los segmentos
```

```
; de datos y de códigos, se debe inicializar la base de los mismos en
; tiempo de ejecución. Para eso se multiplica por 16 el valor del
; segmento de código y se almacena una copia en ebx.
;
      mov [seg_cod+2],ax
      mov [seg_datos+2],ax
;
; En los bytes 2 y 3 de ambos descriptores se almacena la parte baja de
; la base.
;
      shr eax,16
      mov [seg_cod+4],al
      mov [seg_datos+4],al
;
; Después se realiza un desplazamiento para colocar la parte alta de la
; base y cargar el 3er byte. No es necesario inicializar la parte más
; alta (últimos 8 bits), ya que la posición del CS en modo real nunca
; podrá exceder 0FFFFh, lo cual, multiplicado 16 veces, puede llegar a
; ser 0FFFF0h. Nunca un bit mayor al 19 estará seteado.
;
      ;--- Se inicializa el GDTR.
      mov eax,ebx
      add eax,gdt
      mov [gdtr+2],eax
      lgdt [gdtr]       ;se carga el GDTR.
;
; En ebx se almacenó una copia de la base del segmento. Si a ésta se le
; suma el offset de la gdt, se obtiene la posición lineal de la misma
; para almacenar en el GDTR. Luego se carga el GDTR.
;
      ;--- Se inicializa la IDT.
      mov ax,irq0_han
      mov [irq0],ax
;
; En el descriptor de irq0 se debe colocar el offset. En este caso, el
; offset será respecto a la base del segmento de código que se utilizó.
; Como el mismo tiene una base igual a la de modo real, se utiliza el
; offset irq0_han directamente.
;
      ;--- Se carga la IDT.
      sidt [real_idtr]
      mov eax,ebx
      add eax,idt
      mov [idtr+2],eax
      lidt [idtr]
;
; Con el IDTR se debe realizar una operación similar a la del GDTR y,
; como en ebx, se tiene todavía la base lineal del segmento de códigos.
; A este valor se le suma el offset de la idt y se almacena el valor en
; el IDTR. Luego se carga el IDTR.
;
      ;--- A modo protegido
      cli
      mov eax,cr0
      or al,1
      mov cr0,eax
;
; Dado que éste ya es el tercer ejemplo de modo protegido, no es
; necesario explicarlo.
;
        jmp 08h:modo_protegido
```

```
;
; Aquí hay algo nuevo. Como siempre, luego de pasar a modo protegido,
; se realiza un salto. Sin embargo esta vez, como el salto es largo, la
; sintaxis es distinta. Primero se indica el selector y luego el offset.
;
      modo_protegido:
      ;--- Se carga el segmento de datos.
      mov ax,10h
      mov ds,ax
;
; Se carga el selector que apunta al descriptor del segmento de datos
; imagen del de código. Para poder acceder a las variables que están
; después del jmp inicio y antes del label inicio:
;
      ;--- Se guarda el PIC de modo real y se carga el de modo
      ;protegido.
      in al,21h
      mov [mask_real_pic1],al
      mov al,011111110b
      out 21h,al
;
; Ahora se almacena el estado de la máscara del PIC en una variable,
; para luego, antes de retornar al DOS, dejar las mismas intactas. Y se
; coloca en el PIC la nueva máscara habilitando únicamente la irq0.
;
      sti                     ;no olvidar setear las interrupciones.
      esperar:
            cmp byte [cs:scan_code],1
      jne esperar
;
; Luego se habilitan las interrupciones y se espera un scancode igual a
; 1. Si llega dicho scancode se comienza el retorno a modo real.
;
      cli

      ;--- Se carga en todos los selectores un segmento de 64 Kb de
      ; límite.
      mov ax,20h
      mov ds,ax
      mov es,ax
      mov fs,ax
      mov gs,ax

      ;--- Se retorna a modo real.
      mov eax,cr0
      and al,0feh
      mov cr0,eax             ;se pasa a modo real.
      db 0eah                 ;código del salto.
      dw modo_real
      real_cs dw 0
;
; El retorno a modo real no es mucho más complicado que en el ejemplo
; anterior. En esta ocasión, debido a que se modificó el cs al entrar en
; modo protegido, se debe realizar un salto lejano para retornar a modo
; real. El salet debe ser al offset modo_real:, o sea, a la instrucción
; siguiente. Como el compilador no puede interpretar un jmp a esta
; dirección, se lo realiza manualmente. Para ello se debe conocer el
; código de la instrucción jmp far, que es 0eah. Luego del código de la
; instrucción se debe colocar el offset y el selector. Como offset se
; coloca el de la siguiente instrucción, el label modo_real. Como
; selector se utiliza una variable que se carga en el inicio del
```

```
; programa antes de pasar a modo protegido, con la líneas: mov
; [cs:real_cs],cs.
;
      modo_real:

      ;--- Se carga el selector de modo real en todos los segmentos.
      mov ax,cs
      mov ds,ax               ;se recupera el segmento de modo real
      mov es,ax
      mov fs,ax
      mov gs,ax
;
; Una vez en modo real se recargan todos los selectores con valores
; coherentes para que su base quede en modo real.
;
      ;--- Se recuperan la idt y las máscaras.
      lidt [real_idtr]
      mov al,[mask_real_pic1]
      out 21h,al
;
; Se carga el IDTR de modo real que se guardo antes de pasar a modo real
; y se recupera la máscara del PIC de modo real.
;
        sti
        mov ah,4ch
        int 21h
;
; Por último se habilitan las interrupciones y se retorna a modo real.
;
```

```
; ej04.asm - excepciones en modo protegido
;
; Protected Mode by Examples - 1era Edición - Octubre 2004
;
; Autor:
; Mariano Cerdeiro
; <m.cerdeiro@soix.com.ar>
; http://www.soix.com.ar/links/mpbyexamples.html
;
; compilar: nasm ej04.asm -o ej04.com
;
;
; Excepciones
;
; En este ejemplo se estudiarán las excepciones. Las excepciones son
; interrupciones generadas por el procesador cuando se detecta algún
; tipo de violación. Las fuentes pueden ser:
;     - Cuando se detecta un error en el programa.
;     - Cuando el programa la genera.
;     - Por un error de chequeo del procesador.
;
; Las excepciones se clasifican en 3 tipos:
;     - Fault: Son excepciones que pueden ser corregidas. Permiten que
;       el programa continúe su ejecución. Las fault suceden ANTES de
;       que se ejecute la instrucción que la genera.
;     - Trap: Son excepciones que permiten seguir con la ejecución
;       normal del programa pero que, a diferencia de las faltas,
;       suceden luego de la ejecución de la instrucción. O sea, el ip
;       apunta a la siguiente instrucción.
;     - Abort: No se puede conocer con certeza la instrucción que las
;       generó. No hay forma de regresar a la aplicación. Se trata de un
;       error GRAVE y la aplicación que la generó debe ser cerrada.
;
; El procesador tiene reservadas las primeras 32 entradas de
; interrupciones para las excepciones, del vector 0 al 31 decimal. Las
; interrupciones del valor 32 al 255 están destinadas al usuario. Pero
; las IRQ 0 a la 7 ocupan las interrupciones de la 8 a la 15, lo cual es
; un inconveniente, ya que la excepción 8 está superpuesta con la IRQ0,
; la 9 con la IRQ1 y sucesivamente. Es por eso que, cuando ocurre una
; interrupción, como por ejemplo la 8, no se estará seguro de la
; fuente. Podría haber sido generada por una excepción de doble falta o
; por una IRQ del timer tick.
;
; Para resolver este inconveniente se modifica la base del PIC 0, de
; modo que la base del mismo se encuentre arriba de la interrupción 31,
; a partir de donde Intel dejó disponible las interrupciones para el uso
; del programador del sistema.
;
; Las excepciones son las siguientes:
;
; Vector Descripción            Tipo    Error   Fuente de la excepción
;                                       Code
; 0      Divide Error           Fault   No      Instrucciones DIV o IDIV
; 1      Debug                  Fault/  No      Instrucción Int 1 o
;                               Trap            Cualquier instrucción o
;                                               referencia a datos
; 2      NMI Interrupt          -       -       Interrupción no
;                                               enmascarable
; 3      Breakpoint             Trap    No      Instrucción INT3
; 4      Overflow               Trap    No      Instrucción INTO
; 5      BOUND                  Fault   No      Instrucción BOUND
```

```
; 6      Invalid Opcode         Fault   No      Instrucción U2D o
;                                                Instrucción desconocida
; 7      No Math CoProcessor    Fault   No      Instrucciones F___ o
;                                                instrucción WAIT.
; 8      Double Fault           Abort   Yes     Cualquier instrucción
;                                       (cero)  que pueda generar
;                                               excepciones, NMI o INTR
; 9      Coprocessor segment    Fault   No      Instrucciones de coma
;        overrun(386 only)                      flotante
; 10     Invalid TSS            Fault   Yes     Cambio de tarea o acceso
;                                               al TSS
; 11     Segment no present     Fault   Yes     Cargando un selector de
;                                               segmento o al acceder a
;                                               un descriptor de sistema
; 12     Stack Segment Fault    Fault   Yes     Operaciones de pila y al
;                                               cargar ss
; 13     General Protection     Fault   Yes     Cualquier referencia a
;                                               memoria y otros chequeos
;                                               de protección
; 14     Page fault             Fault   Yes     Cualquier referencia a
;                                               memoria
; 15     Reserved               -       -       -
; 16     FPU Error              Fault   No      FPU instructions
; 17     Alignment check        Fault   Yes     Cualquier referencia a
;                                               memoria
; 18     Machine check          Abort   No
; 19     SIMD Exception         Fault   No      Instrucciones SSE/SSE2
; 20-31  Reserved                -       -       -
; 32-255 User defined
;
; Esta lista fue copiada del manual del Pentium 4: IA-32 Intel
; Architecture Software Developer's Manual, Volumen 3: System
; Programming Guide, Capítulo 5: Interrupt and Exception Handling.
;
; Como se puede observar, hay distintos tipos de excepciones, según sea
; la causa que la genere. Además se puede notar que algunas excepciones
; tienen error code. Error code es un valor que se pushea en la pila,
; posterior a los flags, cs y eip, y que brinda información sobre lo que
; a causado la excepción. No todas las excepciones tienen código de
; error, sólo en algunas de ellas.
;
; El Código de error tiene un formato especifico para todas las
; excepciones, salvo para la Page Fault. El formato común es el
; siguiente[1]:
;
;  31                    16 15                     0
; +------------------------+------------------+-+-+-+
; |        Reservado       |     Índice del   |T|I|E|
; |                        |      Selector    |I|D|x|
; +------------------------+------------------+-+-+-+
;
; En este ejercicio se busca colgarse de algunas excepciones y generar
; una intencionalmente. La rutina de atención a las excepciones imprime
; en la pantalla el estado de todos los registros y retorna a modo real.
;
; El ejercicio además utiliza la IRQ 1 para leer una tecla y en función
; de si se trata de un ESC 1 ó 2, no generar excepción, generar una de
; protección general o una de instrucción inválida respectivamente.
```

[1] En el manual de Intel se habla de un error code de 32 bits. Sin embargo, cuando se trabaja en 16 bits, como en este ejercicio, es de 16 bits, la parte baja.

```
;
org 100h
comienzo:jmp inicio
;
; Lo primero que se debe hacer es definir una estructura donde se
; almacenarán todos los registros que luego se imprimirán en pantalla.
;
struc registros_struc
        .reg_EIP       resd 1
        .reg_EFLAGS    resd 1
        .reg_EAX       resd 1
        .reg_ECX       resd 1
        .reg_EDX       resd 1
        .reg_EBX       resd 1
        .reg_ESP       resd 1
        .reg_EBP       resd 1
        .reg_ESI       resd 1
        .reg_EDI       resd 1
        .reg_ES        resw 1
        .reg_CS        resw 1
        .reg_SS        resw 1
        .reg_DS        resw 1
        .reg_FS        resw 1
        .reg_GS        resw 1
endstruc
;
struc excepciones_struc
      .excepcion  resd 1
      .errorcode? resw 1
      .errorcode  resw 1
      .registros  resb registros_struc_size
endstruc
;
; Se declara la variable excepciones del tipo excepciones_struc.
;
excepciones resb  excepciones_struc_size
;
; Se colocan los mensajes que se deben imprimir.
;
eax_msg           db    'EAX=',0
ebx_msg           db    'EBX=',0
ecx_msg           db    'ECX=',0
edx_msg           db    'EDX=',0
esi_msg           db    'ESI=',0
edi_msg           db    'EDI=',0
esp_msg           db    'ESP=',0
ebp_msg           db    'EBP=',0
cs_msg            db    ' CS=',0
ds_msg            db    ' DS=',0
es_msg            db    ' ES=',0
fs_msg            db    ' FS=',0
gs_msg            db    ' GS=',0
ss_msg            db    ' SS=',0
eip_msg           db    'EIP=',0
eflags_msg      db      'EFLAGS=',0
errorcode_msg     db    'ERRORCODE=',0
excepcion_msg     db    'EXCEPCIÓN=',0
;
; La que sigue es la función utilizada para mostrar los registros.
; Esta función llama a otras 3, clrscr, que borra la pantalla,
; mostrar_cadena, que imprime una cadena de caracteres hasta encontrar
```

```
; un NULL y mostrar_valor, que imprime un eax en hexadecimal.
;
mostrar_reg:
;
; Claramente se borra la pantalla, más adelante está la función.
;
      call clrscr
;
; Se guarda el estado de los registros y el DS. En el DS se carga el
; segmento solapado con el código.
;
      push ds
      pushad
      mov ax,10h
      mov ds,ax
;
; Se imprime en pantalla la cadena de EIP, vale aclarar que ds:esi es la
; posición de la cadena a imprimir, la cual debe finalizar con NULL y
; que la posición en la pantalla es expresada en edi.
;
      mov esi,eip_msg
      mov edi,2+80*2
      call mostrar_cadena
;
; Luego se carga el valor de eip en eax y se lo muestra en la posición
; indicada por edi.
;
        mov edi,10+80*2
        mov eax, [excepciones\
            +excepciones_struc.registros+registros_struc.reg_EIP]
        call mostrar_valor
;
; La forma de direccionar las estructuras es algo complejo en nasm. Las
; estructuras se manejan como desplazamientos, de modo que se deben
; realizar simples sumas de desplazamientos para indicar al procesador
; dónde se encuentra el valor del EIP, registro que se desea imprimir.
; "excepciones" indica el offset dentro del segmento de datos donde
; comienza la estructura. Luego, "excepciones_struc.registros" indica
; dónde comienza la estructura de registros dentro del la estructura
; excepciones. Por último "registros_struc.reg_EIP" indica dónde se
; encuentra el EIP, dentro de la estructura registros, dentro de la
; estructura excepciones.
;
; Se imprime la excepción y su mensaje correspondiente.
;

      mov esi,excepcion_msg
      mov edi,70
      call mostrar_cadena
      mov edi,90
      mov eax,[excepciones+excepciones_struc.excepcion]
      call mostrar_valor
;
; Se imprime EAX.
;
        mov esi,eax_msg
        mov edi,2+160*2
        call mostrar_cadena
        mov edi,2+160*2+8
        mov eax,[excepciones+excepciones_struc.registros+\
            registros_struc.reg_EAX]
```

