PROYECTO SISTEMAS OPERATIVOS – GESTION DE MEMORIA

Documentación Proyecto MAPA DE BITS

Table of Contents

Gestión de Memoria Física

Autor:

Erwin Meza Vega emezav@gmail.com

Información del Proyecto

En este proyecto se implementa la lógica para gestionar la memoria física, con un mapa de bits en el cual cada bit (unidad de asignación) referencia 4096 (4 KB) bytes de memoria (constante MEMORY_UNIT_SIZE en physmem.c).

Tamaño del Mapa de Bits

Si se tiene una memoria física (RAM) de 4 GB, el número de unidades sería:

```
    4GB / MEMORY_UNIT_SIZE = 4 GB / 4096 = 1048576 = 1 M de unidades.
```

Dado que un byte almacena 8 bits, el número de bytes requerido para todo el mapa de bits se obtiene al dividir el número de unidades entre 8:

```
   1 M / 8 = 131072 = 128 KB.
```

En los ejemplos se usa un mapa de bits de este tamaño (128 KB), para soportar hasta 4 GB de memoria física.

El mapa de bits referenciado con el puntero memory_bitmap (physmem.c) se configura en la dirección de memoria MMAP_LOCATION. Esta área de memoria se encuentra disponible, debido a que el kernel fue cargado por encima del límite de 1 MB.

La información de la memoria disponible se obtiene de la Estructura de Información Multiboot pasada por el GRUB al kernel (por medio del registro EBX en start.S y luego en la variable global multiboot_info_location definida en el archivo kernel.c.

El código principal del kernel en cmain() invoca a la función setup_memory() (physmem.c), la cual toma la variable global multiboot_info_location y obtiene la información del mapa de memoria construido por GRUB. Con este mapa de memoria inicializa el mapa de bits referenciado por la variable memory_bitmap.

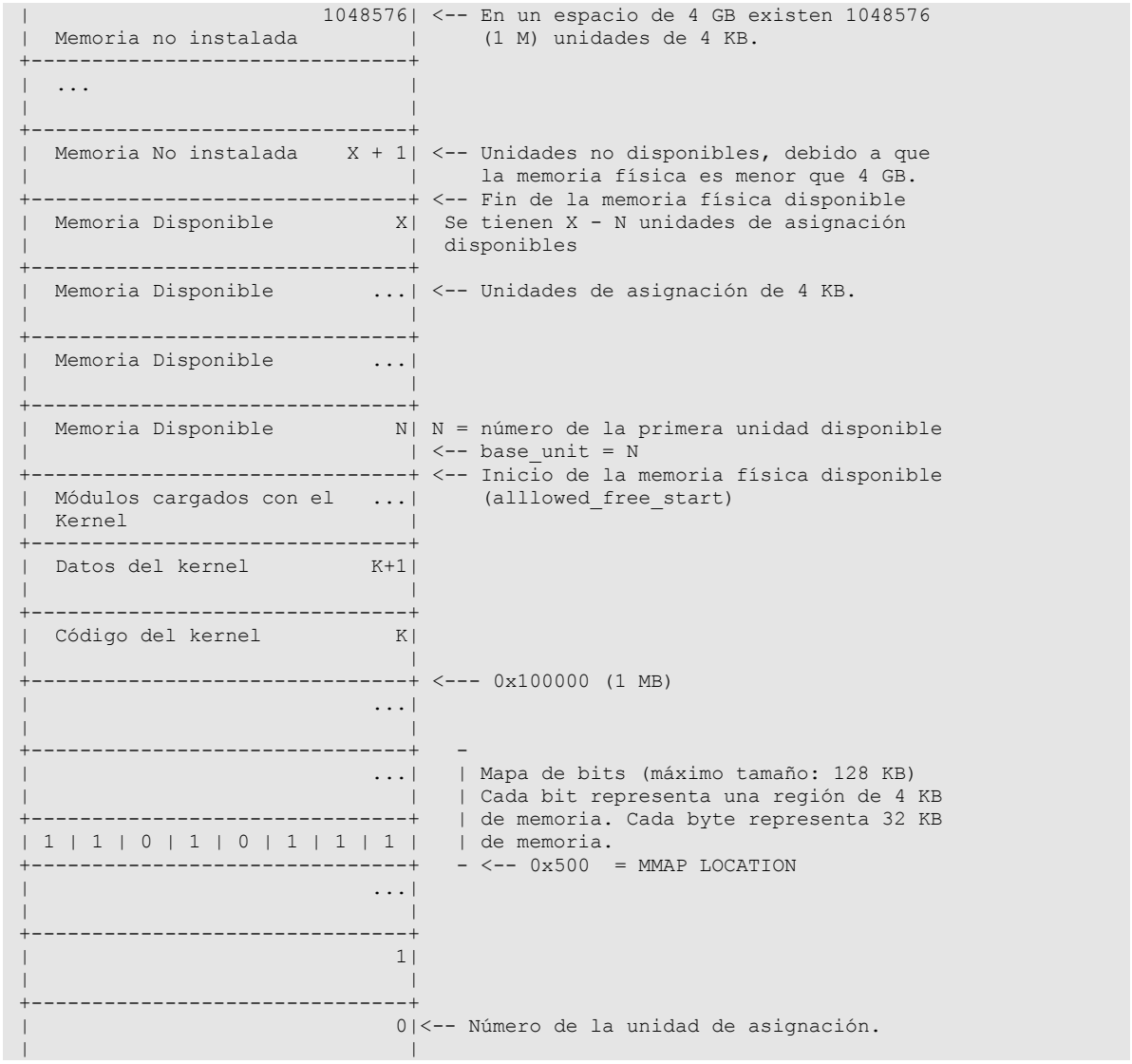
El mapa de bits inicialmente se llena de ceros, para indicar todo el espacio de 4 GB como no disponible. Luego a partir de la información obtenida de GRUB se busca la región de memoria física que se encuentre por encima del kernel y de los módulos cargados, y que esté marcada por GRUB como disponible. Esta región de memoria se marca como memoria disponible dentro del mapa de bits (los bits se establecen en 1).

El inicio de la región de memoria disponible se referencia con la variable global allowed_free_start. Esta variable permite realizar una validación al momento de liberar una unidad de memoria, ya que sólo se puede liberar una unidad que se encuentre por encima de allowed_free_start.

Adicionalmente se establece la variable global base_unit, la cual almacena el número de la unidad de memoria que inicia en allowed_free_start.

La siguiente gráfica ilustra la configuración del mapa de bits en memoria:

```
      Mapa de Bits de la Memoria Física
+------------------------------+ Máximo (teórico) de la memoria (4 GB)
```



Asignación de Memoria

La asignación de memoria se puede realizar de dos formas:

- Asignar una unidad de memoria de 4 KB: Se recorre el mapa de bits buscando un bit que se encuentre en 1 (región disponible). A partir del desplazamiento del bit dentro del mapa de bits, se puede determinar la dirección física que le corresponde. Vea la función allocate_unit() en el archivo physmem.c.
- Asignar una región de memoria de N bytes: Primero se redondea el tamaño solicitado a un múltiplo del tamaño de una unidad de asignación. Luego se busca dentro del mapa de bits un número consecutivo de bits que sumen la cantidad de memoria solicitada. Se retorna la dirección fisica que le corresponde al primer bit en el mapa de bits. Vea la función allocate_unit_region() en el archivo physmem.c.

Liberado de Memoria

Se puede liberar memoria de dos formas:

- Liberar una unidad de memoria: Recibe como parámetro dirección de memoria. Si la dirección de memoria no se encuentra alineada al límite de una unidad de memoria, se toma como dirección el límite

de unidad más cercano por debajo. A partir de la dirección de memoria, se obtiene el bit correspondiente en el mapa de bits, y se marca como disponible. Vea la función free_unit() en el archivo physmem.c.

- Liberar una región de memoria: Recibe como parámetro la dirección de inicio de la región y su tamaño. Si la dirección de inicio de la región no se encuentra alineada al límite de una unidad de memoria, se toma como dirección el límite de unidad más cercano por debajo. Luego, en el mapa de bits se marcan como disponibles los bits que corresponden a las unidades que conforman la región. Vea la función free_region() en el archivo physmem.c.

Ver también:

Páginas relacionadas

Entorno de Desarrollo y Ejecución

Autor:

Erwin Meza Vega emezav@gmail.com

Información del Proyecto : Entorno de Desarrollo y Ejecución

Descripción del Entorno de Desarrollo y Ejecución

Para el desarrollo y la ejecución del software de la Serie Aprendiendo Sistemas Operativos se requiere, además del código, los siguientes programas:

- Editor de texto o IDE (Opcional): Permite la edición de código. Un IDE permite además compilar y ejecutar los ejemplos de forma más ágil. En caso de no contar con un IDE, se puede usar una consola de comandos (Compilación y Ejecución sin un IDE). Los proyectos de la Serie se agrupan como un Workspace y pueden importar directamente en Eclipse.
- Compilador, Ensamblador y Linker (Requerido): Se requiere el compilador GNU de C (**gcc**), el ensamblador (**as**) y el Linker (**ld**), en una versión que permita generar archivos ELF de 32 bits. En sistemas Linux de 32 bits, **gcc** , **as** y **ld** se instalan por defecto o pueden ser instalados fácilmente. En otros sistemas es necesario (compilar o) instalar un "Compilador Cruzado" (cross-compiler) que permita generar archivos ejecutables en formato ELF de 32 bits.
- Utilidades GNU (requeridas): se requiere además otra serie de utilidades GNU como **make** , **dd** , **hexdump** , **addr2line** , y **rm** . Estas utilidades se encuentran disponibles en Linux por defecto, y existen versiones similares para otros sistemas operativos. Por ejemplo en Windows estas utilidades se pueden instalar como parte de MinGW/Msys o Cygwin.
- Utilidad GZIP (Requerida): permite comprimir archivos. Disponible por defecto en Linux, se puede instalar en otros sistemas operativos.
- Emulador de CPU o Máquina Virtual (Requerido) : Un kernel de sistema operativo no puede ser ejecutado directamente en el hardware, si ya existe un sistema operativo ejecutándose. Se requiere un emulador de cpu (como bochs o qemu), o una máquina virtual (como VirtualBox o VMWare) para crear un "computador virtual" en el cual se arranca desde la imagen de disco creada.
- Utilidades para gestión de imágenes de disco (Requerido): Debido a que en la mayoría de los proyectos de la serie se usa una imagen de disco que contiene una partición ext2, es necesario contar con la utilidad **e2fsimage** . Esta utilidad puede ser (compilada o) instalada en Linux y Windows.
- Utilidad para la generación de documentación (Opcional): Cada proyecto de la Serie Aprendiendo Sistemas Operativos permite generar su documentación en formato HTML o RTF gracias al software **Doxygen** . Este software se encuentra disponible para Linux y Windows.

En síntesis, el entorno de desarrollo básico para la Serie consta de:

- Compilador / ensamblador : proporcionados por el sistema operativo (linux) o por sus versiones análogas dentro de MinGW / Msys o en Cygwin (windows).
- Utilidades GNU: proporcionadas por el sistema operativo (linux) o por sus versiones análogas dentro de MinGW / Msys o Cygwin (windows).
- IDE Eclipse: Se usa la Versión Eclipse CDT (C/C++ Development Tools)
- Java JRE: Necesario para ejecutar el IDE Eclipse.
- Emuladores: Qemu o Bochs. Disponibles en Linux y Windows. En Windows, la versión de Bochs que permite usar el depurador gráfico se debe instalar por separado. La Serie incluye también el emulador JPC. Este emulador opera a una velocidad de hasta el 20 % de la velocidad del procesador, por lo cual solo se recomienda su uso si no es posible usar bochs o qemu.

Compilación y Ejecución con Eclipse

Para compilar y ejecutar los ejemplos de Eclipse, se deberá abrir el Workspace (directorio) en el cual se descomprimieron los ejemplos de la serie, por medio de la opción de Eclipse File -> Switch Workspace...

Al seleccionar el directorio que contiene todos los ejemplos puede tener acceso a los ejemplos, cada uno como projecto de Eclipse.

Seleccionar la opción de menú Window -> Show View -> Make Target.

Para cada ejemplo (proyecto) existen los siguientes Make Targets:

- all: Permite compilar el código y crear la imagen de disco
- bochs: Ejecuta el emulador bochs para arrancar la imagen de disco creada. Si algún archivo de código fuente se ha modificado, el código será compilado y la imagen será creada nuevamente.
- bochsdbg: Similar al anterior, pero inicia la versión de Bochs que tiene el depurador gráfico habilitado. Si no se encuentra disponible, se muestra un error.
- jpc: Ejecuta el emulador JPC para arrancar la imagen de disco creada. Si algún archivo de código fuente se ha modificado, el código será compilado y la imagen será creada nuevamente.
- jpcdbg: Similar al anterior, pero inicia un JPC en modo depurador.
- qemu: Ejecuta el emulador qemu para arrancar la imagen de disco creada. Si algún archivo de código fuente se ha modificado, el código será compilado y la imagen será creada nuevamente.

Para limpiar los archivos temporales de compilación y las imágenes de disco, se pueden seleccionar los proyectos en el explorador de proyecto, abrir el menú contextual (click derecho) y seleccionar la opción Clean Project. También se puede seleccionar la opción Project -> Clean.. para limpiar alguno o todos los proyectos.

Compilación y Ejecución sin un IDE

Cada ejemplo puede ser compilado y ejecutado aún si se cumplen todos los requerimientos de software, pero no se cuenta con un IDE. Para ello se deben llevar a cabo los siguientes pasos:

1. Abrir un shell (bash). Este se encuentra disponible en Linux y en Windows mediante MinGW/Msys.
2. Navegar al directorio del ejemplo que se desea ejecutar
3. Ejecutar uno de los siguientes comandos:
    - make : compila el código y crea la imagen de disco
    - make bochs : ejecuta el emulador bochs para que arranque la imagen de disco
    - make bochsdbg : similar al comando anterior, pero ejecuta bochs con el depurador gráfico habilitado, si está instalado. En caso contrario produce error.
    - make jpc: ejecuta el emulador jpc para que arranque la imagen de disco
    - make jpcdbg : ejecuta el depurador jpc para que arranque la imagen de disco
    - make qemu : ejecuta el emulador qemu para que arranque la imagen de disco
    - make clean : borra la imagen de disco y los archivos de compilación

La edición del código se puede realizar con un editor de texto cualquiera, como gedit o vim en Linux, o notepad, pspad o notepad++ en windows.

Desarrollo y Ejecución en Sistemas Operativos de 64 bits

Para sistemas operativos de 64 bits, la estrategia recomendada consiste en instalar VirtualBox o VmWare, instalar un Sistema Operativo de 32 bits como una máquina virtual y dentro de este sistema instalar el software requerido.

Entorno usado para la creación de la Serie

El entorno en el cual se desarrolló la Serie es el siguiente:

- Sistema Operativo: Windows 7
- IDE: Versión de Eclipse CDT. No incluye el JRE de Java.
- Java: J2SE (Java Standard Edition de Oracle)
- Emuladores / Máquinas Virtuales: Qemu, Bochs. Se instaló además una versión de bochs con el depurador gráfico habilitado, cuyo ejecutable se renombró a bochsdbg y se copió en el mismo directorio de la instalación de Bochs. También se incluye el emulador jpc, que solo tiene a Java como requerimiento.
- Compilador, Ensamblador, Linker, Utilidades GNU y otras utilidades requeridas: Proporcionadas por el entorno MinGW/Msys, en la cual se compiló e instaló el compilador cruzado de C (cross-gcc), la utilidad dd y gzip, y la utilidad e2fsimage compilada con CygWin. La instalación de MinGW / Msys incluye las utilidades estándar de linux como cat, dd, rm, mkdir, ls, etc.)