```
        call mostrar_valor
;
; Se imprime EBX.
;
        mov esi,ebx_msg
        mov edi,2+160*2+30
        call mostrar_cadena
        mov edi,2+160*2+8+30
        mov eax,[excepciones+excepciones_struc.registros+\
            registros_struc.reg_EBX]
        call mostrar_valor
;
; Se imprime ECX.
;
        mov esi,ecx_msg
        mov edi,2+160*2+60
        call mostrar_cadena
        mov edi,2+160*2+8+60
        mov eax,[excepciones+excepciones_struc.registros+\
            registros_struc.reg_ECX]
        call mostrar_valor
;
; Se imprime EDX.
;
        mov esi,edx_msg
        mov edi,2+160*2+90
        call mostrar_cadena
        mov edi,2+160*2+8+90
        mov eax,[excepciones+excepciones_struc.registros+\
            registros_struc.reg_EDX]
        call mostrar_valor
;
; Se imprime ESI.
;
        mov esi,esi_msg
        mov edi,2+160*3
        call mostrar_cadena
        mov edi,2+160*3+8
        mov eax,[excepciones+excepciones_struc.registros+\
            registros_struc.reg_ESI]
        call mostrar_valor
;
; Se imprime EDI.
;
        mov esi,edi_msg
        mov edi,2+160*3+30
        call mostrar_cadena
        mov edi,2+160*3+8+30
        mov eax,[excepciones+excepciones_struc.registros+\
            registros_struc.reg_EDI]
        call mostrar_valor
;
; Se imprime ESP.
;
        mov esi,esp_msg
        mov edi,2+160*3+60
        call mostrar_cadena
        mov edi,2+160*3+8+60
        mov eax,[excepciones+excepciones_struc.registros+\
            registros_struc.reg_ESP]
        call mostrar_valor
```

```
;
; Se imprime EBP.
;
        mov esi,ebp_msg
        mov edi,2+160*3+90
        call mostrar_cadena
        mov edi,2+160*3+8+90
        mov eax,[excepciones+excepciones_struc.registros+\
            registros_struc.reg_EBP]
        call mostrar_valor
;
; Se imprime CS.
;
        mov edi,160*4
        mov eax,[excepciones+excepciones_struc.registros+\
            registros_struc.reg_CS]
        call mostrar_valor
        mov esi,cs_msg
        mov edi,160*4
        call mostrar_cadena
;
; Se imprime DS.
;

        mov edi,160*4+20
        mov eax,[excepciones+excepciones_struc.registros+\
            registros_struc.reg_DS]
        call mostrar_valor
        mov esi,ds_msg
        mov edi,160*4+20
        call mostrar_cadena
;
; Se imprime ES.
;
        mov edi,160*4+40
        mov eax,[excepciones+excepciones_struc.registros+\
            registros_struc.reg_ES]
        call mostrar_valor
        mov esi,es_msg
        mov edi,160*4+40
        call mostrar_cadena
;
; Se imprime FS.
;
        mov edi,160*4+60
        mov eax,[excepciones+excepciones_struc.registros+\
            registros_struc.reg_FS]
        call mostrar_valor
        mov esi,fs_msg
        mov edi,160*4+60
        call mostrar_cadena
;
; Se imprime GS.
;
        mov edi,160*4+80
        mov eax,[excepciones+excepciones_struc.registros+\
            registros_struc.reg_GS]
        call mostrar_valor
        mov esi,gs_msg
        mov edi,160*4+80
        call mostrar_cadena
```

```
;
; Se imprime SS.
;
        mov edi,160*4+100
        mov eax,[excepciones+excepciones_struc.registros+\
            registros_struc.reg_SS]
        call mostrar_valor
        mov esi,ss_msg
        mov edi,160*4+100
        call mostrar_cadena
;
; Se imprimen los EFLAGS.
;
        mov edi,160*5+2+14
        mov eax,[excepciones+excepciones_struc.registros+\
            registros_struc.reg_EFLAGS]
        call mostrar_valor
        mov esi,eflags_msg
        mov edi,160*5+2
        call mostrar_cadena
;
; Por último se imprime el error code, en caso de existir alguno.
;
        cmp byte [excepciones+excepciones_struc.errorcode?],1
        jne no_errorcode
                mov edi,160*6+2+12
                mov eax,[excepciones+excepciones_struc.errorcode]
                call mostrar_valor
                mov esi,errorcode_msg
                mov edi,160*6+2
                call mostrar_cadena
      no_errorcode:
;
; Se recuperan los valores de los registros y se retorna.
;
      popad
      pop ds
ret
;
; Esta función es la encargada de mostrar una cadena en la pantalla,
; para ello debe recibir en ds:esi la cadena finalizada en NULL, y en
; edi la posición dentro de la pantalla.
;
mostrar_cadena:
;
; Antes que nada se guardan todos los registros que se van a utilizar y
; se carga el segmento necesario, que en este caso es el flat, para
; poder acceder a la memoria de video.
;
      pushad
      push es
      mov bx,18h
      mov es,bx
;
; Se verifica que la dirección a imprimir sea par y menor a 4096. Luego
; se suma el desplazamiento, para que apunte a la memoria de video.
;
      and edi,0FFEh
      add edi,0b8000h
;
; Se hará un loop para imprimir en la pantalla. Se leen los caracteres
```

```
; de ds:esi mediante la instrucción lodsb y se verifica que el valor sea
; distinto de 0. Luego se lo coloca en es:edi, se incrementa edi y se
; continúa con el siguiente byte.
;
      mov ah,07h
      cadena:
            mov al,[esi]
            cmp al,0
            je salir_cadena
            mov [es:edi],ax
            inc edi
            inc edi
            inc esi
      jmp cadena
      salir_cadena:
;
; Se recuperan los registros y se retorna.
;
      pop es
      popad
ret
;
; La rutina clrscr no merece demasiados comentarios. Completa la
; pantalla con espacios y con atributos de color de letras blancas y
; fondo negro. Esta rutina no recibe ningún parámetro.
;
clrscr:
      push ds
      mov ax,18h
      mov ds,ax

      mov edi,0b8000h
      mov cx,80*25*2/4
      mov eax,07000700h
      clrscr1:
            mov [edi],eax
            add edi,4
            dec cx
      jnz clrscr1
      pop ds
ret
;
; La siguiente función imprime el valor de eax en la pantalla en la
; posición indicada por edi.
;
mostrar_valor:
;
; Primero se guardan los registros que se van a utilizar. En ds se
; carga el segmento flat para poder acceder a la memoria de video.
;
      push ds
      pushad
      mov bx,18h
      mov ds,bx
;
; Se verifica que la dirección a imprimir sea par y menor a 4096. Luego
; se suma el desplazamiento, para que apunte a la memoria de video.
;
      and edi,0FFEh
      add edi,0b8000h
;
```

```
; Y se arma una rutina que escribe el número contenido en eax en formato
; hexadecimal. Para ello se separa el número en nibles (grupos de 4
; bits). A cada nible se le suma 0 ascii y si es mayor a 9 se le suma la
; diferencia entre 9+1 y A ascii, de modo que, de ser 10 el valor del
; dígito, se vea una A.
;
      mov cx,8
      correr:
            rol eax,4
            mov byte [edi],'0'
            mov bl,al
            and bl,0fh
            add [edi],bl
            cmp byte [edi],':'
            jb ok
                  add byte [edi],'A'-':'
            ok:
            inc edi
            inc edi
            dec cx
      jnz correr
;
; Se recuperan las variables y se retorna.
;
      popad
      pop ds
ret
;
; Esta rutina es la que se ocupa de cargar los registros, que estarán
; almacenados en la pila en el orden según el cual fueron pusheados.
; Vale aclarar que el código es de 16 bits, por eso el tamaño mínimo de
; datos a pushear es de 2 bytes. Por el contrario, de estar trabajando
; con pilas de 32 bits, al realizar un push ds u otro selector, el push
; ocupa 4 bytes en la pila, dos del ds y dos que deja libres para
; mantener la pila alineada en 4 bytes.
;
cargar_reg:
      mov [excepciones+excepciones_struc.registros+\
    registros_struc.reg_EAX],eax
      mov [excepciones+excepciones_struc.registros+\
    registros_struc.reg_EBX],ebx
      mov [excepciones+excepciones_struc.registros+\
    registros_struc.reg_ECX],ecx
      mov [excepciones+excepciones_struc.registros+\
    registros_struc.reg_EDX],edx
      mov [excepciones+excepciones_struc.registros+\
    registros_struc.reg_ESI],esi
      mov [excepciones+excepciones_struc.registros+\
    registros_struc.reg_EDI],edi
      mov [excepciones+excepciones_struc.registros+\
    registros_struc.reg_EBP],ebp
      mov [excepciones+excepciones_struc.registros+\
    registros_struc.reg_ESP],esp
      mov ax,[esp+2]
      mov [excepciones+excepciones_struc.registros+\
    registros_struc.reg_EIP],ax
      mov ax,[esp+2+2]
      mov [excepciones+excepciones_struc.registros+\
    registros_struc.reg_CS],ax
      mov [excepciones+excepciones_struc.registros+\
    registros_struc.reg_DS],ds
```

```
      mov [excepciones+excepciones_struc.registros+\
    registros_struc.reg_ES],es
      mov [excepciones+excepciones_struc.registros+\
    registros_struc.reg_FS],fs
      mov [excepciones+excepciones_struc.registros+\
    registros_struc.reg_GS],gs
      mov [excepciones+excepciones_struc.registros+\
    registros_struc.reg_SS],ss
      mov eax,[esp+2+2+2]
      mov [excepciones+excepciones_struc.registros+\
    registros_struc.reg_EFLAGS],eax
ret
;
; Ahora quedan por definir las variables del sistema.
;
modo_virtual_msg  db 'El procesador se encuentra en modo virtual, no'
                  db 'se puede correr la aplicacion.$'
mask_real_pic1          db 0
real_idtr         resb 6
;
; Las primeras 3 serán: el mensaje para indicar que el micro ya se
; encuentra en modo protegido, la variable para almacenar las máscara
; del PIC y el IDTR de modo real.
;
scan_code         db     0
;
; scan_code es la variable global donde la rutina de atención de la irq0
; almacena el scancode leído del teclado, mientras el programa principal
; verifica si es igual a 1 (la tecla escape).
;
gdtr        dw 8*5-1
                  dd 0
;
idtr              dw 8*34-1   ;0,1,2,3,4,5,6,7,8,9
                  dd 0
;
gdt               resb 8
;
; Se reservan 8 bytes para el descriptor nulo.
;
; Primero como descriptor 08h se realiza un segmento de código de
; 64 Kbytes sobre el de modo real.
;
seg_cod:dw 0ffffh
      dw 0
      db 0
      db 10011010b
      db 00000000b
      db 0
;
; Como ya se realizaron algunas veces y debido a que no se puede
; escribir sobre el primer segmento de código definido, de define un
; segmento igual, pero de datos, será el 10h.
;
seg_datos:dw 0ffffh
      dw 0
      db 0
      db 10010010b
      db 0b
      db 0
;
```

```
; Por último se utilizarán dos descriptores más. Uno FLAT, para acceder,
; por ejemplo, a la memoria de video, y realizar las impresiones en
; pantalla. Otro de 64 Kbytes, para colocar antes de retornar a modo
; real. Sus valores son:
;
        dw 0ffffh       ;limite 15.00
        dw 00000h       ;base 15.00
        db 000h         ;base 23.16
        db 10010010b    ;Presente Segmento Datos Read Write
        db 10001111b    ;G y limite 0Fh
        db 00h          ;base 31.24

        dw 0ffffh
        dw 0
        db 0
        db 10010010b
        db 0
        db 0
;
; Luego se define la IDT, en la que se deben indicar todas las
; excepciones e interrupciones que se desean recibir. En este caso se
; usan dos puertas de interrupciones, pero en otros es recomendable usar
; puertas de tarea, por ejemplo en una doble falta, o en la stack fault.
; Esto se debe a que a veces es necesario cambiar de entorno de tarea
; para no seguir generando la excepción. Por ejemplo: si se acaba la
; pila estando en una rutina de nivel 0 y se desea ejecutar una stack
; fault, se necesita pila para colocar las direcciones de retorno. En
; este caso será mejor utilizar tareas alternativas.
;
idt:
      resb  8*6
;
; Invalid Opcode exception handler.
;
int6: dw    int6han
      dw    08h
      db    0
      db    10000110b
      db    0
      db    0
;
; Se reserva espacio para las interrupciones 7,8,9,10,11,12 que no
; se utilizan.
;
      resb 8*6
;
; General Protection exception handler.
;
int13:      dw    int13han
      dw    08h
      db    0
      db    10000110b
      db    0
      db    0
;
; Se reserva espacio para las interrupciones 14, 15, 16, 17, 18, 19, 20,
; 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32.
;
      resb  8*19
irq1: dw    irq1han
      dw    08h
```

```
      db    0
      db    10000110b
      db    0
      db    0
;
; La rutina de irq0 handler se ocupa de incrementar el primer byte de la
; pantalla y leer el último scancode enviado por el teclado.
;
irq1han:
      push ds
      push eax

      mov ax,10h
      mov ds,ax
;
; En la interrupción se lee el scancode del puerto 60h y se lo almacena
; en la variable correspondiente.
;
      in al,60h
      mov [ds:scan_code],al

      mov al,20h
      out 20h,al
;
; No deben olvidarse las 2 líneas anteriores, que son las que indican al
; PIC que la interrupción ha terminado. De lo contrario, no ejecutará
; nunca más una interrupción, ya que no habrá otra de mayor prioridad
; (ésta es la 0, que es la de mayor prioridad).
;
      pop eax
      pop ds
iret
;
; Ambas excepciones hacen lo mismo. Se necesita un método para
; distinguir cuál de ellas ha sucedido, por lo que se hace una rutina de
; 3 líneas por excepción donde se almacenan los registros en la pila y
; se coloca en eax el número de la excepción. Luego se salta, claro, a
; la rutina principal de excepción, que es común a ambas.
;
int6han:
      mov  dword[excepciones+excepciones_struc.excepcion],6
      jmp atender_excepcion
int13han:
      push eax
        mov word [excepciones+excepciones_struc.errorcode?],1
        mov ax,[esp+1*4]
        mov [excepciones+excepciones_struc.errorcode],ax
        pop eax
        add esp,2
        mov dword [excepciones+excepciones_struc.excepcion],13
      jmp atender_excepcion
;
; La rutina de excepción guarda el estado del procesador, lo muestra en
; pantalla y retorna a modo real.
;
atender_excepcion:
      call cargar_reg
      call mostrar_reg
      jmp volver_a_modo_real
;
; Se hace una rutina (macro) para mover ambos PICs, que recibe como
```

```
; parámetros la base de ambos PIC. Para modificar la base del PIC
; se deben enviar las 4 palabras de configuración (queda al lector
; estudiar qué hace cada una de ellas).
;
%macro picmove 2
        pushfd
        cli
        mov al,11h
        out 20h,al              ;ICW1
        mov al,%1
        out 21h,al              ;ICW2
        mov al,04h
        out 21h,al              ;ICW3
        mov al,01h
        out 21h,al              ;ICW4

        mov al,11h
        out 0a0h,al             ;ICW1
        mov al,%2
        out 0a1h,al             ;ICW2
        mov al,02h
        out 0a1h,al             ;ICW3
        mov al,1
        out 0a1h,al             ;ICW4
        popfd
%endmacro
;
; De aquí en más se trata de un programa típico de modo protegido, con
; alguna salvedad, por ejemplo la generación de la excepción, pero ya
; se llegará a eso.
;
inicio:
;
; Se verifica si se está en modo real. En caso contrario se imprime el
; mensaje y se retorna.
;
      smsw ax
      test al,1
      jz en_modo_real
            mov ah,09h
            mov dx,modo_virtual_msg
            int 21h
            mov ah,4ch
            int 21h
      en_modo_real:
;
; Ya que se cambiará de segmento de código al pasar a modo protegido,
; para poder retornar se debe almacenar el de modo real. Por ello es que
; se almacena el segmento de código de modo real en la variable real_cs.
;
      mov ax,cs
      mov [real_cs],ax
;
; Se carga la base del code segment, como también del segmento de datos.
;
      xor eax,eax                   ;carga eax en cero
      mov ax,cs                     ;carga el cs en eax
      shl eax,4                     ;en eax=base lineal de cs
      mov ebx,eax                   ;se guarda en ebx
      mov word [seg_cod+2],ax
      mov word [seg_datos+2],ax
```

```
      shr eax,16
      mov byte [seg_cod+4],al
      mov byte [seg_datos+4],al
;
; Se carga la base del GDTR.
;
      mov eax,ebx
      add eax,gdt
      mov dword [gdtr+2],eax
        lgdt [gdtr];cargo el GDTR
;
; Se guarda el IDTR de modo real en una variable y se carga el nuevo
; IDTR, no sin antes cargar la base del mismo.
;
      sidt [real_idtr]
      mov eax,ebx
      add eax,idt
      mov dword [idtr+2],eax
      lidt [idtr]
;
; Luego se pasa a modo protegido.
;
        cli
        mov eax,cr0
        or al,1
        mov cr0,eax
      jmp 08:modo_protegido
;
; A partir de aquí el procesador corre en modo protegido.
;
      modo_protegido:
;
; Se carga el segmento de datos.
;
      mov ax,10h
      mov ds,ax
;
; Y se inhabilita el uso del resto de los selectores, colocándolos en
; 0, salvo la pila, porque se utilizará la de modo real.
;
      mov ax,0
      mov es,ax
      mov fs,ax
      mov gs,ax
;
; Se modificará la base de los PICs a 32 y 40.
;
      picmove 32,40
;
; Se guarda el valor de la máscara del PIC de modo real y se inicializa
; la máscara a utilizar en este programa, la cual permite únicamente la
; irq0. Luego se habilitan las interrupciones.
;
      in al,21h
      mov [mask_real_pic1],al
      mov al,011111101b
      out 21h,al
;
; Se habilitan las interrupciones, en este caso la del teclado
; únicamente.
;
```

```
      sti
;
; Se espera que el scancode sea igual a 1,2,3, o sea, que se presione la
; tecla escape, 1 ó 2.
;
      esperar:
            cmp byte [cs:scan_code],1
            je volver_a_modo_real
;
; Si la tecla presionada es el escape, se sale a modo real directamente.
;
            cmp byte [cs:scan_code],2
            je write_cs
;
; Si la tecla presionada es el 1, se intenta escribir en el cs,
; generando una excepción de protección general.
;
            cmp byte [cs:scan_code],3
            je invalidopcode
;
; Si se presiona la tecla 2, se intenta ejecutar una instrucción
; inválida generando una excepción de instrucción inválida, de lo
; contrario, se sigue esperando una tecla.
;
      jmp esperar
;
; Se intenta escribir un 0 en el offset de scancode del cs.
;
write_cs:
      mov byte [cs:scan_code],0
;
; Para ejecutar una instrucción inexistente, se puede armar una que no
; esté, o bien ejecutar UD2 (Intel define que nunca lo utilizará como
; instrucción).
;
invalidopcode:
      ud2
;
; A la siguiente instrucción se llegará por el salto que se realiza al
; terminar el análisis de la excepción. Para demostrar esto, se puede
; introducir un cli y un jmp $ y verificar que nunca se ejecutan, ya que
; se bloquearía el sistema.
;
      cli
      jmp $

      volver_a_modo_real:
      cli
;
; Se cargan todos los selectores con el valor recomendado por Intel.
;
      mov ax,20h
      mov ds,ax
      mov es,ax
      mov fs,ax
      mov gs,ax
;
; Se retorna modo real.
;
        mov eax,cr0
        and al,0feh
```

```
        mov cr0,eax           ;se vuelve a modo real
;
; En este caso se realiza el jmp mediante su opcode, ya que se trata de
; saltar a una variable.
;
      db 0eah                 ;código del salto
      dw modo_real
      real_cs resw 1

      modo_real:
;
; Se recuperan todos los selectores a un valor válido.
;
      mov ax,cs
      mov ds,ax               ;se recupera el segmento de modo real
      mov es,ax
      mov fs,ax
      mov gs,ax
;
; Se recupera el viejo IDTR y la vieja máscara del PIC.
;
      lidt [real_idtr]
      mov al,[mask_real_pic1]
      out 21h,al
;
; Se recupera la base de los PICs, como se encontraba en modo real.
;
      picmove 8,70h
;
; Se habilitan las interrupciones y se retorna a modo real.
;
        sti
        mov ah,4ch
        int 21h
fin:
```

```
; ej05.asm - programa multitarea
;
; Protected Mode by Examples - 1era Edición - Octubre 2004
;
; Autor:
; Mariano Cerdeiro
; <m.cerdeiro@soix.com.ar>
; http://www.soix.com.ar/links/mpbyexamples.html
;
; compilar: nasm ej05.asm -o ej05.com
;
; Este ejemplo tiene por finalidad ilustrar la ejecución de dos tareas
; en forma concurrente. El programa salta de una tarea a la otra cada
; vez que hay una interrupción del timertick, hasta que se pulse la
; tecla escape, caso en que retornará a modo real. Las tareas imprimen,
; cada una, un contador ascendente o descendente en distintas partes
; del monitor.
;
;
; Multitarea
;
; Lo primero a saber respecto de la multitarea es que un procesador
; siempre ejecuta una ÚNICA tarea a la vez. La multitarea implica
; ejecutar varias tareas, de a una por vez, seguidas en el tiempo
; por intervalos tan pequeños que aparentan, para el usuario, correr
; simultáneamente. La capacidad de correr varias tareas en forma
; realmente simultánea se denomina multiprocesamiento, para lo que se
; necesitan varios procesadores o un procesador con esa capacidad, como
; por ejemplo los HT (Hyper Threading), que en realidad internamente
; funciona como 2.
;
; A diferencia de la protección, a la multitarea se la puede realizar
; por soft. El procesador provee un método por hardware para realizar
; multitarea. Sin embargo, se pueden correr muchas tareas sin utilizar
; las facilidades que provee el procesador, ya que se puede pasar de
; ejecutar la aplicación X a ejecutar la aplicación Y, almacenando el
; estado actual de todos los registros de la aplicación X y cargando
; los almacenados anteriormente de la aplicación Y. Ejemplo de este
; procedimiento son, para quienes tienen memoria, los programas
; residentes de DOS. Estos programas corrían mientras uno realizaba otra
; tarea. Hoy en día muchos sistemas operativos no utilizan las
; facilidades de multitarea de los procesadores, sino que la realizan
; manualmente, como por ejemplo Linux.
;
; En este ejemplo se estudia cómo realizar multitarea mediante el
; procesador.
;
; Para almacenar el estado de cada una de las tareas, el procesador
; utiliza una estructura denominada TSS (Task State Segment). Al saltar
; de una tarea a otra, se guarda automáticamente el estado actual del
; procesador en el TSS actual y se carga el nuevo estado del procesador
; del TSS de destino.
;
; Cada TSS almacena el estado de los registros del procesador de cada
; tarea, por ejemplo eax, ebx, eip, para que cuando se desee retornar a
; dicha tarea se pueda continuar ejecutando el programa del mismo lugar
; donde fue dejado.
;
;
;
;
```

```
;
;
;
; La estructura del TSS de 32 bits es la siguiente:
;
;  31              16 15                    0
; +------------------------+-----------------------+-+
; |  I/O Map Base Address  |        Reserved       |T|100
; +------------------------+-----------------------+-+
; |        Reserved        |  LDT Segment Selector | 96
; +------------------------+-----------------------+
; |        Reserved        |           GS          | 92
; +------------------------+-----------------------+
; |        Reserved        |           FS          | 88
; +------------------------+-----------------------+
; |        Reserved        |           DS          | 84
; +------------------------+-----------------------+
; |        Reserved        |           SS          | 80
; +------------------------+-----------------------+
; |        Reserved        |           CS          | 76
; +------------------------+-----------------------+
; |        Reserved        |           ES          | 72
; +------------------------+-----------------------+
; |                       EDI                      | 68
; +------------------------------------------------+
; |                       ESI                      | 64
; +------------------------------------------------+
; |                       EBP                      | 60
; +------------------------------------------------+
; |                       ESP                      | 56
; +------------------------------------------------+
; |                       EBX                      | 52
; +------------------------------------------------+
; |                       EDX                      | 48
; +------------------------------------------------+
; |                       ECX                      | 44
; +------------------------------------------------+
; |                       EAX                      | 40
; +------------------------------------------------+
; |                    EFLAGS                      | 36
; +------------------------------------------------+
; |                       EIP                      | 32
; +------------------------------------------------+
; |                       CR3                      | 28
; +------------------------+-----------------------+
; |        Reserved        |          SS2          | 24
; +------------------------+-----------------------+
; |                      ESP2                      | 20
; +------------------------+-----------------------+
; |        Reserved        |          SS1          | 16
; +------------------------+-----------------------+
; |                      ESP1                      | 12
; +------------------------+-----------------------+
; |        Reserved        |          SS0          |  8
; +------------------------+-----------------------+
; |                      ESP0                      |  4
; +------------------------+-----------------------+
; |        Reserved        |   Previous Task Link  |  0
; +------------------------+-----------------------+
;  31                    16 15                    0
;
```

```
;
; En el 80286 el TSS es de 16 bits y por compatibilidad se puede
; utilizar en todos los procesadores actuales. El TSS de 16 bits se
; puede encontrar en los manuales de Intel, pero no es recomendable
; usarlo, ya que muy comúnmente se usa la parte alta de los registros y
; éstos no son almacenados en el TSS de 16 bits.
;
; El Task State Segment es toda la información que almacena el
; procesador de cada tarea. Es probable que en el caso de tratarse de un
; sistema operativo, el mismo desee almacenar algunos datos extras
; además de esta información. Por ejemplo, los sistemas operativos
; almacenan el PID (Process ID), el PPID (Paretn Process ID) y muchos
; otros datos de importancia, según la estructura que tenga el sistema
; operativo y cómo esté diseñado.
;
; Los campos mínimos que utiliza el TSS (que son necesarios por el
; procesador) son los siguientes:
;
;     - Previous Task Link: contiene el selector de la tarea por la que
;       fue llamada, en caso de haber saltado a ésta con call, mediante
;       una interrupción o excepción.
;     - SS0:ESP0,SS1:ESP1,SS2:ESP2: almacena las pilas de los
;       distintos niveles de privilegios (0, 1 y 2). Todavía no se han
;       estudiado niveles de privilegio, sin embargo se puede adelantar
;       que los niveles son 4 y que la idea es separar las aplicaciones
;       del kernel, de modo que, si una aplicación falla el sistema
;       operativo podrá seguir funcionando correctamente. Basta con
;       reiniciar la tarea que falló. Siendo la pila el espacio donde se
;       almacenan datos importantes, como las variables locales, es
;       importante separar la pila de cada nivel de privilegio, evitando
;       que una aplicación, por error o por mala intención, tenga acceso
;       a la pila de otro nivel.
;     - CR3: tabla de entradas de la paginación. Indica la posición
;       física de la PDE (Page Directory Entry).
;     - EIP: desplazamiento de la próxima instrucción a ser ejecutada
;       por el procesador en esta tarea.
;     - EFLAGS: EFLAGS.
;     - EAX,ECX,EDX,EBX,ESP,EBP,ESI,EDI: almacena los 8 registros de
;       propósitos generales.
;     - ES,CS,SS,DS,FS,GS: almacena los 6 selectores.
;     - LDT Segment Selector: almacena el selector de la LDT.
;     - T: Trap Bit, genera una excepción de debug al saltar a una tarea
;       con este bit seteado.
;     - I/O Map Base Address: Indica a partir de qué offset se encuentra
;       el mapa de bits de entrada/salida. Sirve para dar a una tarea
;       acceso únicamente a ciertos puertos. El IO Map debe finalizar
;       con un byte en 0ffh que debe estar fuera del límite del TSS.
;
; Como se deduce de su nombre, el TSS es un segmento donde se almacena
; el estado de una determinada tarea. Los TSS se direccionan a través de
; un descriptor de tipo TSS-32bits.
;
; Repasando el ejemplo 2 se puede ver que existen dos tipos de
; descriptores de TSS de 32 bits:
;
;       1       0       0       1       32-Bit TSS (no ocupada)
;       1       0       1       1       32-Bit TSS (ocupada)
;
; La diferencia es que uno está ocupado y el otro no. El que esté
; ocupado implica que la tarea a la cual hace referencia está en
; ejecución. Dado que las tareas NO SON REENTRANTES, si se trata de
```