Para garantizar su compatibilidad con un entorno de desarrollo / ejecución basado en Linux, La serie también se probó en el mismo computador usando dos distribuciones de Linux que se ejecutan como máquinas virtuales de VirtualBox:

- Una instalación de Ubuntu 11 (32 bits) en la cual se se instaló y configuró el siguiente software:
  - Paquetes base del sistema: coreutils (incluye cat, dd, rm, ls, mkdir, etc.)
  - Versión completa de Eclipse CDT, que incluye el JRE de java.
  - Paquetes de los emuladores: qemu, qemu-common, qemu-kvm, vgabios, bochs, bochs-wx, bochsbios, bximage. Se incluye la versión de bochs con el depurador gráfico.
  - Compiladores: gcc-4.5-base, gcc-4.5 y sus dependencias.
  - Otras utilidades: binutils (incluye as y ld), e2fsimage (depende de e2fsprogs), gzip, make, grub (para crear la plantilla de la imagen de disco)
- Una instalación de Mandriva Free 2010 (32 bits), en la cual se instaló y configuró el siguiente software:
  - Paquetes base del sistema: coreutils (incluye cat, dd, rm, ls, mkdir, etc.)
  - Versión completa de Eclipse CDT, que incluye el JRE de java.
  - Paquetes para los emuladores: qemu (con sus dependencias), bochs (con sus dependencias)
  - Paquetes de los emuladores: qemu, bochs. El paquete de bochs incluye el depurador gráfico por defecto.
  - Otras utilidades: binutils (incluye as y ld), e2fsimage (instalado desde su código fuente, se debe instalar primero el paquete e2fsprogs-devel) gzip, make, grub (para crear la plantilla de la imagen de disco)
-

Ver también:

http://sourceforge.net/projects/e2fsimage/ Página de la utilidad e2fsimage

Programación de procesadores de arquitectura IA-32

Autor:

Erwin Meza Vega emezav@gmail.com

Ir a: Información del Proyecto

Intel, el fabricante de los procesadores de arquitectura IA-32, ha decidido mantener compatibilidad hacia atrás para permitir que el código desarrollado para procesadores desde 386 o 486 pueda ser ejecutado en procesadores actuales. Esto implica una serie de decisiones de diseño en la estructura interna y en el funcionamiento de los procesadores, que en ciertas ocasiones limita a los programas pero que también ofrece una ventaja competitiva relacionada con la adopción masiva de los procesadores y la posibilidad de ejecutar programas creados para procesadores anteriores en las versiones actuales sin virtualmente ninguna modificación.

Cada procesador actual cuenta con algunas o todas las características de la arquitectura IA-32. Por ejemplo, algunos procesadores actuales poseen múltiples núcleos con registros de 32 bits o múltiples núcleos con registros de 64 bits. Un procesador Intel Core 2 Duo cuenta con dos núcleos con registros de 32 bits, y un procesador Xeon generalmente incluye varios núcleos con registros de 64 bits. La generación actual de procesadores Intel Core (Core I3, Core I5 y Core I7) implementan arquitecturas de 2, 4 y hasta 6 núcleos con registros de 64 bits.

No obstante, para mantener la compatibilidad hacia atrás, todos los procesadores inician en un modo de operación denominado Modo Real (también llamado Modo de Direcciones Real o Real Addres Mode), en el cual se comportan como un procesador 8086 con algunas extensiones que le permiten habilitar el modo de operación en el cual aprovechan todas sus características.

El conocimiento de la arquitectura IA-32 ofrece una posibilidad sin igual para el aprendizaje de la programación básica de una amplia gama de procesadores, desde la programación en modo real hasta la programación en modo protegido, usado por los Sistemas Operativos Modernos.

En los siguientes apartados se presentan los conceptos básicos relacionados con la programación de procesadores de la Arquitectura IA-32

Ver también:

http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html (Enlace externo)

Modos de Operación de procesadores IA-32

Autor:

Erwin Meza Vega emezav@gmail.com

Información del Proyecto : Programación de procesadores de arquitectura IA-32 : Modos de Operación

Los procesadores IA-32 pueden operar en varios modos:

- Modo protegido: Este es el modo nativo del procesador. Aprovecha todas las características de su arquitectura, tales como registros de 32 bits, y el acceso a todo su conjunto de instrucciones y extensiones.
- Modo real: En este modo el procesador se encuentra en un entorno de ejecución en el cual se comporta como un 8086 muy rápido, y sólo tiene acceso a un conjunto limitado de instrucciones que le permiten ejecutar tareas básicas y habilitar el modo protegido. La limitación más notable en este modo consiste en que sólo se puede acceder a los 16 bits menos significativos de los registros de propósito general, y sólo se pueden utilizar los 20 bits menos significativos del bus de direcciones. Esto causa que en modo real solo se pueda acceder a 1 Megabyte de memoria. Como se mencionó anteriormente, todos los procesadores de IA-32 inician en este modo.
- Modo de mantenimiento del sistema: En este modo se puede pasar a un entorno de ejecución limitado, para realizar tareas de mantenimiento o depuración.
- Modo Virtual 8086: Este es un sub-modo al cual se puede acceder cuando el procesador opera en modo protegido. Permite ejecutar código desarrollado para 8086 en un entorno multi-tarea y protegido.
- Modo IA32-e: Para procesadores de 64 bits, además de los modos anteriores existen otros dos sub-modos: modo de compatibilidad y modo de 64 bits. El modo de compatibilidad permite la ejecución de programas desarrollados para modo protegido sin ninguna modificación, y el modo de 64 bits proporciona soporte para acceder a los 64 bits de los registros y un espacio de direcciones mayor que 64 Gigabytes.

Ver también:

BIOS y Arranque del Computador
Paso a Modo Protegido en Procesadores IA-32
http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html (Enlace externo)

Organización de Memoria en Procesadores IA-32

Autor:

Erwin Meza Vega emezav@gmail.com

Información del Proyecto : Programación de procesadores de arquitectura IA-32 : Organización de la Memoria

La memoria en los procesadores de arquitectura IA-32 se puede organizar y manejar en tres formas básicas: Modo Segmentado, Modo Real de Direcciones y Modo Plano. A continuación se muestran los detalles de cada uno de estos modos.

Modo Segmentado

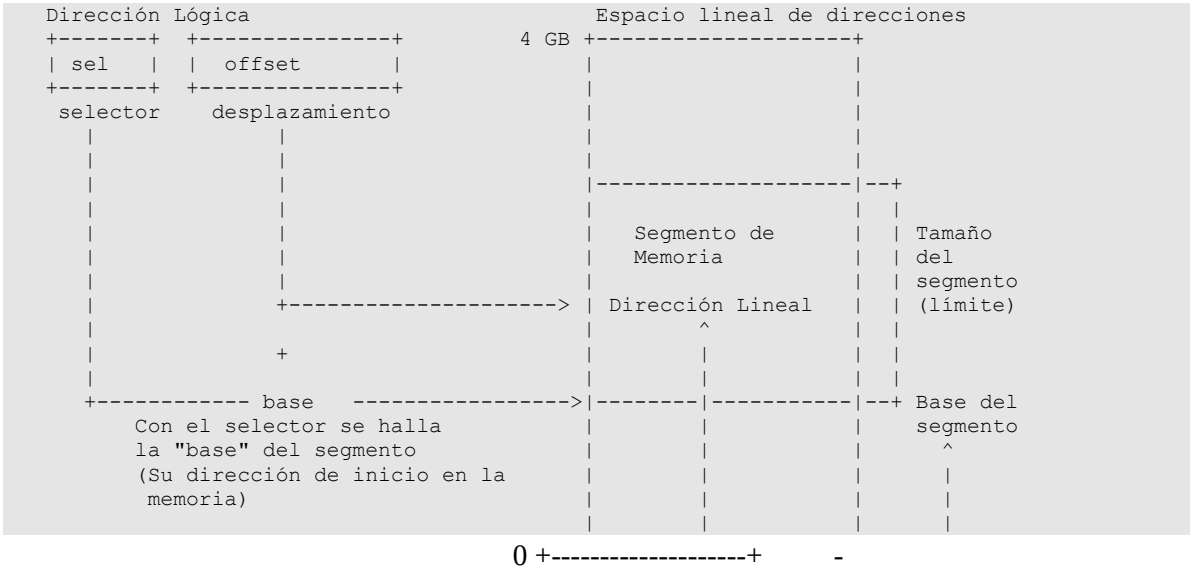
Este es el modo por defecto de organización de memoria. En este modo, la memoria se aprecia como un conjunto de espacios lineales denominados segmentos. Cada segmento puede ser de diferente tipo, siendo los más comunes segmentos de código y datos.

Para referenciar un byte dentro de un segmento se debe usar una dirección lógica , que se compone de un par selector: desplazamiento (offset). El valor del selector se usa como índice en una tabla de descriptores. El descriptor referenciado contiene la base del segmento, es decir la dirección lineal del inicio del segmento.

El desplazamiento (offset) determina el número de bytes que se debe desplazar desde el la base segmento. Así se obtiene una dirección lineal en el espacio de direcciones de memoria.

Si el procesador tiene deshabilitada la paginación (comportamiento por defecto), la dirección lineal es la misma dirección física (En RAM). En el momento de habilitar la paginación, el procesador debe realizar un proceso adicional para traducir la dirección lineal obtenida a una dirección física. El offset se almacena en un registro de propósito general, cuyo tamaño es de 32 bits de esta forma, el tamaño máximo de un segmento es de 4GB.

A continuación se presenta una figura que ilustra cómo realiza este proceso.

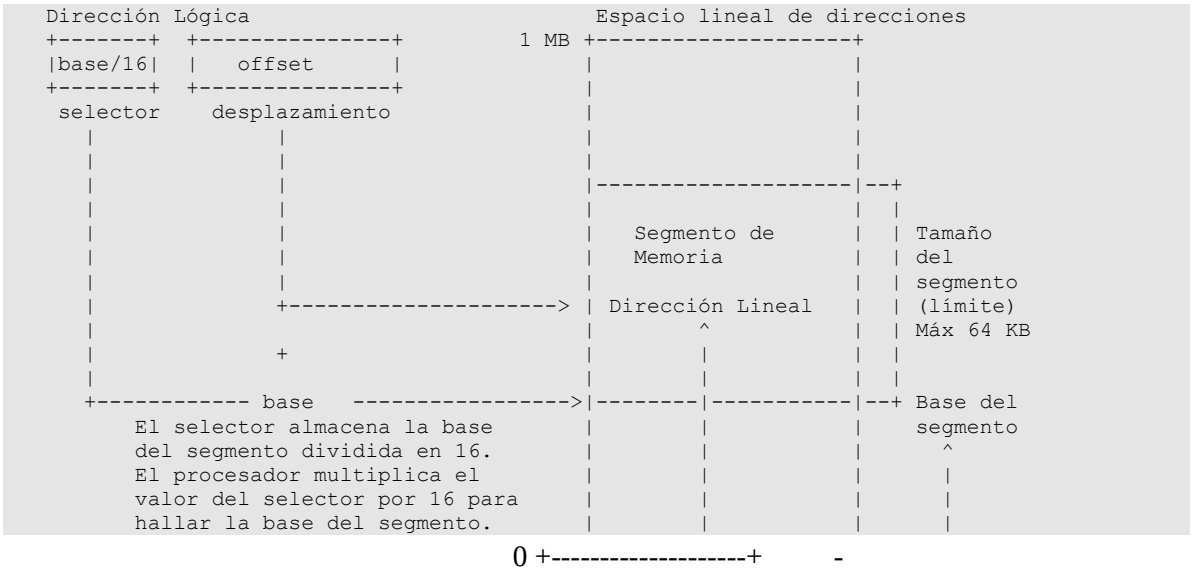


Consulte la página Tabla Global de Descriptores - GDT para más detalles de la traducción de una dirección lógica a una dirección lineal.

Modo Real de Direcciones

El modo real de direcciones es un caso especial del modo segmentado, que se usa cuando el procesador se encuentra operando en Modo Real. Se usa para ofrecer compatibilidad con programas desarrollados para generaciones anteriores de procesadores, que abarcan hasta el propio 8086. En modo real de direcciones el espacio lineal de direcciones se encuentra dividido en segmentos con un tamaño máximo de 64 Kilobytes. Esto se debe a que cuando el computador opera en modo real, sólo es sólo es posible usar los 16 bits menos significativos de los registros de propósito general, que se usan para almacenar el desplazamiento de la dirección lineal dentro del segmento.

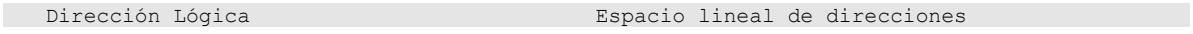
Las direcciones lógicas en modo real también están conformadas por un selector y un offset. Tanto el selector como el desplazamiento tienen un tamaño de 16 bits. Con el fin de permitir el acceso a un espacio de direcciones lineal mayor, el selector almacena la dirección de inicio del segmento dividida en 16. Para traducir una dirección lógica a lineal, el procesador toma el valor del selector y lo multiplica automáticamente por 16, para hallar la base del segmento. Luego a esta base le suma el offset, para obtener una dirección lineal de 20 bits. Así, en modo real sólo se puede acceder al primer MegaByte de memoria. La siguiente figura ilustra el proceso de transformar una dirección lógica a lineal en el modo real de direcciones.

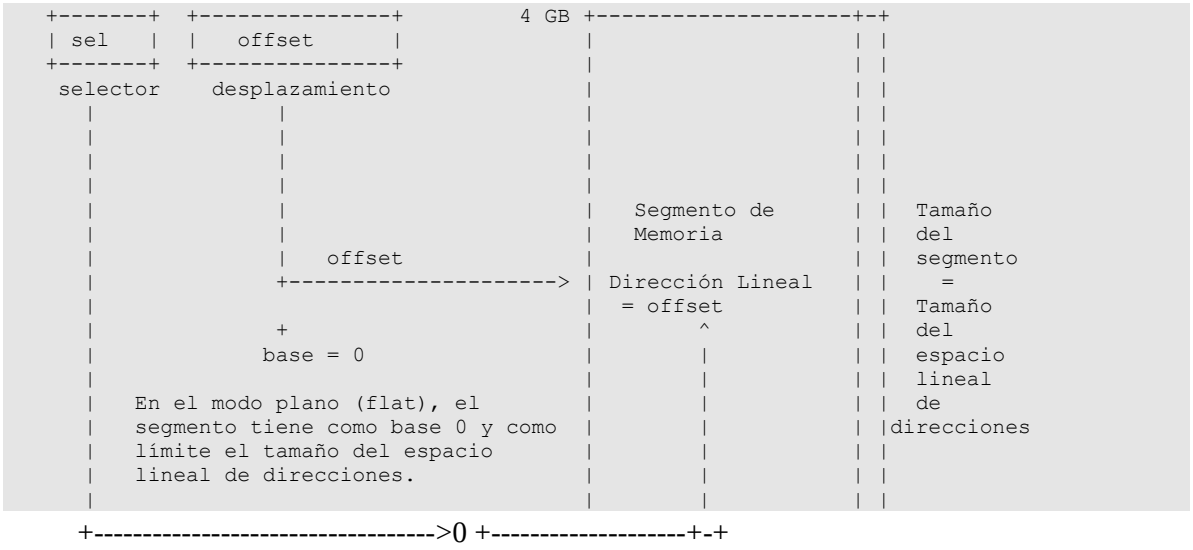


Modo Plano (Flat)

El modo plano es otro caso especial del modo segmentado. La memoria en este modo se presenta como un espacio continuo de direcciones (espacio lineal de direcciones). Para procesadores de 32 bits, este espacio abarca desde el byte 0 hasta el byte 2^32 (4GB). En la práctica, el modo plano se puede activar al definir segmentos que ocupan todo el espacio lineal (con base = 0 y un tamaño igual al máximo tamaño disponible).