```
; saltar o llamar a una tarea que se encuentra ocupada, se generará una
; excepción. Las tareas que no se encuentran ocupadas no se encuentran
; en ejecución.
;
; El descriptor de TSS es:
;
;  31                    16 15                     0
; +----------+-+-+-+-+-----+-+-+-+-+----+-----------+
; |   Base   |G|0|0|A|Limit|P|D P|0|10B1|   Base    | 4
; |   31:23  | | | |V|19:16| | L | |    |   22:16   |
; +----------+-+-+-+-+-----+-+-+-+-+----+-----------+
;
; +------------------------+------------------------+
; |       Base 15:0        |      Limit 15:0        | 0
; |                        |                        |
; +------------------------+------------------------+
;  31                    16 15                     0
;
; Aquí se observa el bit busy, que indica si la tarea está o no
; ocupada.
;
; El procesador cuenta con un registro TR (Task Register), donde se
; almacena el selector que apunta al descriptor del TSS de la tarea que
; se está corriendo en cada momento. El TR puede ser direccionado
; mediante las instrucciones str y ltr, las cuales leen y cargan el TR
; respectivamente. Sin embargo, cabe aclarar que, aunque ltr carga el
; TR, no carga los valores indicados en el TSS en los registros del
; procesador, o sea, no realiza un salto de tarea, sino que su finalidad
; es la siguiente:
;
;   Al realizar un salto de tarea, el procesador almacena su estado en
;   el TSS actual y carga en sus registros los valores indicados por el
;   TSS de destino. El problema surge al realizar el primer salto de
;   tareas, ya que NO EXISTE TSS actual. Es por eso que existe la
;   instrucción TR, que indica al procesador el TSS actual. De esta
;   forma, llegado el caso de un salto de tareas, el estado del micro
;   podrá ser almacenado donde indique el descriptor apuntado por el TR.
;   De no cargar el TR antes de realizar un salto de tareas, se generará
;   una excepción, ya que el procesador no podrá almacenar su estado
;   actual en ningún TSS.
;
; Los descriptores del tipo TSS únicamente se pueden encontrar en la
; GDT. Si se desea colocar un acceso a una tarea en la LDT o en la IDT,
; se debe utilizar un Task Gate:
;
;  31                    16 15                     0
; +------------------------+-+-+-+-+----+-----------+
; |        Reserved        |P|D P|0|0101| Reserved  | 4
; |                        | | L | |    |           |
; +------------------------+-+-+-+-+----+-----------+
;
; +------------------------+------------------------+
; |       TSS Selector     |        Reserved        | 0
; |                        |                        |
; +------------------------+------------------------+
;  31                    16 15                     0
;
; Un descriptor de Task Gate se puede encontrar también en la GDT.
;
;
use16
```

```
org 100h
;
jmp inicio
;
gdtr    dw 7*8-1
        dd 0
;
;
gdt     resb 8
;
; 08h Segmento de código de 64 Kb sobre el de modo real.
;
%define code_sel        08h
seg_code:dw 0ffffh
      dw 0
      db 0
      db 10011010b
        db 0
      db 0
;
; 10h Segmento de datos de 64 Kb sobre el de modo real.
;
%define data_sel        10h
seg_data:dw 0ffffh
      dw 0
      db 0
      db 10010010b
      db 0b
      db 0
;
; 18h Segmento de datos flat.
;
%define flat_sel        18h
seg_flat:dw 0ffffh      ;límite 15.00
        dw 00000h       ;base 15.00
        db 000h         ;base 23.16
        db 10010010b    ;Presente Segmento Datos Read Write
        db 10001111b    ;G y limite 0Fh
        db 00h          ;base 31.24
;
; Aquí comienzan las diferencias con los programas anteriores. Debido a
; que se desea hacer un programa con 2 aplicaciones, se necesitan 2 TSS.
; Además se necesita un selector de TSS cargado en TR antes de realizar
; el primer salto de tarea, por lo que se definen 3 TSS. Uno no se usará
; más que para realizar el primer salto y se lo denominará TSS_inicial.
; Los otros dos serán los TSS de las 2 aplicaciones, TSS superior e
; inferior, según en qué parte de la pantalla muestren, las respectivas
; aplicaciones, el contador. Al largo se lo define en 68h-1, o sea, 103
; bytes, que es el mínimo largo que pueden tener. En los 3 casos se
; apunta al desplazamiento de los TSS dentro del segmento de códigos.
; Sin embargo, este desplazamiento se debe corregir, ya que el campo
; deberá indicar la dirección lineal del TSS.
;
%define tss_inicial     20h
tss1:   dw      67h
        dw      TSS_inicial_struc
        db      0
        db      10001001b
        db      00000000b
        db      0
;
```

```
%define tss_sup         28h
tss2:   dw      67h
        dw      TSS_sup_struc
        db      0
        db      10001001b
        db      00000000b
        db      0
;
%define tss_inf         30h
tss3:   dw      67h
        dw      TSS_inf_struc
        db      0
        db      10001001b
        db      00000000b
        db      0
gdt_end:
;
; En la idt habrá 2 entradas inicializadas, la IRQ0 y la IRQ1. La
; primera de ellas es el timer y por ende la encargada de ejecutar
; alternadamente cada una de las dos tareas. La del teclado tiene como
; finalidad detectar el momento en que el usuario pulsa escape para
; finalizar el programa y retornar a modo real.
;
idtr                    dw 8*10-1       ;0,1,2,3,4,5,6,7,8,9
                        dd 0
real_idtr               resb 6
mask_real_pic1          db 0
;
idt:  resb  8*8
;
irq0: dw    int8han
      dw    08h
      db    0
      db    10000110b
      db    0
      db    0
irq1: dw    int9han
      dw    08h
      db    0
      db    10000110b
      db    0
      db    0
;
; Luego se define la estructura del TSS.
;
struc tss_struc
        .reg_PTL        resw 1
                        resw 1
        .reg_ESP0       resd 1
        .reg_SS0        resw 1
                        resw 1
        .reg_ESP1       resd 1
        .reg_SS1        resw 1
                        resw 1
        .reg_ESP2       resd 1
        .reg_SS2        resw 1
                        resw 1
        .reg_CR3        resd 1
        .reg_EIP        resd 1
        .reg_EFLAGS     resd 1
        .reg_EAX        resd 1
```

```
        .reg_ECX        resd 1
        .reg_EDX        resd 1
        .reg_EBX        resd 1
        .reg_ESP        resd 1
        .reg_EBP        resd 1
        .reg_ESI        resd 1
        .reg_EDI        resd 1
        .reg_ES         resw 1
                        resw 1
        .reg_CS         resw 1
                        resw 1
        .reg_SS         resw 1
                        resw 1
        .reg_DS         resw 1
                        resw 1
        .reg_FS         resw 1
                        resw 1
        .reg_GS         resw 1
                        resw 1
        .reg_LDT        resw 1
        .reg_T          resw 1
        .reg_IOMAP      resw 1
        .IOMAP          resd 1
endstruc
;
; Se definen las pilas, reservando 256 bytes para cada una. El label fin
; está al final del ejecutable.
;
%define pila_sup        fin+100h
%define pila_inf        fin+200h
;
; Una vez definida la estructura de los tss, se los debe inicializar,
; salvo al primero de ellos, ya que no será necesario. Lo mínimo
; necesario es inicializar la pila actual, el cs:eip, el IOMAP en ff el
; último byte, y en este caso se habilitan las interrupciones.
;
TSS_inicial_struc: resb tss_struc_size

TSS_sup_struc istruc tss_struc
        at tss_struc.reg_EIP,           dd tarea_sup
        at tss_struc.reg_EFLAGS,        dd 202h
        at tss_struc.reg_ESP,           dd pila_sup
        at tss_struc.reg_CS,            dw code_sel
        at tss_struc.reg_SS,            dw data_sel
        at tss_struc.reg_IOMAP,         dw 104
        at tss_struc.IOMAP,             dd 0ffffffffh
iend
;
TSS_inf_struc istruc tss_struc
        at tss_struc.reg_EIP,           dd tarea_inf
        at tss_struc.reg_EFLAGS,        dd 202h
        at tss_struc.reg_ESP,           dd pila_inf
        at tss_struc.reg_CS,            dw code_sel
        at tss_struc.reg_SS,            dw data_sel
        at tss_struc.reg_IOMAP,         dw 104
        at tss_struc.IOMAP,             dd 0ffffffffh
iend
;
; Se ralizan las tareas superior e inferior. La primera incrementa un
; valor y lo imprime en pantalla.
;
```

```
tarea_sup:
      sub sp,4
      mov bp,sp
      mov dword [bp+0],0
      mov ax,flat_sel
      mov ds,ax
      tarea_sup_main:
                inc dword [bp+0]
                mov eax,[bp+0]
                mov edi,0b8000h
                call mostrar_valor
jmp tarea_sup_main
;
; La tarea inferior decrementa un valor y lo imprime en pantalla.
;
tarea_inf:
      sub sp,4
      mov bp,sp
      mov dword [bp+0],0
      mov ax,flat_sel
      mov ds,ax
      tarea_inf_main:
                dec dword [bp+0]
                mov eax,[bp+0]
                mov edi,0b8000h+80*2*10
                call mostrar_valor
jmp tarea_inf_main
;
; Ambas taras utilizan una rutina muy similar a la utilizada en el
; ejemplo 4 para imprimir en video el valor de eax.
;
mostrar_valor:
;
; Recibe:
; en eax el valor a imprimir.
; en ds:edi la posición a imprimir.
;
      mov cx,8
      correr:
            rol eax,4
            mov byte [edi],'0'
            mov bl,al
            and bl,0fh
            add [edi],bl
            cmp byte [edi],':'
            jb ok
                  add byte [edi],'A'-':'
            ok:
            inc edi
            inc edi
            dec ecx
      jnz correr
ret
;
; Luego se realiza, en la irq0, el salto de tareas, que es una especie
; de scheduler muy sencillo que salta siempre a la otra tarea,
; verificando primero en cuál se encuentra, mediante el uso de la
; instrucción str.
;
int8han:
      pushad
```

```
      mov al,20h
      out 20h,al

      str ax
      cmp ax,tss_sup
      jne no_tarea_sup
            jmp tss_inf:0
            popad
            iret
      no_tarea_sup:
            jmp tss_sup:0
            popad
            iret
;
; Suele ser difícil comprender el por qué el popad y el iret después del
; salto de tarea. La razón es que cuando se realiza el salto de una
; tarea a otra en el tss correspondiente a la vieja tarea, se almacena
; la dirección de la siguiente instrucción a ejecutar. Si se supone que
; la rutina int8han se ejecuta estando en la tarea superior, el
; procesador ejecutará la instrucción jmp tss_inf:0, o sea, saltará a
; la tarea inferior, y almacenará, en el tss de la tarea superior, como
; próxima instrucción a ejecutar (cs:eip), el popad siguiente. Esto es
; correcto ya que, de volver a la tarea superior, se estaría todavía en
; la rutina de interrupción vieja y se deben recuperar de la pila los
; registros y retornar a la tarea superior (las pilas de ambas tareas
; son distintas).
;
; Por otro lado, se tendrá la rutina del teclado, la cual compara el
; scancode leído con el de la tecla escape y, en caso de ser éste
; último, se vuelve a modo real.
;
int9han:
        pushad
        in al,60h
        mov bl,al

        mov al,20h
        out 20h,al

        dec bl
        jz NEAR retornar_modo_real
        popad
iret
;
; De aquí en más no se encontrará nada nuevo, sino solamente las
; inicializaciones típicas de un programa de modo protegido.
;
inicio:
;
; Se guarda el cs de modo real.
;
      mov ax,cs
      mov [real_cs],ax
;
; Se acomoda el descriptor de código y de datos.
;
      xor eax,eax             ;pone en cero eax.
      mov ax,cs               ;carga el cs en eax.
      shl eax,4               ;en eax=base lineal de cs.
      mov ebx,eax             ;guarda en ebx.
      mov word [seg_code+2],ax
```

```
      mov word [seg_data+2],ax
      shr eax,16
      mov byte [seg_code+4],al
      mov byte [seg_data+4],al
;
; Se carga el IDTR, para lo que se inhabilitan las interrupciones.
;
      cli
      xor eax,eax
      mov ax,cs
      shl eax,4
      mov ebx,eax
      sidt [real_idtr]
      mov eax,ebx
      add eax,idt
      mov dword [idtr+2],eax
      lidt [idtr]
;
; Se guarda la máscara del PIC de modo real y se coloca el del programa
; en cuestión.
;
      in al,21h
      mov [mask_real_pic1],al
      mov al,011111100b
      out 21h,al
;
; Se colocan correctamente las bases de los tss's en los descriptores.
;
      xor eax,eax
      mov ax,cs
      shl eax,4
      mov ebx,eax
      add eax,TSS_inicial_struc
      mov word [tss1+2],ax
      shr eax,16
      mov byte [tss1+4],al
      mov eax,ebx
      add eax,TSS_sup_struc
      mov word [tss2+2],ax
      shr eax,16
      mov byte [tss2+4],al
      mov eax,ebx
      add eax,TSS_inf_struc
      mov word [tss3+2],ax
      shr eax,16
      mov byte [tss3+4],al
;
; Se coloca correctamente la base de la GDT.
;
      mov eax,ebx
        add eax,gdt
        mov dword [gdtr+2],eax
        lgdt [gdtr]             ;se carga el GDTR.
;
; Se hace que la irq0 se ejecute 1.19e6/2E7Ch veces por segundo, o sea,
; cada 10 mseg.
;
      mov ax,2E7ch
      out 40h,al
      mov al,ah
      out 40h,al
```

```
;
; Se pasa a modo protegido.
;
      mov eax,cr0
      or al,1
      mov cr0,eax
      jmp 08h:modo_protegido

      modo_protegido:
;
; Se carga el TSS inicial.
;
      mov ax,tss_inicial
      ltr ax
;
; Se habilitan las interrupciones.
;
      sti

;
; Se permanece para siempre aquí y se deja que el scheduler salte a la
; próxima tarea.
;
      jmp $
;
; Al pulsarse escape, la rutina de atención de la irq1 salta a
; retornar_modo_real.
;
      retornar_modo_real:
;
; Aquí se salta a través de la interrupción del teclado, la cual es una
; interrupt gate, por lo que no es necesario inhabilitar las
; interrupciones, que ya se encuentran inhabilitadas.
;
      mov ax,data_sel
      mov ds,ax
      mov es,ax
      mov fs,ax
      mov gs,ax
      mov ss,ax
      mov esp,0fffeh
      mov eax,cr0
      and al,0feh
      mov cr0,eax             ;se pasa a modo real.
      db 0eah                 ;código del salto.
      dw modo_real
      real_cs dw 0

      modo_real:
      mov ax,cs
      mov ss,ax
      mov sp,0fffeh
      mov ds,ax
      mov es,ax
      mov fs,ax
      mov gs,ax
;
; Se recuperan la IDT y las máscaras.
;
      lidt [real_idtr]
      mov al,mask_real_pic1
```

```
      out 21h,al
;
; Se lleva la IRQ0 a su velocidad normal.
;
      mov al,0h
      out 40h,al
      out 40h,al
      sti
;
; Se termina el programa.
;
        mov ah,4ch
        int 21h
fin:
```

```
; ej06.asm - salto de niveles de privilegio en modo protegido
;
; Protected Mode by Examples - 1era Edición - Octubre 2004
;
; Autor:
; Mariano Cerdeiro
; <m.cerdeiro@soix.com.ar>
; http://www.soix.com.ar/links/mpbyexamples.html
;
; compilar: nasm ej06.asm -o ej06.com
;
; Este programa ejemplifica los distintos métodos posibles para realizar
; saltos entre niveles de privilegio. El programa principal corre en
; nivel 3 y llama a rutinas del sistema operativo para leer el real time
; clock, leer un scan code, imprimir en la pantalla y terminar el
; programa.
;
; En los ejercicios anteriores nunca se mencionaron los niveles de
; privilegio, que son un punto fundamental en el momento de proteger
; una aplicación de otra y al sistema operativo de todas ellas. Los
; sistemas operativos más conocidos hoy en día no utilizan la
; segmentación y los niveles de privilegio para proteger la memoria,
; sino que se sirven de la paginación, ya que los segmentos son todos
; flat. Se denomina a un segmento flat cuando la
; base del mismo es cero y el límite 4 Gb.
;
; Como es sabido, al pasar a modo protegido, se entra al nivel 0 de
; privilegio, o sea, al de mayor privilegio. Para pasar de nivel 0 a 3
; es necesario saltar de tarea. Se podría hacer mediante un retf con
; cambio de nivel, al igual que se hace al retornar de las call gates,
; pero es conceptualmente más complicado de comprender.
;
; El método utilizado para cambiar de nivel de privilegio son dos call
; gates, una para acceder a los servicios del kernel, que deben correr
; en nivel 0 de privilegio, y la otra para acceder al servicio que
; imprime en pantalla, que corre sobre un segmento conforming.
;
; Para llevar a cabo la rutina de los servicios se puede utilizar un
; segmento conforming o uno normal de nivel 0. La diferencia es que,
; utilizando un conforming, no se cambia el nivel de privilegio, sino
; que el mismo se asume del nivel del código que llama. La ventaja
; consiste en que, al no cambiar de nivel, no se debe cambiar la pila
; por una de otro nivel, lo que hace que el acceso sea más rápido y que
; no se necesite ninguna verificación de niveles (una rutina conforming
; no podría acceder a datos o códigos a los que el código peticionante
; no pueda acceder). Por otro lado, es a veces necesario utilizar
; variables que se encuentran en nivel cero, o acceder a puertos que son
; inaccesibles desde niveles poco privilegiados. En estos casos se debe
; utilizar un segmento de nivel 0. Para saltar a un segmento de código
; de nivel 0 es necesario un mecanismo específico, como una
; interrupción, una call gate u otro método que lo permita. El saltar a
; un segmento conforming se puede hacer directamente.
;
; En un tercer caso se podría necesitar, por un lado, acceder a una
; rutina de nivel 0, para acceder a un puerto (por ejemplo el disco
; rígido), y por el otro, colocar los datos leídos en un segmento de
; datos de la rutina peticionante. Podría suceder que la aplicación que
; corre en nivel 3 llame a la rutina enviándole como parámetros una
; dirección del buffer donde escribir es:edi que esté en un nivel de
; mayor privilegio y al que no tiene acceso la aplicación. Para evitar
; este escenario, denominado escenario del Caballo de Troya, existe la
```

```
; instrucción arpl, la cual se utiliza para comparar el nivel de
; privilegio de es y el del cs pusheado en la pila por el procesador. La
; función compara ambos y coloca en el destino el mayor (numéricamente)
; de ambos.
;
; A veces, para evitar el escenario Caballo de Troya, se llama a los
; segmentos conforming como otro m, lo cual es cierto, pero, en
; ocasiones no es aplicable (por ejemplo cuando se necesita acceder a
; datos de nivel 0 o a puertos, como se menciona en el párrafo anterior,
; y a segmentos de nivel del peticionante).
;
; El descriptor de call gate es el siguiente:
;
;  31                    16 15         8 7         0
; +------------------------+-+-+-+-+----+---+-------+
; |     Offset 31..16      |P|DPL|0|D100|000|PC4..0 | 4
; |                        | |   | |    |   |       |
; +------------------------+-+-+-+-+----+---+-------+
;
; +------------------------+------------------------+
; |       Selector         |     Offset 15..0       | 0
; |                        |                        |
; +------------------------+------------------------+
;  31                    16 15                      0
;
;     - Offset 31..16: Parte alta del offset de la rutina de call gate.
;     - P: Indicación de presencia de la rutina de call gate.
;     - DPL: Nivel de privilegio necesario para poder utilizar este
;       descriptor.
;     - D: Indica si la call gate es de 16 o 32 bits, lo cual implica
;       que se almacenen en la pila los registros SP o ESP en caso de
;       cambio de pila y IP o EIP. Además indica si la cantidad indicada
;       en PC4..0 está dada en words o en double word.
;     - PC4..0: Param Count, cantidad de parámetros que son copiados de
;       una pila a la otra, si es que hay salto de nivel de privilegio.
;     - Selector: Selector de código de la atención de la call gate.
;     - Offset 15..0: Parte baja del offset de la rutina de call gate.
;
use16
org 100h
comienz:jmp inicio
;
; Es la primera vez que se decide alinear. Esta directiva comienza la
; próxima instrucción en una dirección múltiplo del valor que se le
; indique. Se utiliza  en este caso para alinear los datos (y/o el
; código) que son muy utilizados para ganar velocidad, a cambio de
; perder algunos bytes de espacio. Alinear el GDTR no tiene mucho
; sentido, ya que el mismo se carga una única vez, mientras que alinear
; la GDT es algo fundamental para obtener un alto rendimiento, ya que
; los micros están optimizados para leer de memoria 64 bytes o más en un
; único ciclo de lectura y acceder a la información alineada. Hay que
; tener en cuenta que el código se lee de a 16 bytes, por lo que, de
; querer acelerar la ejecución de una rutina, por ejemplo el scheduler u
; otra función critica, se podría colocar un align 16 antes de ella.
;
align 4
gdtr dw         gdt_end-gdt-1
     dd         0
;
align 8
gdt:                    resb 8
```

```
;
; 08h Segmento de código de nivel 0 de 64 Kb sobre el de modo real.
;
%define code_sel_0      08h
seg_code_0:     dw 0ffffh
                dw 0
                db 0
                db 10011010b
                db 0
                db 0
;
; 10h Segmento de datos de nivel 0 64 Kb sobre el de modo real.
;
%define data_sel_0      10h
seg_data_0:     dw 0ffffh
                dw 0
                db 0
                db 10010010b
                db 0b
                db 0
;
; 18h Segmento de pila de nivel 0 256 b.
;
%define stack_sel_0      18h
seg_stack_0:    dw 100h
                dw 0
                db 0
                db 10010010b
                db 0b
                db 0
;
; 20h segmento de código de nivel 3.
;
%define code_sel_3      20h+3
seg_code_3:     dw seg_code_3_end-seg_code_3_start-1
                dw 0
                db 0
                db 11111010b
                db 0b
                db 0
;
; 28h Segmento de datos de nivel 0 64 Kb sobre el de modo real.
;
%define data_sel_3      28h+3
seg_data_3:     dw seg_data_3_end-seg_data_3_start-1
                dw 0
                db 0
                db 11110010b
                db 0b
                db 0
;
; 30h Segmento de pila de nivel 3 256 b.
;
%define stack_sel_3      30h+3
seg_stack_3:    dw 100h
                dw 0
                db 0
                db 11110010b
                db 0b
                db 0
;
```

```
; 38h TSS de la tarea inicial
;
%define tss_in_selector 38h
tss_in:         dw 67h
                dw 0
                db 0
                db 10001001b
                db 00000000b
                db 0
;
; 40h TSS de la tarea que corre en nivel 3.
;
%define tss_selector    40h
tss:            dw 67h
                dw 0
                db 0
                db 10001001b
                db 00000000b
                db 0
;
; 48h Segmento de códigos CONFORMING.
;
%define code_sel_con    48h
seg_code_conf:  dw 0ffffh
                dw 0
                db 0
                db 10011110b
                db 10001111b
                db 0
;
; 50h Call Gate que salta a nivel 0 segmento de nivel 0.
;
%define cg_kernel_sel     50h
kernel:         dw 0
                dw code_sel_0
                db 0
                db 11100100b
                dw 0
;
; 58h Call Gate que salta a nivel x segmento conforming.
;
%define cg_print_sel    058h
print:          dw 0
                dw code_sel_con
                db 0
                db 11100100b
                dw 0
;
; 60h Segmento de video de nivel 3.
;
%define video_sel       060h
video:          dw 1000h
                dw 8000h
                db 0bh
                db 11110010b
                db 0b
                db 0
gdt_end:
;
; Fin de la GDT
;
```

```
; Se define la estructura del TSS.
;
struc tss_structure
        .reg_PTL        resw 1
                        resw 1
        .reg_ESP0       resd 1
        .reg_SS0        resw 1
                        resw 1
        .reg_ESP1       resd 1
        .reg_SS1        resw 1
                        resw 1
        .reg_ESP2       resd 1
        .reg_SS2        resw 1
                        resw 1
        .reg_CR3        resd 1
        .reg_EIP        resd 1
        .reg_EFLAGS     resd 1
        .reg_EAX        resd 1
        .reg_ECX        resd 1
        .reg_EDX        resd 1
        .reg_EBX        resd 1
        .reg_ESP        resd 1
        .reg_EBP        resd 1
        .reg_ESI        resd 1
        .reg_EDI        resd 1
        .reg_ES         resw 1
                        resw 1
        .reg_CS         resw 1
                        resw 1
        .reg_SS         resw 1
                        resw 1
        .reg_DS         resw 1
                        resw 1
        .reg_FS         resw 1
                        resw 1
        .reg_GS         resw 1
                        resw 1
        .reg_LDT        resw 1
        .reg_T          resw 1
        .reg_IOMAP      resw 1
        .IOMAP          resd 1
endstruc
;
; Para definir la base de las pilas se utiliza el label fin, que se
; encuentra al final del código, de modo que una pila se encontrará
; desde el final del código hasta 255 bytes y la otra a partir de 256
; bytes pasando el código y hasta 512.
;
%define pila_0          fin
%define pila_3          fin+256
;
; Para la TSS inicial no se define ningún contenido, ya que se la
; necesita únicamente para cambiar de nivel de privilegio, pasando, de
; esta forma, con el salto de tarea de nivel 0 a 3.
;
tss_in_struc:   resb tss_structure_size
;
; La segunda y en realidad única tarea, tendrá definidos sus registros
; de pila en 252, ya que es múltiplo de 4, y sus segmentos serán los
; definidos en la GDT, el EIP estará en 0, los EFLAGS en cero, por ende
; las interrupciones inhabilitadas y el IOPL en 0, el selector de datos
```

```
; estarán en data_sel_3 y el resto de los registros permanecerán en 0.
;
tss_struc:      istruc tss_structure
        at tss_structure.reg_ESP0,          dd 252
        at tss_structure.reg_SS0,           dw stack_sel_0
        at tss_structure.reg_EIP,           dd 0
        at tss_structure.reg_EFLAGS,        dd 002h
        at tss_structure.reg_ESP,           dd 252
        at tss_structure.reg_CS,            dw code_sel_3
        at tss_structure.reg_SS,            dw stack_sel_3
        at tss_structure.reg_DS,            dw data_sel_3
        at tss_structure.IOMAP,             dd 0ffffffffh
iend
;
; Ya que se inicializó antes en la GDT, se tendrán dos entradas al
; sistema, una para pasar a un segmento de nivel 0 y otra para pasar a
; un segmento conforming. El segmento conforming tendrá una única
; rutina, por lo que no es necesario distinguir para qué fue llamado.
; En el otro segmento sí será necesario, por lo que se definen 3
; variables con los números de servicios para no tener que recordarlos.
;
%define EXIT_SERVICE    0
%define KEY_SERVICE     1
%define TIMER_SERVICE   2
;
; La rutina de atención del "kernel" puede ser llamada de 3 formas
; distintas, en función del valor de eax.
;
kernel_routine:
      cmp eax,EXIT_SERVICE
      jne no_EXIT_SERVICE
;
; Si eax es igual a EXIT_SERVICE se retorna a modo real (hay que tener
; en cuenta que esto no se puede hacer desde nivel 3).
;
                jmp retornar_modo_real
      no_EXIT_SERVICE:
      cmp eax,KEY_SERVICE
      jne no_KEY_SERVICE
;
; Si eax es igual a KEY_SERVICE se retorna al último scan_code leído.
;
            in al,60h
            retf
      no_KEY_SERVICE:
      cmp eax,TIMER_SERVICE
      jne no_TIMER_SERVICE
;
; Y si eax es TIMER_SERVICE se lee la hora del Real Time Clock.
;
                xor eax,eax
                mov al,4
                out 70h,al
                in al,71h
                shl eax,8
                mov al,2h
                out 70h,al
                in al,71h
                shl eax,8
                mov al,0
                out 70h,al
```