Dado que en este modo se puede ignorar la base del segmento (al considerar que siempre inicia en 0), el desplazamiento en una dirección lógica es igual a la dirección lineal (Ver figura).





Ver también:

BIOS y Arranque del Computador
Paso a Modo Protegido en Procesadores IA-32
http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
(Enlace externo)

Entorno de ejecución en IA-32

Autor:

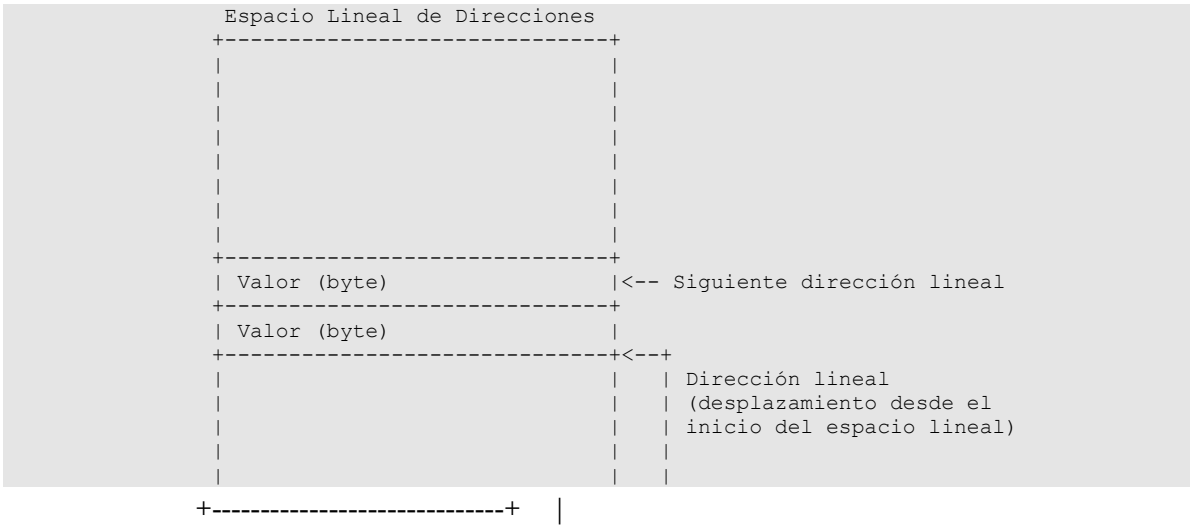
Erwin Meza Vega emezav@gmail.com

Información del Proyecto : Programación de procesadores de arquitectura IA-32 : Entorno de Ejecución

Cualquier programa o tarea a ser ejecutado en un procesador de arquitectura IA-32 cuenta con un entorno de ejecución compuesto por un espacio de direcciones de memoria y un conjunto de registros. A continuación se describen estos componentes.

Espacio Lineal de Direcciones

En la arquitectura IA-32 la memoria puede ser vista como una secuencia lineal (o un arreglo) de bytes, uno tras del otro. A cada byte le corresponde una dirección única (Ver figura).



El código dentro de una tarea o un programa puede referenciar un espacio lineal de direcciones tan grande como lo permitan los registros del procesador. Por ejemplo, en modo real sólo es posible acceder a los 64 KB dentro de un segmento definido (2^16 = 64 KB), y en modo protegido de 32 bits se puede acceder a un espacio lineal de hasta 4 GB (2^32 = 4 GB).

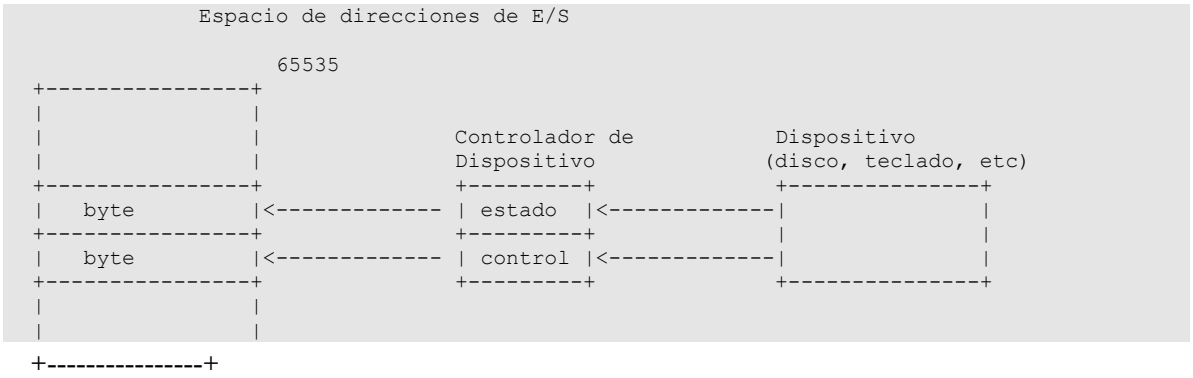
Este espacio lineal puede estar mapeado directamente a la memoria física. Si el procesador cuenta con las extensiones requeridas, es posible acceder a un espacio físico de hasta 64 Gigabytes.

Se debe recordar que la arquitectura IA-32 siempre hace uso de un modelo de memoria segmentado, sin importar su modo de operación (Organización de Memoria en Procesadores IA-32). Los sistemas operativos actuales optan por usar un modelo plano (Flat) en modo protegido, por lo cual pueden tener acceso a todo el espacio lineal de direcciones.

Espacio de Direcciones de Entrada / Salida

Los procesadores IA-32 incluyen otro espacio de direcciones, diferente al espacio lineal de direcciones , llamado espacio de direcciones de Entrada / Salida. A este espacio de 65536 (64K)

direcciones se mapean los registros de los controladores de dispositivos de entrada / salida como el teclado, los discos o el mouse (Ver figura).



El acceso al espacio de direcciones de E/S se realiza a través de un par de instrucciones específicas del procesador (in y out). Al leer o escribir un byte en una dirección de E/S, el byte se transfiere al puerto correspondiente del dispositivo.

Se debe consultar la documentación de cada dispositivo de E/S para determinar cuales son las direcciones de E/S a través de las cuales se puede acceder a los registros de su controlador.

Por ejemplo, el controlador de teclado (8042) tiene asignadas las siguientes direcciones de entrada / salida:

Dirección de E/S	Operación	Descripción
0x60	Lectura	Buffer de entrada
0x60	Escritura	Puerto de comandos
0x64	Lectura	Registro de Estado del teclado
0x64	Escritura	Puerto de comandos

Este controlador deberá ser programado para habilitar la línea de direcciones A20, que en los procesadores actuales se encuentra deshabilitada al inicio para permitir la compatibilidad con programas desarrollados para procesadores anteriores.

Conjunto de Registros IA-32

El procesador cuenta con una serie de registros en los cuales puede almacenar datos. Estos registros pueden ser clasificados en:

- Registros de propósito general: Utilizados para almacenar valores, realizar operaciones aritméticas o lógicas o para referenciar el espacio de direcciones lineal o de E/S. En procesadores IA-32 bits existen ocho (8) registros de propósito general, cada uno de los cuales tiene un tamaño de 32 bits. Estos registros son: EAX, EBX, ECX, EDX, ESI, EDI, ESP y EBP. A pesar que se denominan registros de propósito general, y pueden ser utilizados como tal, estos registros tienen usos especiales para algunas instrucciones del procesador. Ppor ejemplo la instrucción DIV (dividir) hace uso especial de los registros EAX y EDX, dependiendo del tamaño del operando.
- Registros de segmento: Estos registros permiten almacenar apuntadores al espacio de direcciones lineal. Los procesadores IA-32 poseen seis (6) registros de segmento. Estos son: CS (código), DS (datos), ES, FS, GS (datos), y SS (pila). Su uso depende del modo de operación. En modo real, los registros de segmento almacenan un apuntador a la dirección lineal del inicio del segmento dividida en 16. En modo protegido se denominan 'selectores', y contienen un apuntador a una estructura de datos en la cual se describe un segmento de memoria (ver Tabla Global de Descriptores - GDT).
- Registro EFLAGS: Este registro de 32 bits contiene una serie de banderas (flags) que tienen diversos usos. Algunas reflejan el estado del procesador y otras controlan su ejecución. Existen instrucciones

específicas para modificar el valor de EFLAGS. Otras instrucciones modifican el valor de EFLAGS de forma implícita. Por ejemplo, si al realizar una operación aritmética o lógica se obtiene como resultado cero, el bit ZF (Zero Flag) del registro EFLAGS se establece en 1, para indicar esta condición. Esta bandera puede ser chequeada para realizar algún tipo de opración o salto dentro del código.

- Registro EIP: Este registro almacena el apuntador a la dirección lineal de la siguiente instrucción que el procesador debe ejecutar. Esta dirección es relativa al segmento al cual se referencia con el registro de segmento CS.
- Registros de control: El procesador posee cinco (5) registros de control CR0 a CR5. Estos registros junto con EFLAGS controlan la ejecución del procesador.
- Registros para el control de la memoria: Estos registros apuntan a las estructura de datos requeridas para el funcionamiento del procesador en modo protegido. Ellos son: GDTR, IDTR, TR y LDTR.
- Registros de depuración: Estos registros contienen información que puede ser usada para depurar el código que está ejecutando el procesador. Los procesadores IA-32 cuentan con ocho (8) registros de depuración, DR0 a DR7.
- Registros específicos: Cada variante de procesador IA-32 incluye otros registros, tales como los registros MMX, los registros de la unidad de punto flotante (FPU) entre otros.

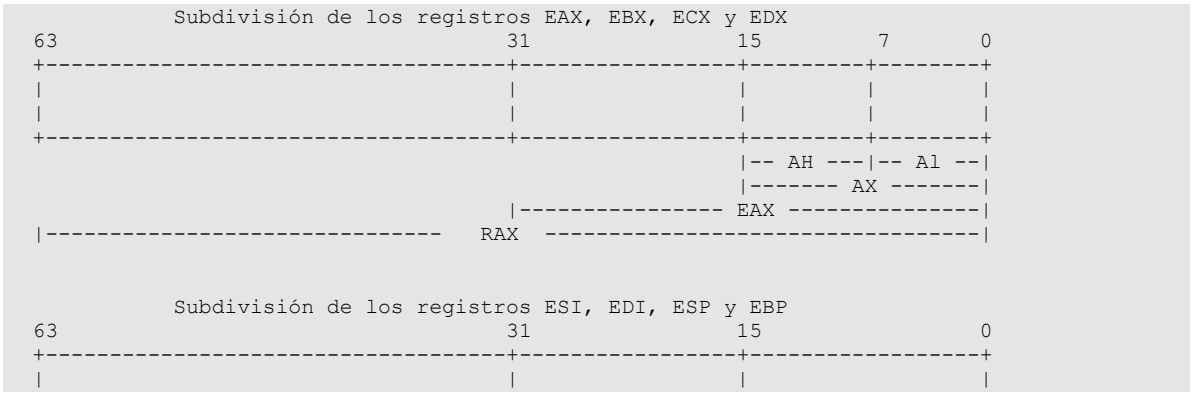
Algunos registros de propósito general pueden ser sub-divididos en registros más pequeños a los cuales se puede tener acceso. Esto permite la compatibilidad con programas diseñados para procesadores anteriores.

A continuación se presentan las posibles sub-divisiones de los registros de propósito general, considerando procesadores de hasta 64 bits:

64 bits	32 bits	16 bits	8 bits	8 bits
RAX	EAX	AX	AH	AL
RBX	EBX	BX	BH	BL
RCX	ECX	CX	CH	CL
RDX	EDX	DX	DH	DL
RSI	ESI	SI	No accesible	
RDI	EDI	DI	No accesible	
RSP	ESP	SP	No accesible	
RBP	EBP	BP	No accesible	

A nivel de programación, es posible acceder a cada uno de estos sub-registros de acuerdo con el modo de operación. Por ejemplo, para modo de direcciones real, es posible usar los registros de 8 bits y los registros de 16 bits. En modo protegido se puede usar los registros de 8, 16 y 32 bits. Si el procesador cuenta con registros de 64 bits y se encuentra en el modo de 64 bits, es posible acceder a los registros de 8, 16, 32 y 64 bits.

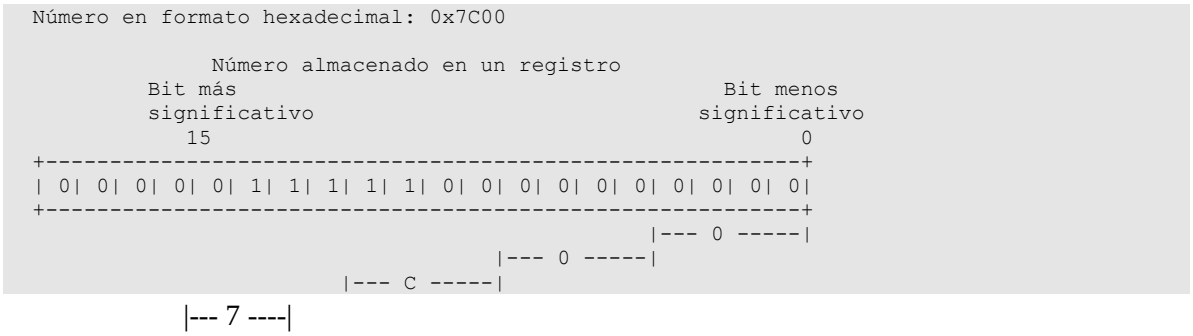
La siguiente figura muestra como se encuentran dispuestos los bits de los registros de propósito general. Los registros EBX, ECX y EDX se encuentran dispuestos de la misma forma que EAX. Los registros EDI, ESP Y EBP se disponen de la misma forma que ESI.



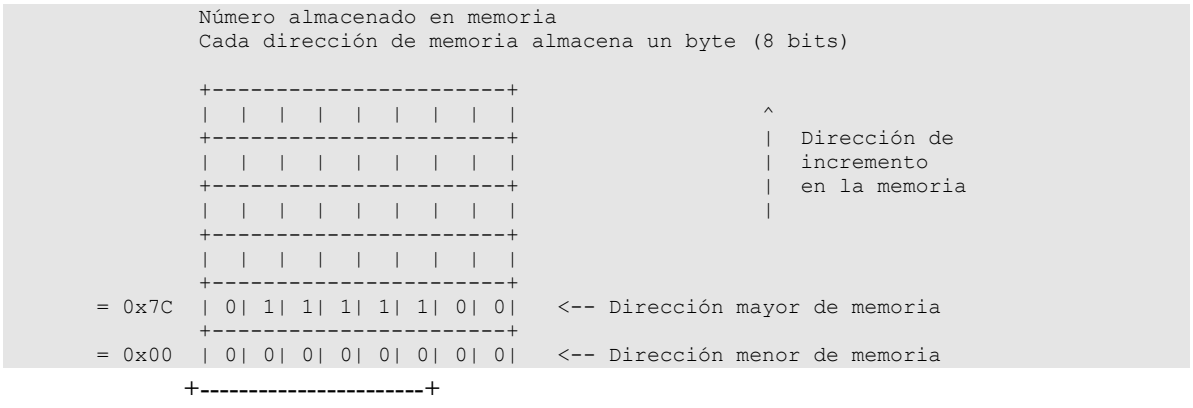
Formato de almacenamiento de datos

El formato de almacenamiento de la arquitectura IA-32 es Little-Endian, lo cual significa que los bits menos significativos de un número se almacenan en las posiciones menores de la memoria y de los registros, y los bits más significativos se almacenan en posiciones superiores. La siguiente figura muestra cómo se almacena el número 0x7C00 en un registro y en la memoria.