```
                in al,71h
            retf
        no_TIMER_SERVICE:
        retf
;
; Esta rutina corre en el nivel de la rutina que la llame, en este
; ejemplo corre en nivel 3. La rutina imprime ds:esi a partir del
; offset indicado en edi en la pantalla hasta encontrar un carácter
; nulo.
;
print_routine:
        push es
        mov ax,video_sel
        mov es,ax

        mov ah,07h
        printing:
                mov al,[ds:esi]
                cmp al,0
                je no_printing
                mov [es:edi],ax
                inc esi
                inc edi
                inc edi
        jmp printing
        no_printing

        pop es
retf
;
; Aquí terminan las rutinas del sistema y comienza la rutina de la
; aplicación que corre en nivel 3.
;
seg_code_3_start:
seg_data_3_start:
;
; Se saltea la única variable, se coloca SHORT para que compile una
; instrucción de 2 bytes, lo cual implica un salto menor de 127 bytes
; hacia adelante.
;
jmp SHORT inicio_task
;
; La variable hora estará en el offset 2 si la base apunta, como se la
; definió, a seg_data_3_start. Se dejan en blanco los campos a llenar,
; pero se indica la puntuación. Es importante no olvidar el 0 al final.
;
%define hora 2
        db "  :  .  ",0

inicio_task:

        main:
;
; La rutina principal no hará más que llamar a los servicios. Primero
; pedirá la hora del sistema, lo cual desde nivel 3 no se puede hacer,
; por no tenerse acceso a los puertos (debido al IOPL y a que no están
; los bits en 0 en el mapa de entrada salida del TSS).
;
                mov eax,TIMER_SERVICE
                call cg_kernel_sel:0
;
```

```
; Luego se realiza la conversión de la hora a ascii para poder
; imprimirla, teniendo en cuenta que el formato entregado por el RTC es
; formato BCD, lo cual simplifica la tarea. Solamente se debe separar en
; nibles y sumar el ascii del 0.
;
                mov edi,hora+7
                mov cx,3
                time_print:
                        mov bl,al
                        and bl,0fh
                        mov byte [edi],"0"
                        add [edi],bl
                        mov bl,al
                        shr bl,4
                        dec edi
                        mov byte [edi],"0"
                        add [edi],bl
                        dec edi
                        dec edi
                        shr eax,8
                        dec ecx
                jnz time_print
;
; Luego se imprime el resultado en la pantalla, en el offset 0, o sea,
; arriba a la izquierda.
;
                mov esi,hora
                mov edi,0
                call cg_print_sel:0
;
; Se lee el último scan code.
;
                mov eax,KEY_SERVICE
                call cg_kernel_sel:0
                dec al
      jnz main
;
; Si el último scancode es 1, se retorna a modo real.
;
      mov eax,EXIT_SERVICE
      call cg_kernel_sel:0
;
; Aquí termina la rutina de nivel 3.
;
seg_data_3_end:
seg_code_3_end:
;
inicio:
;
; Se guarda el cs de modo real en real_cs para cuando se retorne, y se
; calcula la base lineal del programa, que se guarda en ebx para ir
; sumando los desplazamientos que sean necesarios.
;
        xor eax,eax
        mov ax,cs
        mov word [real_cs],ax
        shl eax,4
        mov ebx,eax
;
; Se acomoda la base del code segment de nivel 0.
;
```

```
        mov [seg_code_0+2],ax
        shr eax,16
        mov [seg_code_0+4],al
;
; Se acomoda la base del segmento de datos de nivel 0.
;
        mov eax,ebx
        mov [seg_data_0+2],ax
        shr eax,16
        mov [seg_data_0+4],al
;
; Se acomoda la base de la pila de nivel 0, que apunta a fin.
;
        lea eax,[ebx+pila_0]
        mov [seg_stack_0+2],ax
        shr eax,16
        mov [seg_stack_0+4],al
;
; Se acomoda la base del code segment de nivel 3, el cual comienza en
; seg_code_3_start.
;
        lea eax,[ebx+seg_code_3_start]
        mov [seg_code_3+2],ax
        shr eax,16
        mov [seg_code_3+4],al
;
; Se acomoda la base del data segment de nivel 3, el cual comienza en
; seg_data_3_start.
;
        lea eax,[ebx+seg_data_3_start]
        mov [seg_data_3+2],ax
        shr eax,16
        mov [seg_data_3+4],al
;
; Se acomoda la base de la pila de nivel 3.
;
        lea eax,[ebx+pila_3]
        mov [seg_stack_3+2],ax
        shr eax,16
        mov [seg_stack_3+4],al
;
; Se acomoda la base del tss inicial.
;
        lea eax,[ebx+tss_in_struc]
        mov [tss_in+2],ax
        shr eax,16
        mov [tss_in+4],al
;
; Se acomoda la base del tss de la tarea de nivel 3.
;
        lea eax,[ebx+tss_struc]
        mov [tss+2],ax
        shr eax,16
        mov [tss+4],al
;
; Se acomoda la base del segmento conforming.
;
        lea eax,[ebx]
        mov [seg_code_conf+2],ax
        shr eax,16
        mov [seg_code_conf+4],al
```

```
;
; Se acomoda el offset de la call gate del kernel.
;
        lea eax,[kernel_routine]
        mov [kernel],ax
        shr eax,16
        mov [kernel+6],ax
;
; Se acomoda el offset de la call gate del print.
;
        lea eax,[print_routine]
        mov [print],ax
        shr eax,16
        mov [print+6],ax
;
; Se acomoda la base de la gdt.
;
        lea eax,[ebx+gdt]
        mov [gdtr+2],eax
        lgdt [gdtr]
;
; Se pasa a modo protegido.
;
        cli
        mov eax,cr0
        or al,1
        mov cr0,eax
        jmp 08h:modo_protegido

      modo_protegido:
;
; Se carga el TR.
;
        mov ax,tss_in_selector
        ltr ax
;
; Se salta a la otra tarea para pasar de nivel.
;
        jmp tss_selector:0
;
; Y se retorna a modo real de la forma normal.
;

       retornar_modo_real:
       cli
       mov ax,data_sel_0
       mov ds,ax
       mov es,ax
       mov fs,ax
       mov gs,ax
       mov ss,ax
       mov eax,cr0
       and al,0feh
       mov cr0,eax             ;pasa a modo real.
       db 0eah                 ;código del salto.
       dw modo_real
       real_cs resw 1

       modo_real:
       mov ax,cs
       mov ss,ax
```

```
        mov sp,0fffeh
        mov ds,ax
        mov es,ax
        mov fs,ax
        mov gs,ax

        mov ah,4ch
        int 21h
fin:
```

```
; ej07.asm - modo protegido 32 bits y pila expand down
;
; Protected Mode by Examples - 1era Edición - Octubre 2004
;
; Autor:
; Mariano Cerdeiro
; <m.cerdeiro@soix.com.ar>
; http://www.soix.com.ar/links/mpbyexamples.html
;
; compilar: nasm ej07.asm -o ej07.com
;
; Los microprocesadores de Intel IA-32 están optimizados para correr
; código de 32 bits. Hasta este ejemplo se utilizó siempre código de 16
; bits, el único que se puede correr en modo real. En modo protegido, y
; mediante el seteo del bit D del descriptor de segmento de código, se
; puede setear si el segmento es de 16 bits o de 32. Además de indicarlo
; en el descriptor de código, es necesario indicar cuándo compilar con
; código de 32 bits al compilador. La diferencia entre código de 16 y 32
; reside en qué tamaño de registros y en qué tipo de direccionamiento
; son utilizados de forma predeterminada. En modo de 16 bits se pueden
; usar registros de 32 y direccionamiento de 32, así como en un segmento
; de 32 se pueden usar direccionamiento de 16 y registros de 16. Sin
; embargo, en los segmentos de 16 bits las instrucciones de 16 bits son
; más cortas y se ejecutan más rápido; lo contrario sucede entonces en
; un segmento de 32 bits. Las instrucciones más cortas son las de 32
; bits y por eso son más rápidas. También se obtienen otras ventajas,
; como que los segmentos de códigos pueden tener un largo mayor a 64 Kb.
;
; Además del segmento de código de 32 bits, en este ejemplo se utiliza
; un segmento de pila de 32 bits expand down, lo que implica que SIEMPRE
; se pushean o popean valores de 32 bits para mantener alineada la pila,
; por lo que un push ax decrementará esp en 4, copiará ax y dejará dos
; bytes en 0, o en lo que estaban según el modelo del procesador. Este
; manejo de la pila es transparente para el programador, cuando hace pop
; ax, recibirá dos bytes de ax, y los otros dos serán desechados, de
; modo que se asegura tener la pila siempre alineada a 4 bytes, lo cual
; mejora la performance. El segmento de pila es además expand down, lo
; que implica que los offset válidos van desde el límite a 0FFFFFFFFh,
; con lo que se optimiza el uso de la pila en caso de necesitar
; ampliarla. Un ejemplo:
;
; Si se tiene una pila en un segmento de datos normal con límite 4
; Kbytes, se la inicializará con esp apuntando a 4 Kbytes. Si ésta se
; llena, esp apuntará a 0. En caso de que se la desee ampliar, no
; bastará con cambiar el límite, ya que esp seguirá siendo 0. Es por eso
; que se debe cambiar el límite, por ejemplo por 8 Kbytes, copiar los 4
; Kbytes de datos de la pila que se encontraban en los offset 0 a 4 Kb a
; los offsets 4 Kb a 8 Kb y apuntar el esp a 4 Kb. Todo esto es para
; modificar el tamaño de la pila en caso de que se trate de un segmento
; normal. En cambio, en un segmento expand down, se coloca inicialmente
; el segmento en expand down con límite en 0FFFFF000h y el esp apuntando
; en 0FFFFFFFCh, por lo que lo que se tienen 4 Kb de espacio en pila.
; Cuando se va llenado la pila, esp se decrementa hasta llegar a
; 0FFFFF000h, momento en que se deberá ampliar la pila. Sin embargo, en
; este caso bastará con cambiar el límite que anteriormente estaba en
: 0FFFFF000h por 0FFFFE000h, con lo que el esp seguirá apuntando a
; 0FFFFF000h y no habrá que mover los datos de lugar. De esta forma,
; ampliar la pila requiere únicamente modificar su límite. Es por eso
; que se recomienda utilizar los segmentos expand down para los
; segmentos de pila, como se hace en este ejemplo.
;
```

```
use16
org 100h
jmp inicio
;
gdtr:
        dw gdt_end-gdt-1
        dd 0
;
gdt:  resb  8
;
; 08h Segmento de código de 64 Kb sobre el de modo real.
;
%define cs_16_selector 8
        dw     0FFFFh          ;límite en modo real 15.00
        dw     0               ;base 15.00
        db     0               ;base 23.16
        db     10011010b       ;tipo P1 DPL0 S1 CR
        db     00000000b       ;G0 D/B0 Lim 19.16 0
        db     0               ;base 31.24
;
; 10h Segmento de datos de 64 Kb sobre el de modo real.
;
%define ds_16_selector 10h
        dw     0FFFFh          ;límite en modo real 15.00
        dw     0               ;base 15.00
        db     0               ;base 23.16
        db     10010010b       ;tipo P1 DPL0 S1 DW
        db     00000000b       ;G0 D/B0 Lim 19.16 0
        db     0               ;base 31.24
;
; Un segmento flat, como se ha mencionado anteriormente, se denomina a
; un segmento que tiene de base 0 y límite 4 Gb.
;
; 18h Segmento de código de 32 bits flat.
;
%define cs_32_selector 18h
        dw     0FFFFh          ;límite en modo real 15.00
        dw     0               ;base 15.00
        db     0               ;base 23.16
        db     10011010b       ;tipo P1 DPL0 S1 DW
        db     11001111b       ;G0 D/B0 Lim 19.16 0
        db     0               ;base 31.24
;
; 20h Segmento de datos flat.
;
%define flat_selector 20h
        dw     0FFFFh          ;límite en modo real 15.00
        dw     0               ;base 15.00
        db     0               ;base 23.16
        db     10010010b       ;tipo P1 DPL0 S1 DW
        db     10001111b       ;G0 D/B0 Lim 19.16 0
        db     0               ;base 31.24
;
; 28h Segmento de pila EXPAND DOWN de 4096 bytes.
;
%define ss_32_selector 28h
        dw     0FFFEh
        dw     00000h
        db     000h
        db     10010110b
        db     11001111b
```

```
        db     0
gdt_end:
;
; Para dejar más prolijo el programa principal, se define un macro con:
;
; %macro el_nombre cantidad_de_parametros
;
; El primer parámetro es el nombre del macro y el segundo la cantidad
; de parámetros que recibe. Para llamar a un macro de 1 o más
; parámetros se hace:
;
; el_nombre %1, %2
;
; Para finalizar el macro se utiliza la sentencia:
;
; %endmacro.
;
; Si se desean utilizar labels dentro del macro para saltos se debe
; usar, por ejemplo:
;
; %%pepe:
;
; De esta forma el macro puede ser colocado en varias partes del
; programa, y en el momento de compilación se cambiarán los %% por
; valores .00pepe, luego .01pepe, etcétera, para que en el linkeo no se
: encuentren duplicados los labels.
;
%macro      init_gdt 0
      xor eax,eax
      mov ax,cs
      shl eax,4
      lea ecx,[eax+4096+4095]
      and ecx,0FFFFF000h
      mov ebx,gdt
      mov [ebx+cs_16_selector+2],ax
      mov [ebx+ds_16_selector+2],ax
      mov [ebx+ss_32_selector+2],cx
      ror eax,16
      ror ecx,16
      mov [ebx+cs_16_selector+4],al
      mov [ebx+cs_16_selector+7],ah
      mov [ebx+ds_16_selector+4],al
      mov [ebx+ds_16_selector+7],ah
      mov [ebx+ss_32_selector+4],cl
      mov [ebx+ss_32_selector+7],ch
      rol eax,16
      add eax,gdt
      mov [gdtr+2],eax
      lgdt [gdtr]             ;carga la GDT
      clc                     ;no puede fallar
%endmacro
;
; Se comienza el programa principal.
;
inicio:
;
; Se inicializa la GDT mediante el MACRO correspondiente.
;
      init_gdt
;
; Se guarda el cs para retornar a modo real.
```

```
;
      mov [real_cs],cs
;
; Como en modo protegido 32 bits se está utilizando un segmento flat,
; la base se encontrará en 0 y el límite en 4 Gb. El código cargado por
; DOS puede quedar cargado en distintos offset, por lo que es necesario
; calcular el offset inicial del programa. El valor del offset se
; almacena en una variable denominada offset_32, que se utiliza para
; realizar un salto largo al segmento flat.
;
      xor eax,eax
      mov ax,cs
      shl eax,4
      add eax,cs_32_start
      mov [offset_32],eax
;
; Se pasa a modo protegido de 16 bits.
;
      cli
      mov eax,cr0
      or al,1
      mov cr0,eax
      jmp cs_16_selector:modo_protegido_16
      modo_protegido_16:
;
; E inmediatamente a modo protegido de 32 bits.
;
      db 066h
      db 0eah
      offset_32 dd 0
      dw cs_32_selector
;
; Este código se ejecuta al retornar a modo protegido de 16 bits. Se
; carga la pila con valores de 16 bits.
;
      retornar_modo_real:
      mov ax,ds_16_selector
      mov ss,ax
      mov esp,0fffeh
;
; Se retorna a modo real.
;
      mov eax,cr0
      and al,0feh
      mov cr0,eax
      db 0eah
      dw modo_real
      real_cs resw 1
;
; En modo real se inicializan nuevamente todos los selectores.
;
      modo_real:
      mov ax,cs
      mov ss,ax
      mov sp,0fffeh
      mov ds,ax
      mov es,ax
      mov fs,ax
      mov gs,ax
;
; Y se retorna al DOS.
```

```
;
      mov ah,4ch
      int 21h
;
; Lo que sigue es el código de 32 bits. La sentencia:
;
; align 16
;
; Alinea el código a una posición múltiplo de 16 bytes mientras que:
;
; use 32
;
; Indica al compilador que a partir de ese punto debe codificar las
; instrucciones como de 32 bits.
;
align 16
use32
cs_32_start:
;
; Se inicializa la pila de 32 bits con un segmento expand down.
;
      mov ax,ss_32_selector
      mov ss,ax
      mov esp,0ffffffch
;
; No se hace más que verificar el correcto funcionamiento de la pila.
;
      push eax
      pop ebx
      mov eax,ebx
;
; Y se retorna a modo protegido de 16 bits.
;
      db 066h
      db 0eah
      dw retornar_modo_real
      dw cs_16_selector
fin:
```

```
; ej08.asm - paginación
;
; Protected Mode by Examples - 1era Edición - Octubre 2004
;
; Autor:
; Mariano Cerdeiro
; <m.cerdeiro@soix.com.ar>
; http://www.soix.com.ar/links/mpbyexamples.html
;
; compilar: nasm ej08.asm -o ej08.com
;
; En este ejercicio se ejemplifica el funcionamiento del procesador con
; la paginación habilitada. El programa funciona de modo similar a un
; inicializador de un sistema operativo. Se comienza a ejecutar en modo
; real, copia su código de 32 bits por arriba del mega + 64 Kbytes de
; memoria, donde carga en el siguiente orden:
;     - GDT
;     - Código del programa
;     - PDE
;     - PTE
;
; Donde PDE y PTE son tablas de paginación que se explican en los
; párrafos siguientes. El ejercicio mediante la paginación deja un mapa
; de memoria lineal de la siguiente forma:
;
;    Memoria Lineal       Memoria Física      Descripción
;   Base      Límite      Base      Límite
; 00000000h  00000FFFh  00110000h  00110FFFh   GDT inicializada
; 00001000h  0000FFFFh  ---------  ---------   GDT no inicializada
; 00010000h  00010FFFh  ---------  ---------   IDT no inicializada
; 00011000h  00011FFFh  00111000h  00111FFFh   Código de 32 bits
; 00012000h  00012FFFh  00112000h  00112FFFh   PDE
; 00013000h  00013FFFh  00113000h  00123FFFh   PTE
; 00014000h  00412FFFh  ---------  ---------   PTE's no inicializadas
;
; NOTA: Los límites en esta tabla hacen referencia a la máxima dirección
; lineal, NO al largo del segmento.
;
; Como se puede observar, se reservan 4 Mbytes de memoria lineales para
; las futuras tablas PTE's, pero esto no ocupa memoria física, ya que no
; se puede direccionar ese espacio de memoria hasta no inicializarlo, al
; igual que el espacio reservado linealmente para la GDT no inicializada
; y la IDT no inicializada.
;
; Una vez en esta situación, el programa inicializa memoria para la
; pila. Como la pila es decreciente, apunta al final del área del
; sistema operativo, lo más normal es a 1 Gb, 2 Gb, 3 Gb ó 4 Gb, en este
; ejemplo el esp apunta a 3ffffffc, o sea, 4 Kbytes por debajo del Gb de
; memoria.
;
; Hay un punto que no se mencionó, pero que se deduce del párrafo
; anterior: para este ejercicio se utilizan 2 segmentos, uno de código
; de 32 bits y otro de datos flat, por lo que la base es 0 y el límite 4
; Gbytes.
;
;
; Paginación
;
; La unidad de paginación se habilita seteando el bit PE (bit 31) del
; cr0. Al habilitar la paginación, todas las direcciones, una vez
```

; transformadas en lineales (base del segmento + dirección efectiva), se
; descomponen para transformarse en física de distintas formas, según el
; tipo de paginación implementada. En el 80386, primer procesador con
; paginación de la línea Intel IA-32, las páginas son todas de 4 Kb. Más
; adelante aparecieron otras implementaciones para trabajar con páginas
; de 2 y 4 Mbytes. En este ejemplo se utilizan únicamente páginas de
; 4 Kb.
;
; La paginación es ideal para implementar la memoria virtual, pero los
; principales sistemas operativos de hoy la utilizan además como
; mecanismo de protección, protegiendo así las áreas de memoria del SO
; de cada aplicación y del kernel.
;
; Si se implementa la paginación mediante páginas de 4 Kb, el mecanismo
; divide a las direcciones lineales en 3 campos, 2 de 10 bits y uno de
; 12 bits. Para decodificar estos campos se utilizan dos tipos de
; tablas, PDE (Page Directory Entry) y PTE (Page Table Entry). La PDE se
; encuentra físicamente en la dirección de memoria apuntada por el cr3,
; alineada en 4 Kbytes, por lo que los 12 bits menos significativos del
; cr3 no son tenidos en cuenta. Cada entrada de la PDE está compuesta
; por 4 bytes, por lo que la PDE tiene 1024 entradas. Cada una de ellas
; puede apuntar a la dirección física de una PTE, cuyas 1024 entradas,
; también de 4 bytes cada una, pueden apuntar a una página en memoria.
;
;   +----------+----------+------------+
;   :0000000011:0000000111:000000000000:
;   +----------+----------+------------+
;   PDE Index PTE Index PAGE Offset
;
;   PDE   PTE   PAGE
;   +--------------+   +---------------+   +---------------+
; 1023:   :   1023:   :   :   :
; 1022:   :   1022:   :   :   :
; 1021:   :   1021:   :   :   :
;   //   //   //   //   //   /
;   7:   :   7:   :--+   :   :
;   6:   :   6:   : :   :   :
;   5:   :   5:   : :   :   :
;   4:   :   4:   : :   :   :
;   3:   :---+ 3:   : :   :   :
;   2:   :   : 2:   : :   :   :
;   1:   :   : 1:   : :   :   :
;   0:   :   : 0:   : :   :   :
; +--->+---------------+   +-->+---------------+ +-->+---------------+
; :
; +---------------+
; :   CR3   :
; +---------------+
;
; Entonces, por ejemplo, para decodificar la dirección lineal binaria:
; 00000000110000000111000000000000b a física, el procesador busca, a
; partir de la base indicada por el cr3, la tercera entrada de la PDE,
; donde se indica la base física en memoria de la PTE. A partir de la
; base de la PTE se selecciona la entrada número 7, obteniéndose de esta
; la dirección física de la base de la página. Por último, los 12 bits
; de la dirección lineal son el offset dentro de la página que se está
; referenciando.
;
; Para llevar a cabo un programa que implemente la paginación de 4
; Kbytes se debe definir al menos una PDE, con al menos una entrada que
; apunte a una PTE. También se debe tener en cuenta que al habilitar e

```
; inhabilitar la paginación, las direcciones lineales y físicas tienen
; que coincidir.
;
use16
org 100h
jmp inicio
;
%define     gdt_basep   110000h
%define     gdt_base    0
%define     os_basep    111000h
%define     os_base     11000h
;
; Se hacen algunas definiciones. La 'p' al final hace referencia a que
; se trata de una posición de memoria física. De esta forma, la base
; física de la gdt está en 110000h, mientras que la lineal está en 0.
;
gdtr    dw      gdt_end-gdt-1
        dd      0
;
gdt:    resb    8
;
; 08h Segmento de código de 64 Kb sobre el de modo real.
;
%define cs_sel_16 8
cs_des_16 dw      0FFFFh      ;límite en modo real 15.00
        dw  0                 ;base 15:00
        db  0                 ;base 23:16
        db  10011010b         ;tipo P1 DPL0 S1 CR
        db  00000000b         ;G0 D/B0 Lím 19.16 0
        db  0                 ;base 31:24
;
; 10h Segmento de datos de 64 Kb sobre el de modo real.
;
%define ds_sel_16 10h
ds_des_16 dw      0FFFFh      ;límite en modo real 15.00
        dw  0                 ;base 15:00
        db  0                 ;base 23:16
        db  10010010b         ;tipo P1 DPL0 S1 DW
        db  00000000b         ;G0 D/B0 Lim 19:16 0
        db  0                 ;base 31:24
;
; 18h Segmento de código flat de 32 bits.
;
%define cs_sel_32 18h
cs_des_32 dw      0ffffh      ;límite 15:00
      dw    0                 ;base 15:00
      db    0                 ;base 23:16
      db    10011010b         ;tipo P1 DPL0 S1 CR
      db    11001111b         ;G1 D/B1 Lim 19:16 fh
      db    0                 ;base 31:24
;
; 20h Segmento de datos flat de 32 bits.
;
%define ds_sel_32 20h
ds_des_32 dw      0ffffh      ;límite 15:00
      dw    0                 ;base 15:00
      db    0                 ;base 23:16
      db    10010010b         ;tipo P1 DPL0 S1 DW
      db    11001111b         ;G1 D/B1 Lim 19:16 fh
      db    0                 ;base 31:24
gdt_end:
```

```
;
inicio:
;
; Se habilita A20, ya que se utiliza memoria por arriba del mega, es
; necesario habilitar A20.
;
      cli
      mov al,0d1h
      out 64h,al
      mov al,0dfh
      out 60h,al      ;Habilita A20.
;
; Se calcula la base del cs y se almacena en ebx.
;
      xor eax,eax
      mov ax,cs
      shl eax,4
      mov ebx,eax
;
; Se carga el GDTR con la base de la gdt.
;
      add eax,gdt
      mov [gdtr+2],eax
      lgdt [gdtr]
;
; Se inicializa la base del segmento de código y del de datos de 16
; bits.
;
        mov eax,ebx
        mov [cs_des_16+2],ax
        mov [ds_des_16+2],ax
        shr eax,16
        mov [cs_des_16+4],al
        mov [ds_des_16+4],al
;
; Se almacena el cs para retornar a modo real.
;
        mov [real_cs],cs
;
; Se pasa a modo protegido
;
        cli
        mov eax,cr0
        or al,1
        mov cr0,eax
        jmp cs_sel_16:modo_prot_16

modo_prot_16:
      mov ax,ds_sel_16
      mov ds,ax
      mov ax,ds_sel_32
      mov es,ax
;
; Se copia la gdt a gdt_basep, para lo que se apunta el ds al segmento
; de modo real y el es al flat. Luego se realiza una especie de rep
; movsb, pero en forma de loop, ya que se está copiando, en un segmento
; de 16 bits, datos de un segmento de límite de 16 bits a uno de límite
; de 32 bits. Para copiar de ds:si a es:edi no sería correcto utilizar
; rep movsd.
;
      mov esi,gdt
```

```
      mov edi,gdt_basep
      mov ecx,gdt_end-gdt
      .rep_movsb:
            mov al,[ds:si]
            mov [es:edi],al
            inc esi
            inc edi
            dec ecx
      jnz .rep_movsb
      mov dword [gdtr+2],gdt_basep
      lgdt [gdtr]
;
; Se copia el código de 32 bits a partir de os_basep.
;
copy_code:
;
; Como el label rep_movsb es un label local (el . al inicio indica
; que es local), se pueden definir muchos con el mismo nombre, siempre
; y cuando entre ellos exista un label normal, por lo que aquí se
; encuentra el label copy_code.
;
      mov esi,modo_prot_32
      mov edi,os_basep
      mov [cs_32_entry_point],edi
      mov ecx,(modo_prot_32_end-modo_prot_32)
      .rep_movsb:
            mov al,[ds:esi]
            mov [es:edi],al
            inc esi
            inc edi
            dec ecx
      jnz .rep_movsb
;
; Se copia el código, sólo el de 32 bits, a la base os_basep.
;
; Se debe hacer un salto arriba del mega de memoria y al segmento de
; código flat de 32 bits. Sin embargo, por estar en un segmento de 16
; bits, los offset de los saltos lejanos son de 16 bits, y por eso se
; antepone el prefijo 66h para realizar un salto a un offset mayor a 2
; bytes.
;
; La próxima instrucción a ejecutar después del salto es jmp inicio32
; luego del label modo_prot_32
;
      db 066h,0eah
      cs_32_entry_point dd 0
      dw cs_sel_32
;
; Las siguientes líneas de código son ejecutadas en el momento de
; retornar a modo protegido de 16 bits.
;
return_modo_real_16:
;
; Se carga un valor de selector válido antes de retornar a modo real,
; como también una pila.
;
      cli
      mov ax,ds_sel_16
      mov ds,ax
      mov es,ax
      mov fs,ax
```