El número 0x7C00 almacenado en un registro se ve así:



El mismo número almacenado en memoria se ve así:

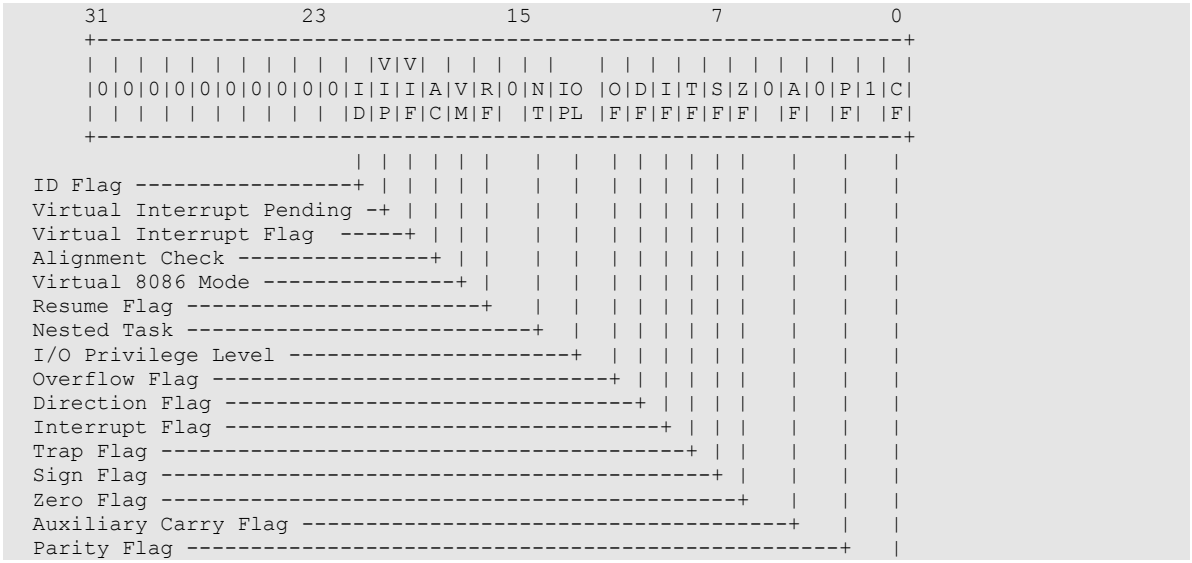


Registros principales de IA-32

Si bien todos los registros de la arquitectura IA-32 son importantes, existen algunos que determinan el modo de ejecución el estado del procesador. A continuación se presenta el formato de estos registros.

Registro EFLAGS

Este registro almacena información del estado del procesador y configura su ejecución. Tiene el formato que se presenta en la siguiente figura.

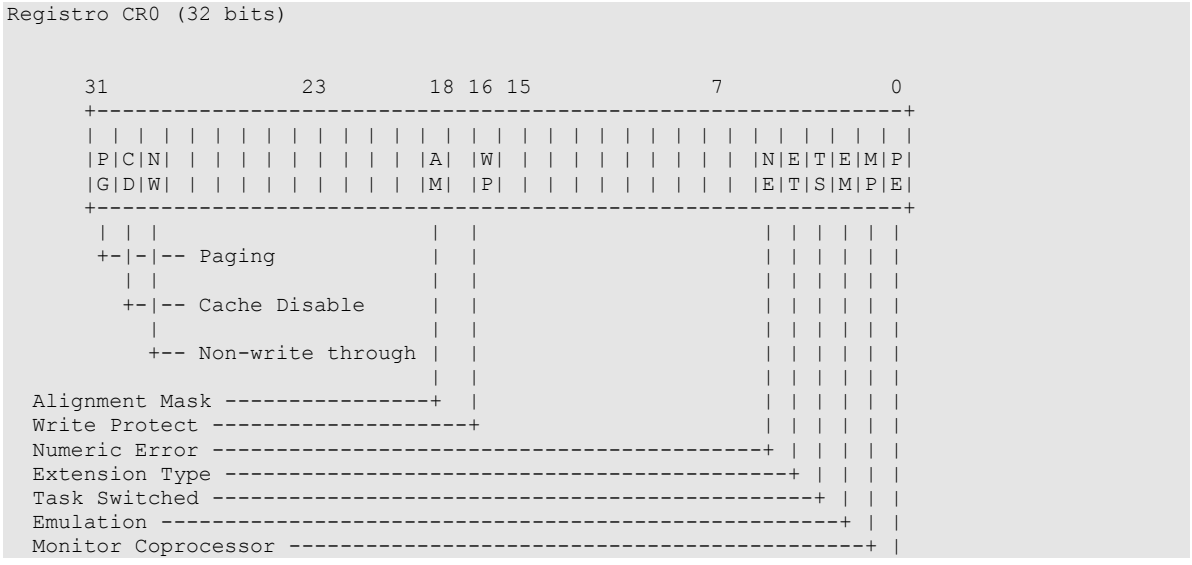


Los bits del registro EFLAGS se pueden clasificar en:

- Bits de estado: Reflejan el estado actual del procesador. Son bits de estado: OF, SF, ZF, AF y PF.
- Bits de control: Controlan de alguna forma la ejecución del procesador. Dentro de EFLAGS se encuentra el bit DF, que permite controlar la dirección de avance en las operaciones sobre cadenas de caracteres.
- Bits del sistema: Los bits ID, VIP, VIF, AC, VM, RF, NT, IOPL, IF y TF son usados por el procesador para determinar condiciones en su ejecución, o para habilitar / deshabilitar determinadas características. Por ejemplo, estableciendo el bit IF en 1 se habilitan las interrupciones, mientras un valor de 0 en este bit deshabilita las interrupciones.
- Bits reservados: Estos bits se reservan por la arquitectura IA-32 para futura expansión. Deben permanecer con los valores que se muestran en la figura (cero o uno). No se deben usar, ya que es posible que en versiones posteriores de los procesadores IA-32 tengan un significado específico.

Registro CR0

A continuación se ilustra el Control Register 0 (CR0). Este registro controla aspectos vitales de la ejecución, como el modo protegido y la paginación.



Protection Enable --+

Los bits más importantes de CR0 desde el punto de vista de programación son el bit 0 (Protection Enable – PE), y el bit 31 (Paging – PG). Estos permiten habilitar el modo protegido y la paginación, respectivamente.

No obstante antes de pasar a modo protegido y de habilitar la paginación se deben configurar unas estructuras de datos que controlan la ejecución del procesador.

Para habilitar el modo protegido se deberá tener configurada de antemano una Tabla Global de Descriptores - GDT. Esta tabla deberá ser configurada por el cargador de arranque o por el código inicial del kernel.

Ver también:

Ensamblador para procesadores IA-32
Modos de Operación de procesadores IA-32
BIOS y Arranque del Computador
Paso a Modo Protegido en Procesadores IA-32
http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
(Enlace externo)

Paso a Modo Protegido en Procesadores IA-32

Autor:

Erwin Meza Vega emezav@gmail.com

Información del Proyecto : Programación de procesadores de arquitectura IA-32 : Paso a Modo Protegido

El manual de intel "Intel Architecture Software Developer'sManual , Volume 3: System Programming" (Codigo 243192), en su sección 8.8.1 Switching to Protected Mode, especifica el procedimiento requerido para pasar de modo real a modo protegido. Este procedimiento asegura compatibilidad con todos los procesadores Intel de arquitectura IA-32. Los pasos son:

1. Deshabilitar las interrupciones por medio de la instruccion CLI
   - No incluido en el manual de Intel: Por razones históricas, cuando el procesador se encuentra en modo real, su línea de direcciones 20 (A20 Gate) se encuentra deshabilitada, lo cual causa que cualquier dirección de memoria mayor a 2^20 (1MB), sea truncada al limite de 1 MB. Debido a que en modo protegido es necesario usar los 32 bits del bus de direcciones para referenciar hasta 2^32 = 4GB de memoria, es necesario habilitar la linea de direcciones A20. Esto se logra por medio del controlador 8042 (teclado), el cual se puede encontrar fisicamente en la board o integrado dentro de su funcionalidad.
2. Ejecutar LGDT para cargar una tabla global de descriptores (GDT) válida.
3. Ejecutar una instruccion MOV para establecer el bit 0 del registro de control CR0 (PE = Protection Enable).
4. Inmediatamente después de establecer el bit PE en CR0, se debe realizar un jmp/call para limpiar la cola de pre-fetch. Si se habilitó paginacion (bit PG = Page Enable), las instrucciones MOV y JMP/CALL deberan estar almacenadas en una página cuya dirección virtual sea idéntica a la dirección fisica.
5. Si se va a utilizar un LDT, se debe cargar por medio de la instrucción LLDT.
6. Ejecutar una instrucción LTR para cargar el Task Register con el selector de la tarea inicial, o de un área de memoria escribible que pueda ser utilizada para almacenar información de TSS en un Task Switch (cambio de contexto).
7. Luego de entrar en modo protegido, los registros de segmento (DS, ES, FS, GS y SS) aún contienen los valores que tenían en modo real (el jmp del paso 4 sólo modifica CS). Se deben cargar selectores validos en estos registros, o el selector nulo (0x0).
8. Ejecutar LIDT para cargar una Tabla de Descriptores de Interrupcion válida. La IDT deberá contener entradas válidas al menos para las 32 primeras entradas, que corresponden a las excepciones de la arquitectura IA-32.
9. Habilitar las interrupciones por medio de la instrucción STI.

Cuando se usa un cargador compatible con la Especificación Multiboot para cargar el kernel (como GRUB), se cuenta con las siguientes facilidades:

1. La línea de direcciones A20 ya se encuentra activada.
2. Ya se ha configurado una GDT temporal. La especificación Multiboot insiste en que se deberá configurar y cargar una GDT propia del kernel tan pronto como sea necesario.
3. El kernel debe configurar una pila (los registros SS y ESP) tan pronto como sea posible.

Ver también:

Tabla Global de Descriptores - GDT
http://www.gnu.org/software/grub/manual/multiboot/multiboot.html Especificación Multiboot (Enlace externo)
http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html (Enlace externo)

Tabla Global de Descriptores - GDT

Autor:

Erwin Meza Vega emezav@gmail.com

Información del Proyecto : Programación de procesadores de arquitectura IA-32 : GDT

La GDT es una tabla que usa el procesador cuando se encuentra en modo protegido para traducir direcciones lógicas a direcciones lineales de memoria. Si la paginación se encuentra deshabilitada, la dirección lineal hallada se toma como una dirección física directamente.

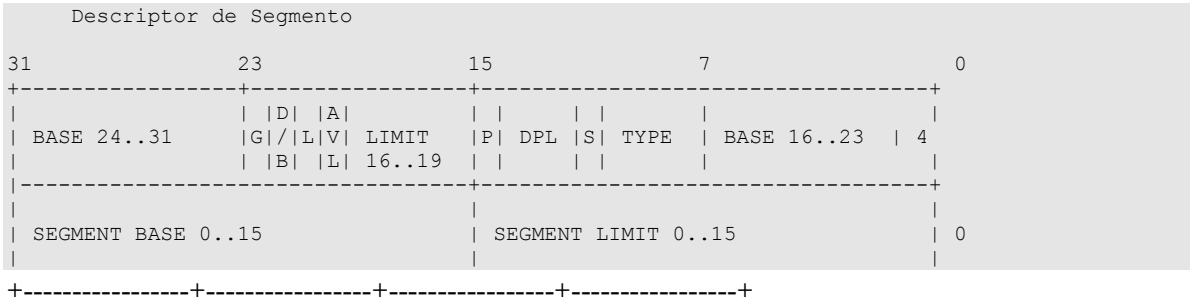
Nota:

Antes de pasar a modo protegido, se debe configurar una GDT que contenga al menos dos descriptores: un descriptor de segmento de código y un descriptor de segmento de datos.

Descriptores de segmento

La GDT está compuesta por uno o más descriptores de segmento. Cada descriptor de segmento es una estructura de datos que contiene información de la ubicación de un segmento en memoria, su tipo y algunas opciones de protección.

Un descriptor de segmento ocupa 8 bytes (64 bits), distribuidos de la siguiente forma:



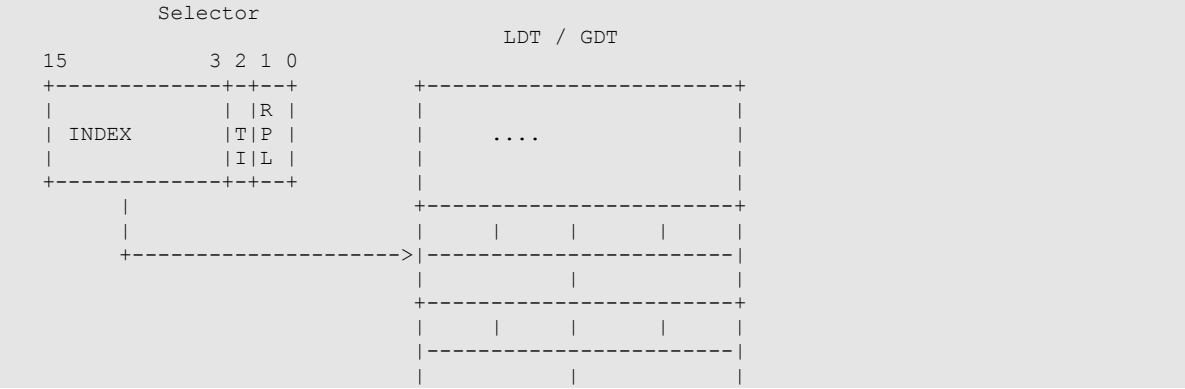
En el Manual de Intel Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1 (pág 95) describe cada uno de los campos de un descriptor. A continuación se presenta una breve reseña de estos campos.