```
      mov gs,ax
      mov ss,ax
      mov esp,0fffeh
      mov eax,cr0
      and al,0feh
      mov cr0,eax
      db 0eah
      dw modo_real
      real_cs dw 0
;
; En modo real se vuelve a colocar un valor válido en todos los
; selectores y en la pila.
;
modo_real:
      mov ax,cs
      mov ds,ax
      mov es,ax
      mov fs,ax
      mov gs,ax
      mov ss,ax
      mov esp,0fffeh
;
; Por último se retorna al DOS.
;
      mov ah,4ch
      int 21h
;
; Al igual que en el ejemplo anterior, se utiliza segmento de 32 bits y
; se alinea la primera instrucción a 16 bytes.
;
use32
align 16
modo_prot_32:
      jmp inicio32
;
; Para una mejor performance del programa las variables se alinean a 4
; bytes. Por otro lado, Intel también recomienda separar páginas de
; código de páginas de datos, lo que en este ejercicio no se hace.
;
align 4
freememF    equ   $-modo_prot_32+os_basep
freemem     equ   $-modo_prot_32+os_base
      dd    ((os_basep+modo_prot_32_end-modo_prot_32+4095)&0fffff000h)
PDEF        equ   $-modo_prot_32+os_basep
PDE         equ   $-modo_prot_32+os_base
      dd    0
PTEF        equ   $-modo_prot_32+os_basep
PTE         equ   $-modo_prot_32+os_base
      dd    0
;
; Las líneas a partir de align 4 hasta aquí son tal vez algo complejas
; de comprender. Hacen referencia a 3 variables. Para definir una
; variable normalmente se utiliza:
;
; nombre_var      dd    0
;
; Donde nombre_var es, para el compilador, el offset de la variable.
; Pero en este ejemplo no se puede direccionar la variable de este
; modo, ya que el offset sería en relación al org 100h y no al offset
; real de las variables que es respecto a la base del segmento la cual
; es 0.
```

```
; El código a partir del label modo_prot_32 se copia por arriba del mega
; a la posición os_basep (111000h). Al habilitarse la paginación, una
; vez inicializada, la posición os_base es igual a 11000h y es por eso
; que los offsets de las variables son distintos antes y después de
; habilitar e inicializar la paginación. Para solucionar esto se definen
; dos offsets por variable, el terminado en "F" hace referencia a la
; posición física de la variable y a la lineal, antes de habilitar la
; paginación, mientras que la que no tiene una "F" se refiere a al
; offset una vez habilitada la paginación y finalizada su
; inicialización.
;
; A partir de aquí comienza el programa de 32 bits.
;
inicio32:
;
; Se cargan los selectores de 32 bits en todos los selectores a usar y
; además se inicializa la pila apuntando a un sector de memoria todavía
; no decodificado por la paginación, que no se encuentra aún habilitada.
; Hasta que se decodifique se debe tener especial cuidado en no utilizar
; la pila. En caso de que se utilice por error se generará una
; excepción, a menos que el equipo cuente con 1 Gb de memoria, que es
; adonde se encuentra apuntando la pila.
;
      mov ax,ds_sel_32
      mov es,ax
      mov ds,ax
      mov ss,ax
      mov esp,3ffffffch
;
; Se coloca la PDE y al PTE en el primer espacio libre de memoria, para
; lo que se utiliza la variable freememF.
;
      mov eax,[freememF]
      mov [PDEF],eax
      add eax,1000h
      mov [PTEF],eax
      add dword [freememF],2000h
;
; Se colocan en cero todas las entradas de la PDE y PTE.
;
      mov edi,[PDEF]
      mov ecx,2048
      xor eax,eax
      cld
      rep stosd
;
; Se inicializa el cr3 apuntando a la posición física de la PDE.
;
      mov eax,[PDEF]
      mov cr3,eax
;
; Se inicializa la PTE apuntando a la posición física de la PTE. Los
; bits 0, 1 y 2 indican respectivamente que la página está presente,
; es de usuario y de escritura, a eso se debe la instrucción or con
; el valor 111b como origen.
;
      mov eax,[PTEF]
      or al,111b
      mov edi,[PDEF]
      mov [edi],eax
;
```

```
; Se inicializan las entradas de la PTE a las posiciones físicas que
; corresponden a la gdt y al programa principal a partir de la base
; física gdt_basep.
;
      mov edi,gdt_basep
      shr edi,12
      shl edi,2
      add edi,[PTEF]
      mov eax,gdt_basep
      or al,111b
;
; En ecx se indica la cantidad de páginas a direccionar.
; (modo_prot_32_end-modo_prot_32+4095)/4096 son la cantidad de páginas
; utilizadas por el código, a este valor se le suma una página por la
; GDT, una por la PDE y una por la PTE.
;
; Se hace un ciclo para setear la memoria lineal con la física
; correspondiente.
;
      mov ecx,(modo_prot_32_end-modo_prot_32+4095)/4096+3
      init_PTE1:
            mov [edi],eax
            add edi,4
            add eax,1000h
            dec ecx
      jnz init_PTE1
;
; Se habilita la paginación.
;
      mov eax,cr0
      or eax,80000000h
      mov cr0,eax
;
; A este punto se tiene un mapa de memoria como el siguiente:
;
; <bochs:14> show "tab"
; cr3: 00112000
; 00110000 - 00110000:   110000 -   110000
; 00111000           :   111000 (  111000) in TLB
; 00112000 - 00113000:   112000 -   113000
;
; El show “tab” es el comando de bochs para mostrar la las tablas de
; paginación. Primero indica el valor del cr3 y luego decodifica las
; direcciones lineales en físicas. El bochs, además de decodificar
; la memoria, indica qué páginas se encuentran en la TLB, por ejemplo,
; la 111000 en este caso.
;
; Se direcciona la posición lineal 0 apuntando a la GDT.
;
      mov edi,[PTEF]
      mov eax,gdt_basep
      or al,7
      mov [edi],eax
;
; Entonces queda un mapa de memoria como el siguiente:
;
; <bochs:21> show "tab"
; cr3: 00112000
; 00000000 - 00000000:   110000 -   110000
; 00110000           :   110000 (  110000) in TLB
; 00111000           :   111000 (  111000) in TLB
```

```
; 00112000 - 00112000:   112000 -   112000
; 00113000           :   113000 (  113000) in TLB
;
; Donde la posición lineal 0 apunta a la GDT, como la 11000.
;
; En la siguientes líneas se hace lo mismo pero con el código, la PDE y
; la PTE. A partir de la dirección lineal 11000 ya que se dejan 656536
; bytes para la GDT y 4096 para la IDT. A pesar de que inicialmente se
; utiliza sólo una página para la GDT, se deja el espacio lineal como
; para utilizarla entera, al igual que una página para la IDT, que en
; este ejemplo no se utiliza. Las direcciones físicas a utilizar son
; a partir de os_basep.
;
      mov edi,11000h
      shr edi,12
      shl edi,2
      add edi,[PTEF]
      mov eax,os_basep
      or al,111b
;
; En ecx se indica la cantidad de páginas a direccionar.
; (modo_prot_32_end-modo_prot_32+4095)/4096 son la cantidad de páginas
; utilizadas por el código, a este valor se le suma una página por la
; PDE y una por la PTE.
;
       mov ecx,(modo_prot_32_end-modo_prot_32+4095)/4096+2
       init_PTE2:
              mov [edi],eax
              add edi,4
              add eax,1000h
              dec ecx
       jnz init_PTE2
;
; Quedando el mapa de direccionamiento de la siguiente forma:
;
; <bochs:44> show "tab"
; cr3: 00112000
; 00000000 - 00000000:   110000 -   110000
; 00011000 - 00013000:   111000 -   113000
; 00110000           :   110000 (  110000) in TLB
; 00111000           :   111000 (  111000) in TLB
; 00112000 - 00112000:   112000 -   112000
; 00113000           :   113000 (  113000) in TLB
;
; Se puede observar que cada dirección física se encuentra mapeada
; dos veces linealmente. Como la idea es acomodar todo al inicio
; de la memoria lineal, se deben utilizar las direcciones lineales
; superiores a 110000, para lo que se debe seguir ejecutando el mismo
; código, pero en las direcciones lineales 11000 en vez de 111000.
; Para esto se debe realizar un salto. Además, a partir de aquí,
; se utilizarán las direcciones lineales 12000 y 13000 para direccionar
; la PDE y la PTE. Para ello se modifican las variables PDE y PTE, donde
; además se ve que PTEF y PTE son la misma variable FÍSICAMENTE, como
; PDE y PDEF, pero linealmente son distintas.
;
      sub dword [PDE],os_basep-os_base
      sub dword [PTE],os_basep-os_base
;
; Se realiza el salto.
;
      jmp cs_sel_32:bajando-modo_prot_32+os_base
```

```
;
bajando:
;
; Ahora habrá que borrar las entradas que ya no se utilizarán, para
; dejar un diagrama de memoria prolijo. Se borran todas las entradas
; superiores al mega.
;
      mov edi,gdt_basep
      shr edi,12
      shl edi,2
      add edi,[PTE]
      xor eax,eax
      mov ecx,(modo_prot_32_end-modo_prot_32+4095)/4096+2+1
      rep stosd
;
; Queda un diagrama de memoria incorrecto, ya que algunas páginas se
; encontraban en la TLB y, al borrarlas de la PDE, no se borran
; automáticamente de la PDE. Es necesario flushear la TLB cargando
; nuevamente el cr3.
;
      mov eax,cr3
      mov cr3,eax
;
; Ahora el diagrama de memoria es:
;
; <bochs:59> show "tab"
; cr3: 00112000
; 00000000 - 00000000:   110000 -   110000
; 00011000 - 00013000:   111000 -   113000
;
; En este punto, sólo falta inicializar la pila, que se encuentra
; apuntando a 3fffffff, direcciones lineales que no existen.
;
; Primero se debe generar una PTE que pueda direccionar esa memoria.
;
      mov edi,3ffff000h
      shr edi,22
      shl edi,2
      add edi,[PDE]
      mov eax,[freemem]
      or eax,7
      mov [edi],eax
      add dword [freemem],1000h
;
; Esta nueva página de memoria (PTE) se debe poder direccionar en
; espacio lineal para poder acceder a ella.
;
      mov edi,3ffff000h
      shr edi,12
      shl edi,2
      add edi,[PTE]
      shr edi,12
      shl edi,2
      add edi,[PTE]
      mov [edi],eax
;
; Se inicializa toda la PTE en 0.
;
      mov edi,3ffff000h
      shr edi,12
      shl edi,2
```

```
      and edi,0fffff000h
      add edi,[PTE]
      mov ecx,1024
      cld
      xor eax,eax
      rep stosd
;
; Se tiene un mapa de memoria como el siguiente:
;
; <bochs:11> show "tab"
; cr3: 00112000
; 00000000 - 00000000:   110000 -   110000
; 00011000           :   111000 (  111000) in TLB
; 00012000           :   112000 (  112000) in TLB
; 00013000           :   113000 (  113000) in TLB
; 00112000           :   114000 (  114000) in TLB
;
; De esta forma, a la PTE de la pila se la direcciona mediante
; la página ubicada a partir de 112000 lineal.
;
; Se inicializa la PTE con el puntero a la PILA.
;
      mov eax,[freemem]
      or al,7
      add dword [freemem],1000h
      mov edi,3ffff000h
      shr edi,12
      shl edi,2
      add edi,[PTE]
      mov [edi],eax
;
; Finalmente el mapa de memoria resulta:
;
; <bochs:24> show "tab"
; cr3: 00112000
; 00000000 - 00000000:   110000 -   110000
; 00011000           :   111000 (  111000) in TLB
; 00012000           :   112000 (  112000) in TLB
; 00013000           :   113000 (  113000) in TLB
; 00112000           :   114000 (  114000) in TLB
; 3ffff000 - 3ffff000:   115000 -   115000
;
; Falta inicializar la gdt en su nueva posición de base lineal 0,
; para lo que se utiliza la pila.
;
      push gdt_base
      sub esp,2
      mov word [esp],gdt_end-gdt-1
      lgdt [esp]
      add esp,6
;
; Hasta aquí se realizó el proceso de inicialización de un programa
; con paginación. Si se lo desea continuar, aquí es donde se debe
; agregar código. Este ejemplo termina aquí, sólo continúa con el
; procedimiento para retornar a modo real. Se debe inhabilitar la
; paginación, lo cual se debe hacer desde una dirección lineal que
; físicamente tenga el mismo valor. Por eso se vuelve a armar la misma
; estructura de memoria que existía anteriormente.
;
      mov edi,gdt_basep
      shr edi,12
```

```
      shl edi,2
      add edi,[PTE]
      mov eax,gdt_basep
      or eax,7
      mov ecx,(modo_prot_32_end-modo_prot_32+4095)/4096+2+1
      init_PTE3:
            mov [edi],eax
            add eax,1000h
            add edi,4
            dec ecx
      jnz init_PTE3
;
; Se flushea la TLB.
;
      mov eax,cr3
      mov cr3,eax
;
; Dejando el mapa de memoria:
;
; <bochs:3> show "tab"
; cr3: 00112000
; 00000000 - 00000000:   110000 -   110000
; 00011000           :   111000 (  111000) in TLB
; 00012000 - 00013000:   112000 -   113000
; 00110000 - 00113000:   110000 -   113000
; 3ffff000 - 3ffff000:   115000 -   115000
;
; Se carga nuevamente el GDTR.
;
        push gdt_basep
        sub esp,2
        mov word [esp],gdt_end-gdt-1
        lgdt [esp]
        add esp,6
;
; Se vuelve a las direcciones físicas y lineales equivalentes.
;
      jmp cs_sel_32:subiendo-modo_prot_32+os_basep

subiendo:
;
; Se resetea el bit PE, inhabilitando así la paginación.
;
      mov eax,cr0
      and eax,7fffffffh
      mov cr0,eax
;
; Y se salta al procedimiento de retorno a modo real.
;
      jmp cs_sel_16:return_modo_real_16
modo_prot_32_end:
```