- Limit (20 bits): Define el tamaño del segmento. Para calcular el tamaño en bytes de un segmento se toma el valor de Límite y se verifica el bit G (Granularidad). Si G = 0, el valor de límite se expresa en bytes. Si G = 1, el valor de límite expresa en unidades de memoria de 4 KB. De esta forma, el máximo tamaño de un segmento, si todos los bits de Límite se encuentran en 1 y el bit G se encuentra en 1 es de 4 GB (2 ^ 20 * 4 KB = 2 ^ 20 * 2 ^ 12 bytes = 2 ^ 32 = 4 GB). Los bits que conforman el valor de Limit se encuentran dispersos dentro del descriptor.
- Base (32 bits): Define la dirección lineal en la cual inicia el segmento en el espacio lineal de direcciones. Dado que se dispone de 32 bits para Base, el segmento puede empezar en cualquier dirección en el espacio lineal. Los bits que conforman el valor de Base se encuentran dispersos dentro del descriptor.
- Type: Permite especificar el tipo de segmento y especifica los tipos de acceso permitidos. Por ejemplo, se puede definir un segmento de datos de sólo lectura o un segmento de datos de lectura / escritura.
- S: Determina el tipo de descriptor de segmento. Si el descriptor es para un segmento de código o datos, S = 1. Si el descriptor es de otro tipo (descriptor del sistema), S = 0.
- DPL: Especifica el nivel de privilegios del segmento. Debido a que sólo se tienen 2 bits para DPL, los privilegios válidos son 0, 1 2 y 3. El nivel de mayor privilegios es 0.
- P: Permite verificar si el segmento está presente en memoria (P = 1) o no (P = 0)
- AVL: Este bit se encuentra disponible para uso del sistema.

- L: Este bit debe ser cero para procesadores de 32 bits. Sólo se usa en procesadores de 63 bits, y permite definir que se está describiendo un segmento de código de 64 bits.
- D/B — Default operation size: Permite definir el tamaño de operando por defecto del segmento. Para segmentos de código y datos de 32 bits este bit se deberá establecer en 1, mientras que para segmentos de 16 bits se deberá establecer en 0.

Selectores

Los descriptores que se almacenan en la GDT se referencian a través de **Selectores** . Estos permiten establecer la posición (el índice) en la cual se encuentra en descriptor de segmento, la Tabla de Descriptores en la cual se encuentra y los privilegios de acceso.



En donde:

- INDEX : Posición en la tabla en la cual se encuentra el descriptor.
- TI (Table Indicator) : Indicador de tabla dentro de la cual se debe buscar el descriptor (0 = GDT, 1 = LDT)
- RPL (Requestor Privilege Level) : Nivel de privilegios del solicitante.

En modo protegido de 32 bits, la primera entrada de la GDT es nula. Así, si un selector (registro de segmento) puede tomar el valor de 0 para indicar que no ha sido configurado.

La LDT es similar en formato a la GDT, pero se diferencia en:

- Es usada por una tarea para gestionar sus segmentos
- Si primera entrada, a diferencia de la GDT sí puede ser utilizada.
- Si una tarea tiene definida su GDT, debe existir una entrada dentro de la GDT que describa la LDT.

Direcciones lógicas en Modo Real y Modo Protegido

Si importar su modo de operación, los procesadores IA-32 usan siempre direcciones lógicas que se obtienen al combinar dos tipos de registros:

- Un registro de segmento (CS, DS, ES, FS, GS y SS)
- Un registro de propósito general, ESB, EBP, EIP o un valor inmediato, dependiendo de la instrucciÓn.

Por ejemplo: CS:EIP, DS:SI, ES:DI.

Estas direcciones lógicas de memoria se transforman en direcciones lineales de formas diferentes en modo real y modo protegido:

En modo real, los registros de segmento (CS, DS, ES, FS, GS y SS) almacenan la dirección base del segmento dividida en 16. El registro de propósito general contiene el desplazamiento a partir de la dirección base del segmento.

La dirección lineal se calcula de la siguiente forma:

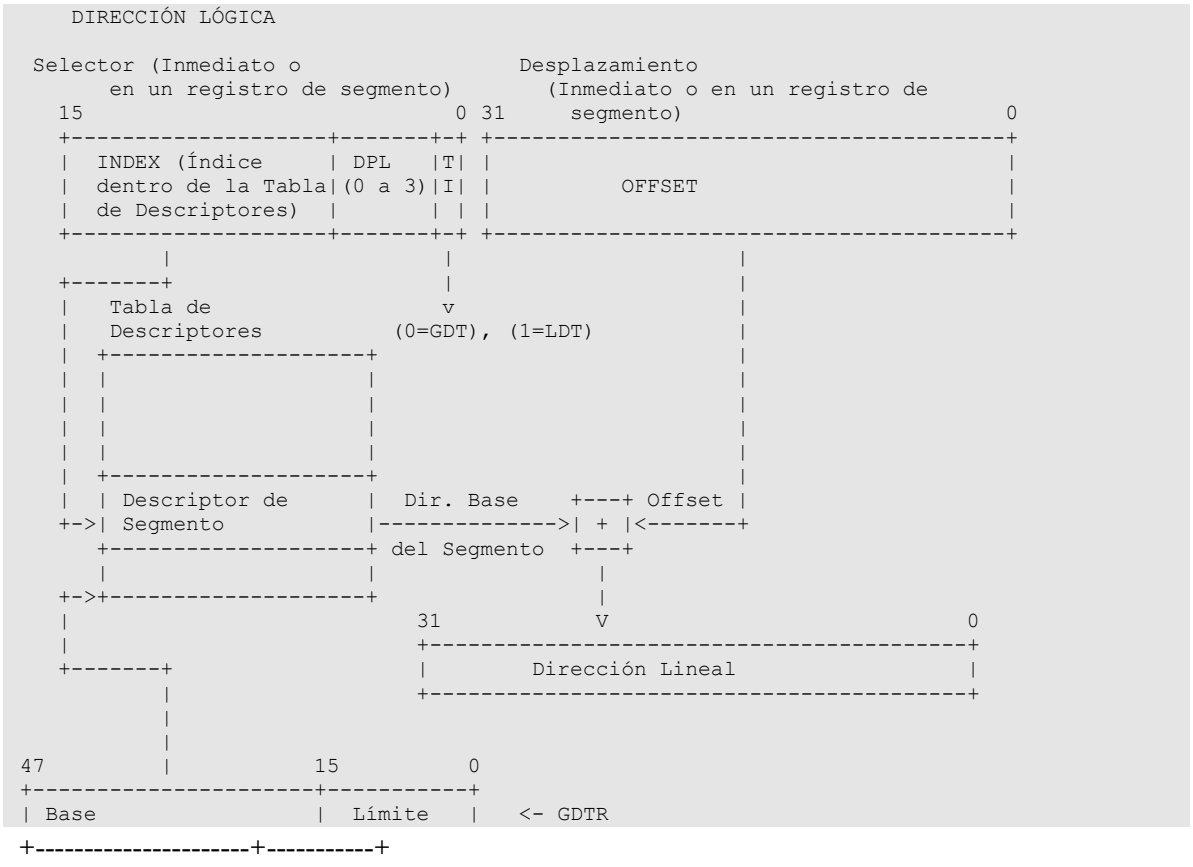
dir. lineal = segmento x 16 + offset

En modo protegido, los registros de segmento (CS, DS, ES, FS, GS y SS) almacenan el selector que referencia a un descriptor de segmento dentro de la GDT o la LDT. El registro de propósito general contiene el desplazamiento a partir de la dirección base del segmento.

Para calcular una dirección lineal se realiza el siguiente proceso:

1. Con el atributo TI del selector, se determina si el descriptor de segmento se buscará en la GDT o la LDT (GDT por defecto).
2. Con el atributo RPL se determina si se tiene permiso para acceder al descriptor.
3. Con el atributo INDEX del selector, se busca el descriptor correspondiente en la tabla especificada.
4. Del descriptor se obtiene la dirección base del segmento, la cual se suma al desplazamiento para obtener la dirección lineal.

Gráficamente la traducción de una dirección lógica a una dirección lineal en modo protegido se realiza de la siguiente forma:



Ejemplos de GDT

Como se mencionó anteriormente, antes de habilitar el modo protegido del procesador es necesario configurar y cargar la GDT. A continuación se presenta el contenido de una GDT, suponiendo que

se definen los segmentos de código y datos para el kernel en modo plano (flat) con nivel de privilegios 0.

Para este caso, la GDT contiene tres entradas (tres descriptores de segmento):

- El primer descriptor es nulo (todos sus campos son cero). Requerido por la arquitectura IA-32.
- El segundo descriptor se usa para describir el segmento de código del kernel, con los siguientes parámetros:
  - Base: 0 = 0x00000000
  - Límite: 4 GB = 0xFFFFF, Bit G = 1
  - Nivel de Privilegios: 0
  - Tipo de Segmento: Código, Lectura / Ejecución
- El tercer descriptor se usa para describir el segmento de datos del kernel, con los siguientes parámetros:
  - Base: 0 = 0x00000000
  - Límite: 4 GB = 0xFFFFF, Bit G = 1
  - Nivel de Privilegios: 0
  - Tipo de Segmento: Datos, Lectura / Escritura
-

El orden de los descriptores (después del primero) puede ser cualquiera, aunque generalmente se define primero el descriptor de segmento de datos y luego el descriptor de segmento de código.

A continuación se muestra el contenido de los tres descriptores de segmento.

- Descriptor de segmento nulo:

```
 Descriptor nulo (Primera entrada de la GDT)


+-----------------+-----------------+-----------------------------------+
|                 | |D| |A|         | |     | |       |                 |
| BASE 24..31     |G|/|L|V| LIMIT   |P| DPL |S| TYPE  | BASE 16..23  | 4
|                 | |B| |L| 16..19  | |     | |       |                 |
|-----------------------------------+-----------------------------------+
|                                   |                                   |
| SEGMENT BASE 0..15                | SEGMENT LIMIT 0..15               | 0
|                                   |                                   |
+-----------------+-----------------+-----------------+-----------------+

 31               23                15                7                 0
+-----------------+-----------------+-----------------------------------+
|                 | | | | |         | |     | |       |                 |
| 0 0 0 0 0 0 0 0 |0|0|0|0| 0 0 0 0 |0| 0 0 |0|0 0 0 0| 0 0 0 0 0 0 0 0 | 4
|                 | | | | |         | |     | |       |                 |
|-----------------------------------+-----------------------------------+
|                                   |                                   |
|  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  |  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  | 0
|                                   |                                   |
```

- +----------------+----------------+----------------+----------------+
  - El selector correspondiente a este descriptor es:

```
         15           3 2 1 0
         +--------------+-+--+
         |0000000000000|0|00|   Selector Nulo = 0x00
```

  - +-------------+-+--+ Índice = 0, TI = 0 (GDT), DPL = 0x00
- Descriptor de segmento de código:

```
 Descriptor de Segmento de Código (Segunda entrada en la GDT)

 +-----------------+-----------------+-----------------------------------+
|                 | |D| |A|         | |     | |       |                 |
| BASE 24..31     |G|/|L|V| LIMIT   |P| DPL |S| TYPE  | BASE 16..23  | 4
|                 | |B| |L| 16..19  | |     | |       |                 |
|-----------------------------------+-----------------------------------+
|                                   |                                   |
| SEGMENT BASE 0..15                | SEGMENT LIMIT 0..15               | 0
|                                   |                                   |
+-----------------+-----------------+-----------------+-----------------+
```

```
 31                23                15                7                  0
+-----------------+-----------------+-----------------------------------+
|                 | | | | |         | |     | |       |                   |
| 1 1 1 1 1 1 1 1 |1|1|0|0| 1 1 1 1 |1| 0 0 |1|1 0 1 0| 1 1 1 1 1 1 1 1 | 4
|                 | | | | |         | |     | |       |                   |
|-----------------------------------+-----------------------------------+
|                                   |                                   |
|  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1  |  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1  | 0
|                                   |                                   |
+-----------------+-----------------+-----------------+-----------------+

 Observe que el campo Type tiene valor 1010, definido en el manual de
 Intel como Segmento de Código, Lectura y Ejecución. Consulte la tabla 3-1
```

- en ese documento.
  - El selector correspondiente a este descriptor es:

```
         15           3 2 1 0
         +-------------+-+--+
         |0000000000001|0|00|   Selector de Código: 1000 = 0x08
```

  - +-------------+-+--+ Índice = 0, TI = 0 (GDT), DPL = 0x00
- Descriptor de segmento de datos:

```
 Descriptor de Segmento de Datos (Tercera entrada en la GDT)

 +-----------------+-----------------+-----------------------------------+
|                 | |D| |A|         | |     | |       |                   |
| BASE 24..31     |G|/|L|V| LIMIT   |P| DPL |S| TYPE  | BASE 16..23  | 4
|                 | |B| |L| 16..19  | |     | |       |                   |
|-----------------------------------+-----------------------------------+
|                                   |                                   |
| SEGMENT BASE 0..15                | SEGMENT LIMIT 0..15               | 0
|                                   |                                   |
+-----------------+-----------------+-----------------+-----------------+

 31                23                15                7                  0
+-----------------+-----------------+-----------------------------------+
|                 | | | | |         | |     | |       |                   |
| 1 1 1 1 1 1 1 1 |1|1|0|0| 1 1 1 1 |1| 0 0 |1|0 0 1 0| 1 1 1 1 1 1 1 1 | 4
|                 | | | | |         | |     | |       |                   |
|-----------------------------------+-----------------------------------+
|                                   |                                   |
|  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1  |  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1  | 0
|                                   |                                   |
+-----------------+-----------------+-----------------+-----------------+

 Observe que el campo Type tiene valor 0010, definido en el manual de
 Intel como Segmento de Datos, Lectura y Escritura. Consulte la tabla 3-1
```

- en ese documento.
  - El selector correspondiente a este descriptor es:

```
         15           3 2 1 0
         +-------------+-+--+
         |0000000000010|0|00|   Selector de Código: 10000 = 0x10
```

  - +-------------+-+--+ Índice = 2, TI = 0 (GDT), DPL = 0x00
-

Representación de la GDT en Lenguaje Ensamblador

La GDT no es más que un arreglo o secuencia de descriptores, uno tras de otros en memoria. Por lo tanto, puede ser expresada en lenguaje ensamblador de la siguiente forma:

```
gdt:
    /* La primera entrada del gdt debe ser nula */
   .word 0x0000
   .word 0x0000
   .byte 0x00
   .byte 0x00
   .byte 0x00
   .byte 0x00
```

```
.word 0xFFFF  /* Limite 0..15 = FFFF */
.word 0x0000  /* Base 0..15 = 0000 */
.byte 0x00    /* Base 16..23 = 00 */
.byte 0x9A    /* 10011010 P=1, DPL=0, S=1, Tipo: codigo, read/execute,
     non conforming */
.byte 0xCF /* 11001111 G=1, D/B=1 (32 bits), L=0, AVL=0, Limite 16..19=F */
.byte 0x00 /* Base 24..31 = 00 */

.word 0xFFFF  /*Limite 0..15 = FFFF */
.word 0x0000  /*Base 0..15 = 0000 */
.byte 0x00    /*Base 16..23 = 00 */
.byte 0x92  /*10010010 P=1, DPL=0, S=1, Tipo: datos, read/write */
.byte 0xCF /*11001111  G=1, D/B=1 (32 bits) , L=0, AVL=0, Limite 16..19=F  */
```

.byte 0x00 /*Base 24..31 = 00 */

Representación de la GDT en lenguaje C

En Lenguaje C es necesario definir una estructura de datos que agrupe los campos de un descriptor de segmento, y luego un arreglo que contenga descriptores. Existe muchas representaciones diferentes, a continuación se muestran algunos ejemplos:

1. Una representación de descriptor de segmento en la cual se usan enteros de 32 bits:

```
struct gdt descriptor {
 /* Bits menos significativos del descriptor. Agrupan
  * Limite 0..15 y Base 0..15 del descriptor */
  unsigned int low : 32;
  /* Bits más significativos del descriptor. Agrupan Base 16..23, Tipo,
   * S, DPL, P, Límite, AVL, L, D/B, G y Base 24..31 */
  unsigned int high: 32;
}  attribute  ((packed));

/* Definición de la GDT */
```

struct gdt[MAX_GDT_ENTRIES] __attribute__((aligned(8))); La documentación de IA-32 exige que la GDT se encuentre alineada a un límite de 8 bytes. Para esta representación de descriptores de segmento el código de inicialización puede ser el siguiente:

```
   /* Inicializar la primera entrada en nulo*/
   gdt[0].low = 0;
   gdt[0].high = 0;

   /* Inicializar la segunda entrada con un descriptor de segmento de
      código flat, nivel de privilegios 0 */
   gdt[1].low = 0x0000FFFF;
   gdt[1].high = 0x00CF9A00;

   /* Inicializar la tercera entrada con un descriptor de segmento de datos
   flat, nivel de privilegios 0 */
   gdt[2].low = 0x0000FFFF;
   gdt[2].high = 0x00CF9200;
```

2. Una representación de descriptor de segmento en la cual los campos se especifican y se acceden por separado:

```
struct gdt descriptor {
 unsigned limit_low   : 16, /* limite 0..15*/
       base low   : 16, /* base 0..15 */
       base_middle  : 8,  /* base 16..23*/
       type     : 4,  /* Informacion de acceso */
       code_or_data  : 1, /* 1 = codigo o datos, 0 = sistema */
       dpl    : 2, /* Nivel de privilegio */
       present   : 1, /* Presente */
       limit high   : 4, /* limite 16..19*/
       available   : 1, /* Disponible */
       l           : 1, /* Reservado */
       db    : 1, /* 0 = 16 bits, 1 = 32 bits */
       granularity  : 1, /* 0 = bytes, 1 = 4096 bytes */
       base high   : 8; /* base 24..31*/
}__attribute__((packed)); /* Evitar posible alineacion del compilador */
```

```
/* Definición de la GDT */
```
struct gdt_descriptor gdt[MAX_GDT_ENTRIES] __attribute__((aligned(8))); Para esta representación de descriptores de segmento el código de inicialización puede ser el siguiente:

```
   /* Inicializar la primera entrada en nulo*/
   gdt[0].limit_low = 0x0000;   /* limite 0..15*/
   gdt[0].base_low = 0x0000;    /* base 0..15 */
   gdt[0].base_middle = 0x00;   /* base 16..23 */
   gdt[0].type = 0x0;           /* Informacion de acceso */
   gdt[0].code or data = 0;     /* 1 = codigo o datos, 0 = sistema */
   gdt[0].dpl = 0;              /* Nivel de privilegio */
   gdt[0].present = 0;          /* Presente */
   gdt[0].limit_high = 0x0;     /* limite 16..19*/
   gdt[0].available = 0;        /* Disponible */
   gdt[0].l = 0;                /* Reservado */
   gdt[0].db = 0;               /* 0 = 16 bits, 1 = 32 bits */
   gdt[0].granularity = 0;      /* 0 = bytes, 1 = 4096 bytes */
   gdt[0].base_high= 0x00;      /* base 24..31*/

   /* Inicializar la segunda entrada con un descriptor de segmento de
      código flat, nivel de privilegios 0 */
   gdt[1].limit low = 0xFFFF    /* limite 0..15*/
   gdt[1].base_low = 0x0000;    /* base 0..15 */
   gdt[1].base_middle = 0x00;   /* base 16..23 */
   gdt[1].type = 0xA;           /* Informacion de acceso: Código, R/X */
   gdt[1].code_or_data = 1;     /* 1 = codigo o datos, 0 = sistema */
   gdt[1].dpl = 0;              /* Nivel de privilegio */
   gdt[1].present = 1;          /* Presente */
   gdt[1].limit_high = 0xF;       /* limite 16..19*/
   gdt[1].available = 0;        /* Disponible */
   gdt[1].l = 0;                /* Reservado */
   gdt[1].db = 1;               /* 0 = 16 bits, 1 = 32 bits */
   gdt[1].granularity = 1;      /* 0 = bytes, 1 = 4096 bytes */
   gdt[1].base_high= 0;         /* base 24..31*/

   /* Inicializar la tercera entrada con un descriptor de segmento de datos
   flat, nivel de privilegios 0 */
   gdt[2].limit_low = 0xFFFF    /* limite 0..15*/
   gdt[2].base low = 0x0000;    /* base 0..15 */
   gdt[2].base_middle = 0x00;   /* base 16..23 */
   gdt[2].type = 0x2;           /* Informacion de acceso: Datos, R/W */
   gdt[2].code_or_data = 1;     /* 1 = codigo o datos, 0 = sistema */
   gdt[2].dpl = 0;              /* Nivel de privilegio */
   gdt[2].present = 1;          /* Presente */
   gdt[2].limit high = 0xF;     /* limite 16..19*/
   gdt[2].available = 0;        /* Disponible */
   gdt[2].l = 0;                /* Reservado */
   gdt[2].db = 1;               /* 0 = 16 bits, 1 = 32 bits */
   gdt[2].granularity = 1;      /* 0 = bytes, 1 = 4096 bytes */
   gdt[2].base_high= 0;         /* base 24..31*/
```

Carga de la GDT

Para cargar la GDT se utiliza la instrucción de ensamblador

```
lgdt ptr_addr
```

La instrucción lgdt toma el puntero y lo carga en el registro GDTR del procesador. Así, la traduccion de direcciones lógicas a direcciones físicas se realiza por hardware.

ptr_addr corresponde a la dirección de memoria en la cual se encuentra una estructura de datos que describe la GDT. Esta estructura de datos se denomina 'puntero al GDT', 'GDT Pointer'.

El puntero al GDT tiene el siguiente formato:

```
 47                 15              0
 +-----------------------------------+
 |      base         |    límite    |
```

+----------------------------------+ en donde: base = dirección lineal del gdt, que corresponde a la direccion de memoria de GDT.

límite = tamaño de la GDT - 1.

Una vez que se ha cargado la GDT, se debe "pasar" nuevamente a modo protegido, estableciendo el bit 0 (PE) del registro CR0.

Después se debe realizar un far jmp (ljmp) usando un selector que haga referencia a un descriptor de segmento de código configurado en la GDT. Esto se puede lograr mediante una instrucción

```
ljmp sel : offset.
```

También se puede lograr mediante una instrucción iret, para simular un retorno de interrupción, almacenando antes en la pila los valores de EFLAGS, CS e IP adecuados para apuntar a una instrucción dentro de un segmento de código definido en la GDT. Este proceso se muestra a continuación:

```
push FLAGS_VAL
push SELECTOR VAL
push OFFSET_VAL
iret
```

En donde FLAGS_VAL será el valor del registro EFLAGS luego de retornar de la interrupción, SELECTOR_VAL deberá ser un selector que referencia a un descriptor de segmento de código configurado en la GDT, y OFFSET_VAL deberá ser el desplazamiento dentro del segmento de código en el cual se desea continuar la ejecución del kernel.

Finalmente se deben configurar los demás registros de segmento (DS, ES, FS, GS y SS) para que contengan los selectores a descriptores de segmento de datos configurados dentro de la GDT.

Al ejecutar estos pasos, el procesador estará usando la GDT configurada. En caso de cualquier error, el procesador lanzará una excepción de protección general y se reiniciará. Si esto pasa, se debe verificar que los descriptores dentro de la GDT se encuentren bien definidos, y que los valores de los selectores almacenados en los registros de segmento hagan referencia a descriptores válidos dentro de la GDT.

IDT y Gestión de Interrupciones

Autor:

Erwin Meza Vega emezav@gmail.com

Información del Proyecto : Programación de procesadores de arquitectura IA-32 : IDT y Gestión de Interrupciones

Una interrupción es una señal que rompe con el flujo normal del procesador, y que debe ser atendida inmediatamente.

Fuentes de Interrupción

En la arquitectura IA-32 se definen tres fuentes básicas de interrupción:

- Excepciones: Son condiciones de error que se presentan en la ejecución. Por ejemplo, al realizar una división por cero se lanza la excepción Division By Zero. Es una interrupción generada internamente por el procesador.
- Interrupciones de los dispositivos de hardware (discos, teclado, floppy, etc). Los dispositivos de hardware realizan solicitudes de interrupción (Interrupt Request - IRQ). Cada IRQ tiene asociado un número de interrupción predefinido, pero es posible cambiarlo por programación.
- Interrupciones por software, generadas mediante la instrucción

```
int N
```

- donde N es el número de interrupción.

La arquitectura IA-32 soporta 256 interrupciones. De estas, las 32 primeras (número 0 a 31) se asignan por defecto a las excepciones del procesador.

A continuación se muestra una descripción de las interrupciones para IA-32.

```
Número de     | Descripción
Interrupción  |
(dec/hex)     |
--------------|-------------------------------------------------
  0   0x00    | Divide error:
              | Ocurre durante una instrucción DIV, cuando el divisor
              | es cero o cuando ocurre un desbordamiento del cociente.
              | Esta excepción no genera código de error.

  1   0x01    | (Reservada)
              | Esta excepción no genera código de error.

  2   0x02    | Nonmaskable interrupt:
              | Ocurre debido a una interrupcion de hardware que no se puede
              |  enmascarar.
              | Esta excepción no genera código de error.

  3   0x03    | Breakpoint:
              | Ocurre cuando el procesador encuentra una instrucción INT 3
              | Esta excepción no genera código de error.

  4   0x04    | Overflow:
              | Ocurre cuando el procesador encuentra una instrucción INTO
              | y el bit OF (Overflow) del registro EFLAGS se encuentra activo.
              | Esta excepción no genera código de error.

  5   0x05    | Bounds check (BOUND instruction):
              | Ocurre cuando el procesador, mientras ejecuta una instrucción
              | BOUND, encuentra que el operando excede el límite especificado.
              | Esta excepción no genera código de error.

  6   0x06    | Invalid opcode:
```

```
            | Ocurre cuando se detecta un código de operación inválido.
            | Esta excepción no genera código de error.

 7  0x07    | Device Not Available (No Math Coprocessor)
            | Ocurre para alguna de las dos condiciones:
            | - El procesador encuentra una instrucción ESC (Escape)
            |   y el bit EM (emulate) bit de CR0 (control register zero)
            |   se encuentra activo.
            | - El procesador encuentra una instrucción WAIT o una instrucción
            |   ESC y los bits MP (monitor coprocessor) y TS (task switched)
            |   del registro CR0 se encuentran activos.
            | Esta excepción no genera código de error.

 8  0x08    | Double fault:
            | Ocurre cuando el procesador detecta una excepción mientras trata
            | de invocar el manejador de una excepción anterior.
            | Esta excepción genera un código de error.

 9  0x09    | Coprocessor segment overrun:
            | Ocurre cuando se detecta una violación de página o segmento
            | mientras se transfiere la porción media de un operando de
            | coprocesador al NPX.
            | Esta excepción no genera código de error.

10  0xA     | Invalid TSS:
            | Ocurre si el TSS es inválido al tratar de cambiar de tarea (Task
            |  switch).
            | Esta excepción genera código de error.

11  0xB     | Segment not present:
            | Ocurre cuando el procesador detecta que el bit P (presente) de un
            | descriptor de segmento es cero.
            | Esta excepción genera código de error.

12  0xC     | Stack exception:
            | Ocurre para las siguientes condiciones:
            | - Como resultado de una violación de límite en cualquier operación
            |   que se refiere al registro de segmento de pila (SS)
            | - Cuando se trata de establecer SS con un selector cuyo descriptor
            |   asociado se encuentra marcado como no presente, pero es válido
            | Esta excepción genera código de error.

13  0xD     | General protection violation (GP):
            | Cada violación de protección que no causa otra excepción
            | causa una GP.
            | - Exceder el límite de segmento para CS, DS, ES, FS, o GS
            | - Exceder el límite de segmento cuando se referencia una tabla de
            |   descriptores
            | - Transferir el control a un segmento que no es ejecutable
            | - Escribir en un segmento de datos de sólo lectura o en un
            |   segmento de código
            | - Leer de un segmento marcado como sólo de ejecución
            | - Cargar en SS un selector que referencia a un segmento de sólo
            |   lectura
            | - Cargar SS, DS, ES, FS, o GS con un selector que referencia a un
            |   descriptor de tipo "sistema"
            | - Cargar DS, ES, FS, o GS con un selector que referencia a un
            |   descriptor de segmento marcado como ejecutable que además no se
            |   puede leer
            | - Cargar en SS un selector que referencia un descriptor  de
            |   segmento ejecutable
            | - Acceder a la memoria por medio de DS, ES, FS, o GS cuando estos
            |   registros de segmento contienen un selector nulo
            | - Pasar la ejecución (task switch) a una tarea marcada como "Busy"
            | - Violar las reglas de privilegios
            | - Cargar CR0 con los bits PG=1 y PE=0 (habilitar la paginación y
            |      no habilitar el modo protegido)
            | - Lanzar una interrupción o una excepción a través de un trap
            |   gate desde Modo Virtual 8086 a un privilegio (DPL) diferente
            |   de cero
            | Esta excepción genera código de error.
```

```
14  0xE     | Page fault:
            | Ocurre cuando la paginación está habilitada (PG = 1) en CR0
            | y el procesador detecta alguna de las siguientes condiciones
            | cuando trata de traducir una dirección lineal a física:
            | - El directorio de tabla de páginas o la tabla de páginas requerido
            |   para realizar la traducción tiene 0 en su bit de presente (P)
            | - El procedimiento actual no tiene los suficientes privilegios para
            |   acceder la página indicada.
            | Esta excepción genera código de error.

15  0xF     | (Reservada)
            | Esta excepción no genera código de error.

16  0x10    | x87 FPU Floating-Point Error (Math Fault)
            | Ocurre cuando el procesador detecta una señal del coprocesador en el
            | pin de entrada ERROR#.

17  0x11    | Alignment Check
            | Ocurre cuando se realiza una referencia de datos en la memoria a
            | una región no alineada.
            | Esta excepción genera código de error.

18  0x12    | Machine Check
            | Depende del modelo y las características del procesador.
            | Esta excepción no genera código de error.

19  0x23    | SIMD Floating-Point Exception
            | Ocurre cuando existe un error en las instrucciones SSE/SSE2/SSE3.
            | Esta excepción no genera código de error.


20  0x24    | Reservadas por Intel.
 hasta      |
31  0x1F    | Estas excepciones no generan código de error.
            |


32  0x20    | Interrupción externa o interrupción invocada mediante la
 hasta      | instrucción INT N
255 0xFF    | Estas interrupciones no generan código de error
```

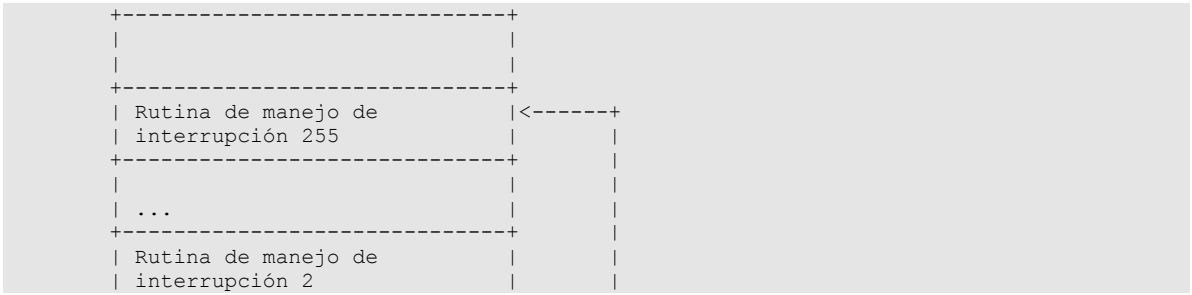
Nota:

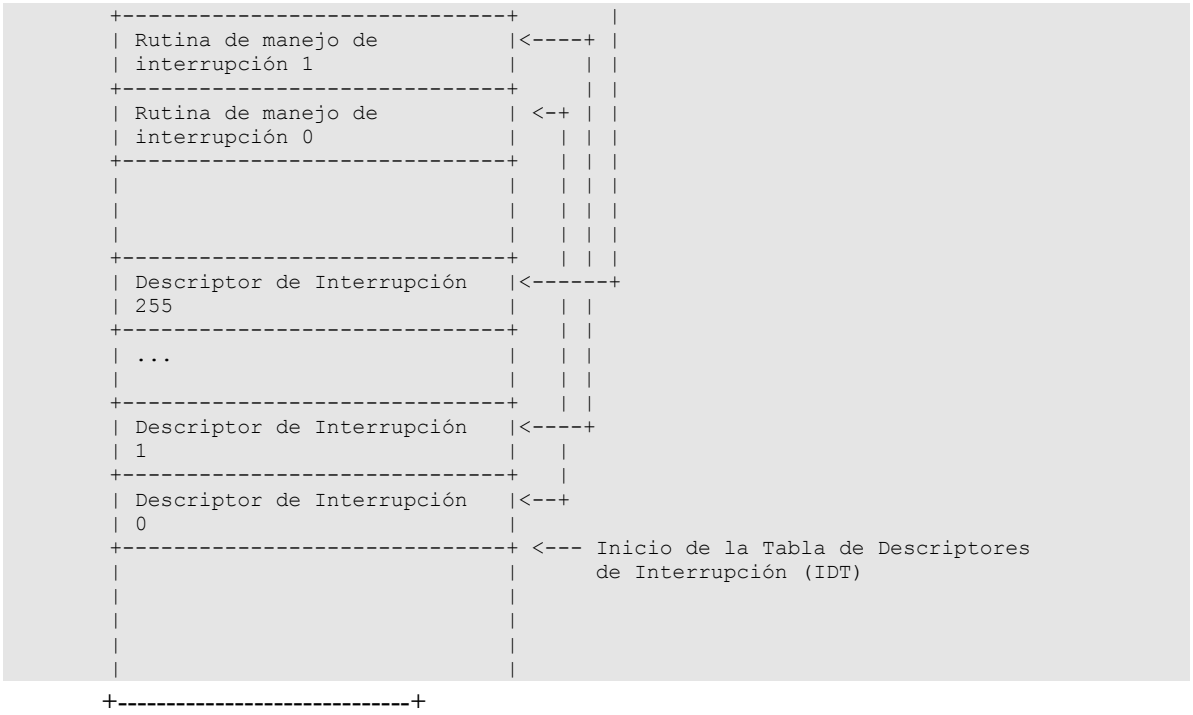
Tabla adaptada de Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1. Sección 5.3.1.

Tabla de Descriptores de Interrupción (IDT)

La IDT es una estructura de datos que usa el procesador en el momento en que ocurre la interrupción, y que debe estar configurada antes de habilitar las interrupciones. Es una tabla que contiene una serie de entradas denominadas "descriptores", que definen entre otros parámetros la dirección de memoria en la cual se encuentra cada rutina de manejo de interrupción.

El siguiente esquema muestra la IDT y las rutinas de manejo de interrupción en memoria:

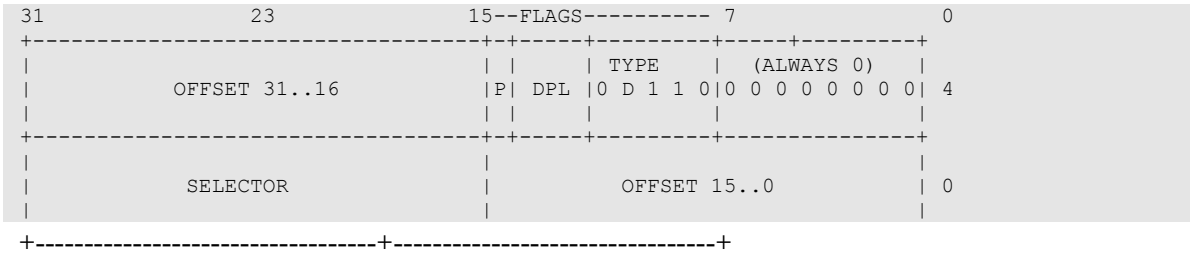




La IDT está conformada por 256 descriptores, uno para cada interrupción. Cada descriptor ocupa 8 bytes, y puede ser de uno de los siguientes tipos:

- Task-Gate
- Interrupt-Gate
- Trap-Gate

Cada entrada tiene el siguiente formato:



En donde:

- Offset: Desplazamiento (offset) en el cual se encuentra la rutina de manejo de interrupción (la dirección de memoria de la rutina) dentro de un segmento de código.
- Selector: Selector que referencia al descriptor de segmento de código en la GDT dentro del cual se encuentra la rutina de manejo de interrupción.
- D : Tipo de descriptor : (0=16 bits), (1=32 bits)
- FLAGS : compuesto por los bits P (1 bit), DPL (2 bits) y TYPE (5 bits). Para un interrupt gate, el valor de FLAGS es 0x8E = 10001110 (P = 1, DPL = 0, D = 1)

La dirección lógica segmento : offset que se obtiene del descriptor se traduce a una dirección lineal. Si la paginación se encuentra deshabilitada (por defecto), la dirección lineal es la misma dirección física en la cual se encuentra la rutina que atenderá la interrupción.

Carga de la IDT

Para cargar la IDT se utiliza la instrucción de ensamblador

```
lidt ptr_addr
```

La instrucción idt toma el puntero y lo carga en el registro IDTR del procesador. ptr_addr corresponde a la dirección de memoria en la cual se encuentra una estructura de datos que describe la IDT. Esta estructura de datos se denomina 'puntero a la IDT', 'IDT Pointer'.

El puntero al IDT tiene el siguiente formato:

```
 47                  15              0
 +-----------------------------------+
 |      base         |    límite    |
```

 +---------------------------------+ en donde: base = dirección lineal de la IDT, que corresponde a la dirección de memoria de IDT.

límite = tamaño de la IDT en Bytes. Si la IDT tiene 256 entradas y el tamaño de cada entrada es de 8 bytes, el tamaño total de la IDT es de 2048 bytes (2 KB).

Gestión de Interrupciones en IA-32

Cuando ocurre la interrupción N, el procesador busca la entrada N dentro de la IDT, y de ella obtiene la dirección lineal de la rutina de manejo de la interrupción N a la cual se debe pasar el control.

Antes de pasar el control a la rutina de manejo de interrupción, el procesador almacena en el tope de la pila el valor de EFLAGS, CS y EIP en este orden. Esto permitirá ejecutar la rutina de manejo de interrupción y luego continuar la ejecución en el punto exacto en el que se interrumpió al procesador.

Si la interrupción genera un código de error (algunas de las excepciones de IA-32), este código de error también se inserta en la pila.

A continuación se ilustra el estado de la pila en el momento de pasar el control a la rutina de manejo de interrupción.

1. Cuando ocurre una interrupción que no tiene código de error:

```
      Estado de la pila en el momento  de pasar el control a la rutina de manejo
      de interrupción

      +--------------------+
      |  valor             |
      +--------------------+ <- SS: ESP antes de ocurrir la interrupción
      | EFLAGS             |
      +--------------------+
      | CS                 |
      +--------------------+
      | EIP                |
      +--------------------+ <- SS: ESP al momento de pasar el control a la
      | Espacio            |   rutina de manejo de interrupción
      | disponible en la   |
      | pila               |
      |                    |
      |                    |
      +--------------------+ <---  Base de la pila
      |                    |
      |                    |
      +--------------------+
```

Si el segmento de código en el cual se encuentra la rutina de manejo de interrupción tiene un nivel de privilegios (DPL) diferente al nivel de privilegios del segmento de código en el que se encuentra el código interrumpido, ocurre un "Cambio de Contexto" de pila. El procesador almacena primero SS y ESP, y luego almacena EFLAGS, CS e IP en la pila.

```
    +--------------------------+
    | valor en la pila         | Pila del código (o tarea) antes de ocurrir la
    |--------------------------| interrupción o excepción
    | valor en la pila         |
    +--------------------------+  <-- OLD SS: OLD ESP --+
                                                       |
                                                       |
 +-----------------------------------------------------+
 |  +--------------------------+
 |  | old ss                   | Estos valores son almacenados automáticamente
 |  |--------------------------| en la pila cuando ocurre una interrupción
 +--|--> old esp               | o excepción y hay cambio de privilegios
    |--------------------------| ..
    | eflags                   | ..
    |--------------------------| ..
    | old cs                   | ..
    |--------------------------| ..
    | old eip                  | ..
    +--------------------------+  <-- ESP
```

2. Cuando ocurre una interrupción que tiene código de error asociado:

```
     Estado de la pila en el momento  de pasar el control a la rutina de manejo
     de interrupción

     +--------------------+
     |  valor             |
     +--------------------+ <- SS: ESP antes de ocurrir la interrupción
     | EFLAGS             |
     +--------------------+
     | CS                 |
     +--------------------+
     | EIP                |
     +--------------------+
     | Código de error    |
     +--------------------+ <- SS: ESP al momento de pasar el control a la
     | Espacio            |        rutina de manejo de interrupción
     | disponible en la   |
     | pila               |
     |                    |
     |                    |
     +--------------------+ <---  Base de la pila
     |                    |
     |                    |
     +--------------------+
```

Si el segmento de código en el cual se encuentra la rutina de manejo de interrupción tiene un nivel de privilegios (DPL) diferente al nivel de privilegios del segmento de código en el que se encuentra el código interrumpido, ocurre un "Cambio de Contexto" de pila. El procesador almacena primero SS y ESP, y luego almacena EFLAGS, CS e IP en la pila.

```
    +--------------------------+
    | valor en la pila         | Pila del código (o tarea) antes de ocurrir la
    |--------------------------| interrupción o excepción
    | valor en la pila         |
    +--------------------------+  <-- OLD SS: OLD ESP --+
                                                       |
                                                       |
 +-----------------------------------------------------+
 |  +--------------------------+
 |  | old ss                   | Estos valores son almacenados automáticamente
 |  |--------------------------| en la pila cuando ocurre una interrupción
 +--|--> old esp               | o excepción y hay cambio de privilegios
    |--------------------------| ..
    | eflags                   | ..
    |--------------------------| ..
    | old cs                   | ..
    |--------------------------| ..
    | old eip                  | ..
    |--------------------------| ..
    | Código de error          | ..
    +--------------------------+  <-- ESP
```

Con el fin de mantener un marco de pila uniforme, se inserta un código de error de cero en la pila para aquellas interrupciones que no generan codigo de error.

Rutinas de manejo de interrupción (ISR)

Las rutinas de manejo de interrupción son el elemento más cercano al hardware. Estas rutinas deben interactuar directamente con los registros del procesador, por lo cual se implementan en lenguaje ensamblador. Su tarea consiste en guardar el estado del procesador (el valor de todos sus registros), y luego invocar el código en C que manejará la interrupción. Después se deberá recuperar el estado del procesador y retornar mediante la instrucción **iret** .

A continuación se presentan ejemplos de rutinas de manejo de interrupción.

- Rutina para el manejo de la interrupción 5 (no genera código de error)

```
isr5:
   /* Deshabilitar las interrupciones*/
  cli
  /* Ahora se crea un marco de pila estandar para invocar la rutina general
  interrupt_dispatcher. */
  /* Codigo de error = 0 */
 push 0
 /* # de excepcion generada */
 push 5
 /* Almacenar en la pila los registros de propósito general en el siguiente
 orden: eax, ecx, edx, ebx, esp original, ebp, esi, y edi */
 pusha
 /* Almacenar en la pila los registros de segmento de datos */
 push ds
 push es
 push fs
 push gs

 /* Este marco de pila se crea en el contexto de ejecución actual. */

 /*
 La pila luce así:
 +--------------------------+
 | old SS                   |  (Si ocurre un cambio de contexto de pila)
 |--------------------------|
 | old ESP                  |  (Si ocurre un cambio de contexto de pila)
 |--------------------------| ..
 | EFLAGS                   | Estos valores son almacenados automáticamente
 |--------------------------| en la pila cuando ocurre una interrupción
 | old CS                   | ..
 |--------------------------| ..
 | old EIP                  | ..
 |--------------------------| ------------------------------------------
 | 0 (código de error)      | push 0 (código de error = 0)
 |--------------------------|
 | # de excepción generada  | push \id
 |--------------------------|
 | EAX                      | pusha
 |--------------------------|
 | ECX                      |(recuerde que pusha almacena en la pila los
 |--------------------------|registros en el siguiente orden:
 | EDX                      |eax, ecx, edx, ebx, esp original, ebp, esi,
 |--------------------------|edi)
 | EBX                      |
 |--------------------------|
 | ESP antes de pusha       |
 |--------------------------|
 | EBP                      |
 |--------------------------|
 | ESI                      |
 |--------------------------|
 | EDI                      |
 |--------------------------|-----------------------------------------
 | DS                       | ahora los registros de segmento de datos
```

```
|--------------------------|
| ES                       |
|--------------------------|
| FS                       |
|--------------------------|
| GS                       |
|--------------------------|<-- ESP
*/

   /* Configurar los registros de segmento de datos para que contengan
el selector de datos para el kernel definido en la GDT */
movw ax, KERNEL_DATA_SELECTOR
mov ds, ax
mov es, ax
mov fs, ax
mov gs, ax

/* Almacenar la posición actual del apuntador de la pila ss:esp */
mov [current_ss], ss
mov [current_esp], esp

/* Invocar al código en C que manejará la interrupción */
call interrupt dispatcher

/* Recuperar el apuntador de la pila ss:esp almacenado luego de crear
el marco de pila para la interrupción */
mov ss, [current ss]
mov esp, [current esp]

/* Ahora sacar los parámetros enviados a la pila en orden inverso */
pop gs
pop fs
pop es
pop ds
/* los registros de propósito general */
popa

/* Código de error e interrupcion generada */
add esp, 8

/*
Ahora la pila luce asi:
+--------------------------+
| old ss                   | Si ocurrió un cambio de contexto de pila,
|--------------------------| se almacena la posición de la pila anterior
| old esp                  | (SS:ESP).
|--------------------------|
| eflags                   | Estado del procesador (EFLAGS)
|--------------------------|
| old cs                   | Dirección lineal CS:EIP a la cual se debe
|--------------------------| retornar (punto en el cual se interrumpió
| old eip                  | el procesador)
+--------------------------+ <-- ESP (tope de la pila)
*/

/* Retornar de la interrupcion */
iret
/* Esta rutina 'no retorna', ya que continua la ejecucion en el contexto
```

- que fue interrumpido. */
- Rutina para el manejo de la interrupción 7 (genera código de error)

```
isr7:
   /* Deshabilitar las interrupciones*/
  cli
  /* Ahora se crea un marco de pila estandar para invocar la rutina general
  interrupt_dispatcher. */
  /* El código de error es almacenado  por el procesador en la pila
  de forma automática cuando ocurre la excepcion
  */
 /* # de excepcion generada */
 push 7
 /* Almacenar en la pila los registros de propósito general en el siguiente
```

```
orden: eax, ecx, edx, ebx, esp original, ebp, esi, y edi */
pusha
/* Almacenar en la pila los registros de segmento de datos */
push ds
push es
push fs
push gs

/* Este marco de pila se crea en el contexto de ejecución actual. */

/*
La pila luce así:
+---------------------------+
| old SS                    |  (Si ocurre un cambio de contexto de pila)
|---------------------------|
| old ESP                   |  (Si ocurre un cambio de contexto de pila)
|---------------------------| ..
| EFLAGS                    | Estos valores son almacenados automáticamente
|---------------------------| en la pila cuando ocurre una interrupción
| old CS                    | ..
|---------------------------| ..
| old EIP                   | ..
|---------------------------| ------------------------------------------
| (código de error)         | push 0 (código de error = 0)
|---------------------------|
| # de excepción generada   | push \id
|---------------------------|
| EAX                       | pusha
|---------------------------|
| ECX                       |(recuerde que pusha almacena en la pila los
|---------------------------|registros en el siguiente orden:
| EDX                       |eax, ecx, edx, ebx, esp original, ebp, esi,
|---------------------------|edi)
| EBX                       |
|---------------------------|
| ESP antes de pusha        |
|---------------------------|
| EBP                       |
|---------------------------|
| ESI                       |
|---------------------------|
| EDI                       |
|---------------------------|------------------------------------------
| DS                        | ahora los registros de segmento de datos
|---------------------------|
| ES                        |
|---------------------------|
| FS                        |
|---------------------------|
| GS                        |
|---------------------------|<-- ESP
*/

   /* Configurar los registros de segmento de datos para que contengan
el selector de datos para el kernel definido en la GDT */
movw ax, KERNEL DATA SELECTOR
mov ds, ax
mov es, ax
mov fs, ax
mov gs, ax

/* Almacenar la posición actual del apuntador de la pila ss:esp */
mov [current ss], ss
mov [current_esp], esp

/* Invocar al código en C que manejará la interrupción */
call interrupt dispatcher

/* Recuperar el apuntador de la pila ss:esp almacenado luego de crear
el marco de pila para la interrupción */
mov ss, [current_ss]
```

```
mov esp, [current_esp]

/* Ahora sacar los parámetros enviados a la pila en orden inverso */
pop gs
pop fs
pop es
pop ds
/* los registros de propósito general */
popa
/* Código de error e interrupcion generada */
add esp, 8

/*
Ahora la pila luce asi:
+--------------------------+
| old ss                   | Si ocurrió un cambio de contexto de pila,
|--------------------------| se almacena la posición de la pila anterior
| old esp                  | (SS:ESP).
|--------------------------|
| eflags                   | Estado del procesador (EFLAGS)
|--------------------------|
| old cs                   | Dirección lineal CS:EIP a la cual se debe
|--------------------------| retornar (punto en el cual se interrumpió
| old eip                  | el procesador)
+--------------------------+ <-- ESP (tope de la pila)
*/

/* Retornar de la interrupcion */
iret
/* Esta rutina 'no retorna', ya que continua la ejecucion en el contexto
```

- que fue interrumpido. */

La función interrupt_dispatcher puede ser implementada en C y contener el código para manejar las interrupciones.

Un punto clave de la implementación de la rutina es el siguiente:

```
   /* Almacenar la posición actual del apuntador de la pila ss:esp */
mov [current_ss], ss
mov [current_esp], esp

/* Invocar al código en C que manejará la interrupción */
call interrupt dispatcher

/* Recuperar el apuntador de la pila ss:esp almacenado luego de crear
el marco de pila para la interrupción */
mov ss, [current ss]
mov esp, [current_esp]
```

Este código almacena en las variables globales del kernel current_ss y current_esp los valores de ss y esp, que permiten apuntar a la posición de la pila en la cual se almacenó el estado del procesador. Luego se puede invocar a la función en C interrupt_dispatcher(). Esta función contendrá el código para manejar, y dado que current_ss y current_esp son variables globales, podrá tener acceso al estado del procesador en el momento en que ocurrió la interrupción.

Cuando interrupt_dispatcher() retorna, se recupera el apuntador a la pila en la posición en la cual se almacenó el estado del procesador. De esa posición se recupera el estado de los registros a sus valores en el momento en que ocurrió la interrupción, se extrae también el código de error.

```
/* Recuperar el apuntador de la pila ss:esp almacenado luego de crear
 el marco de pila para la interrupción */
 mov ss, [current_ss]
 mov esp, [current_esp]

 /* Ahora sacar los parámetros enviados a la pila en orden inverso */
 pop gs
 pop fs
 pop es
 pop ds
```

```
/* los registros de propósito general */
popa
/* Código de error e interrupcion generada */
add esp, 8
```

Ahora en el tope de la pila se encuentra la posición de memoria a la cual se debe retornar de la interrupción, y si ocurrió un cambio de contexto de pila también se encuentra

```
+--------------------------+
| old ss                   | Si ocurrió un cambio de contexto de pila,
|--------------------------| se almacena la posición de la pila anterior
| old esp                  | (SS:ESP).
|--------------------------|
| eflags                   | Estado del procesador (EFLAGS)
|--------------------------|
| old cs                   | Dirección lineal CS:EIP a la cual se debe
|--------------------------| retornar (punto en el cual se interrumpió
| old eip                  | el procesador)
+--------------------------+ <-- ESP (tope de la pila)
```

La instrucción

```
iret
```

Extrae el valor de EIP, CS y EFLAGS, y el valor de SS y ESP si ocurrió un cambio de contexto de pila. Con esto, la ejecución continúa exactamente en el sitio en el cual fue interrumpido el procesador.

Ver también:

idt.h Archivo con las definiciones necesarias para la gestión de interrupciones
isr.S Archivo que contiene la implementación de las rutinas de manejo para las 256 interrupciones de IA-32
idt.c Archivo que contiene las rutinas para configurar y cargar la IDT y para gestionar las interrupciones.
Tabla Global de Descriptores - GDT

PIC y Manejo de Solicitudes de Interrupción

Esta página explica los conceptos básicos relacionados con el manejo de solicitudes de interrupción de dispositivos de Entrada y Salida (E/S).

PIC - Programmable Interrupt Controller

Cuando un dispositivo de Entrada / Salida requiere atención, lanza una Solicitud de Interrupción (Interrupt Request - IRQ). Estas IRQ son recibidas por un dispositivo llamado el PIC PIC (Programmable Interrupt Controller). El trabajo del PIC consiste en recibir y priorizar las IRQ recibidas, y enviar una señal de interrupción a la CPU.

En la arquitectura IA-32 el sistema cuenta con dos controladores PIC, uno llamado "Maestro" y otro "Esclavo", que se encuentra conectado en cascada al PIC Maestro. Cada PIC puede atender 8 líneas de IRQ, por lo tanto se pueden atender hasta 16 solicitudes.

Al arranque del sistema, las líneas de interrupción IRQ0 a IRQ 5 se encuentran mapeadas a las interrupciones numero 0x8 a 0xF. Las líneas de interrupción IRQ8 a IRQ 15 se encuentran mapeadas a las interrupciones 0x70 a 0x77.

Lista de Solicitudes de Interrupción

La lista de IRQ es la siguiente:

IRQ	Número de Interrupción	Descripción
IRQ0	0x08	Timer
IRQ1	0x09	Teclado
IRQ2	0x0A	Cascade para el PIC esclavo
IRQ3	0x0B	Puerto serial 2
IRQ4	0x0C	Puerto serial 1
IRQ5	0x0D	AT: Puerto paralelo2 PS/2 : reservado
IRQ6	0x0E	Diskette
IRQ7	0x0F	Puerto paralelo 1
IRQ8/IRQ0	0x70	Reloj de tiempo real del CMOS
IRQ9/IRQ1	0x71	Refresco vertical de CGA
IRQ10/IRQ2	0x72	Reservado
IRQ11/IRQ3	0x73	Reservado
IRQ12/IRQ4	0x74	AT: reservado. PS/2: disp. auxiliar
IRQ13/IRQ5	0x75	FPU (Unidad de Punto Flotante)
IRQ14/IRQ6	0x76	Controlador de disco duro
IRQ15/IRQ7	0x77	Reservado

Al observar la tabla anterior, se hace evidente que existe un problema: las interrupciones 0x8 a 0x0F también son utilizadas para las excepciones de la arquitectura IA-32, ya que éstas siempre ocupan las interrupciones 0-31.

Por esta razón, es necesario reprogramar el PIC para que las interrupciones de entrada/salida se mapeen después de las excepciones de IA-32, es decir desde la interrupción numero 32 en adelante. A la IRQ 0 (Timer) le corresponderá la interrupción número 32, y asi sucesivamente.

Características del PIC

Este microcontrolador maneja las interrupciones en la arquitectura IA-32, y posee los siguientes puertos de entrada/salida:

```
Puerto  Descripción
-----   ---------------------------------------
0x20    Registro de comandos y estado del PIC maestro
0x21    Registro de máscara de IRQ y datos del PIC maestro
0xA0    Registro de comandos y datos del PIC esclavo
```
0xA1 Registro de máscara de IRQ y datos del PIC esclavo

Configuración del PIC

La configuración del PIC se realiza por medio de Palabras de Control de Inicialización (Initialization Control Words - ICW).

Initialization Control Word 1 (ICW1).

Esta es la palabra primaria para inicializar el PIC, y su valor debe ser escrito en el registro de comandos del PIC. El formato de la ICW1 es el siguiente:

```
Bit  Descripción
---  ------------------------------------------------------
0    1 = El PIC debe esperar a ICW 4 durante la inicializacion
1    1 = Sólo hay un PIC en el sistema. 0 = El PIC se encuentra conectado en
     cascada con otros PIC, y se debe enviar ICW3 al controlador.
2    1 = El intervalo de la dirección CALL es 4, 0 = el intervalo es 8.
  Este bit es ignorado en x86 y su valor por defecto es 0.
3    1 = Operar en modo disparado por nivel. 0 = Operar en modo disparado por
     borde.
4    Bit de inicializacion. 1 = El PIC debe ser inicializado
5    Dirección del vector de interrupcion (solo para MCS-80/85), en IA-32 = 0
6    Dirección del vector de interrupcion (solo para MCS-80/85), en IA-32 = 0
```
7 Dirección del vector de interrupcion (solo para MCS-80/85), en IA-32 = 0

Para inicializar el PIC se requiere que los bits 0 y 4 sean en 1 y los demás tengan valor 0. Esto significa que el valor de ICW1 es 0x11. ICW1 debe ser escrita en el registro de comandos del PIC maestro (dirección de e/s 0x20).

Se debe recordar que el PIC se encuentra en cascada con otro (PIC esclavo), por lo tanto tambien se debe escribir ICW1 en el registro de comandos del PIC esclavo (dirección de e/s 0xA0).

Initialization Control Word 2 (ICW2)

Esta palabra permite definir la dirección base (inicial) del vector de interrupción (el número) que el PIC va a utilizar. El formato de ICW2 es el siguiente:

```
Bit    Descripción
---    ------------------------
0-2    Bits 8-10 de la dirección en la IDT en modo MCS-80/85
3-7    Bits de la dirección en la IDT en modo MCS-80/85. En x86 especifica la
```
dirección del número de interrupción base.

Debido a que las primeras 32 entradas están reservadas para las excepciones en la arquitectura IA-32, ICW2 debe contener un valor mayor o igual a 32 (0x20).

Al utilizar los PIC en cascada, se debe enviar ICW2 a los dos controladores en su registro de datos (0x21 y 0xA1 para maestro y esclavo respectivamente), indicando la dirección en la IDT que va a ser utilizada por cada uno de ellos.

Las primeras 8 IRQ van a ser manejadas por el PIC maestro y se mapearán en la interrupción 32 (0x20) en adelante. Las siguientes 8 interrupciones las manejará el PIC esclavo, y se mapearán a la interrupción 40 (0x28) en adelante.

Initialization Control Word 3 (ICW3)

Esta palabra permite definir las lineas de IRQ que van a ser compartidas por los PIC maestro y esclavo. Al igual que ICW2, ICW3 también se escribe en los registros de datos de los PIC (0x21 y 0xA1 para el PIC maestro y esclavo, respectivamente).

Para el PIC maestro, ICW3 tiene el siguiente formato:

```
Bits    Descripción
----    ------------------------------------------------------------
```

0-7 Determina la línea de IRQ que está conectada al PIC esclavo

El bit 0 representa la línea de IRQ 0, el bit 1 representa la línea de IRQ 1, el bit 2 representa la línea de IRQ 2, y así sucesivamente.

Dado que en la arquitectura IA-32 el PIC maestro se conecta con el PIC esclavo por medio de la linea IRQ 2, el valor de ICW3 debe ser 00000100 (0x04), que define el bit 3 (correspondiente a la linea IRQ2) en 1.

Para el PIC esclavo, ICW3 tiene el siguiente formato:

```
Bits     Descripción
---      -------------------------
0-2      Número de IRQ que el maestro utiliza para conectarse (en notación
         binaria)
```

3-7 Reservado, debe ser 0

El número de la linea se debe representar en notación binaria. Por lo tanto, 000 corresponde a la línea de IRQ 0, 001 corresponde a la línea de IRQ 1, 010 corresponde a la línea de IRQ 2, y asi sucesivamente. Debido a que se va a utilizar la línea de IRQ 2, el valor de ICW3 para el PIC esclavo debe ser 0x00000010, (0x02).

Inicialization Control Word (ICW4)

Esta palabra controla el funcionamiento general del PIC. Su formato es el siguiente:

```
Bit   Descripción
---   ----------------------------------
0     1 = modo x86 0 = modo MCS-80/86
1     1 = En el ultimo pulso de recepcion de interrupción, el controlador
      realiza una operacion End Of Interrupt (EOI)
2     Sólo se debe usar si el bit 3 es 1. Si este bit es 1, selecciona el
      buffer maestro.
3     1= operar en modo de bufer
4     Special Fully Nested Mode. Usado en sistema con una gran cantidad de
      controladores.
```

5-7 Reservado, debe ser 0.

De esta forma, solo es necesario establecer el bit 0 en ICW4 y escribir ICW4 en los registros de datos del PIC maestro y esclavo (0x21 y 0xA1).

Palabras de Control de Operación (Operation Control Word)

Antes de retornar de la rutina de servicio de interrupción, se debe enviar al PIC maestro (y esclavo si se establece operacion en cascada) una instrucción EOI (End of Interrupt). Existen dos tipos de EOI: Específica y no especifica.

Al terminar la rutina de servicio de interrupción, se debe envar OCW2 a los registros de comando del PIC maestro y esclavo (0x20 y 0xA0).

El formato de OCW2 es el siguiente:

```
Bit         Descripción
-------     ----------------------
0, 1, 2     Nivel de interrupciones en el cual debe reaccionar
            el controlador (siempre 0)
3, 4        Siempre 0
5           Solicitud de fin de interrupción. Debe ser 1
6           Seleccion. Debe ser 0.
```

8 Rotación. Debe ser 0.

De esta forma, el único bit activo de OCW2 es el bit 5, = 00100000 = 0x20. Este es el valor a enviar a los registros de comando del PIC maestro y esclavo (0x20 y 0xA0 respectivamente).

Re-Programación del PIC

El proceso de re-programar el PIC se realiza mediante el siguiente código:

1. Enviar Initialization Command Word 1 - ICW1 al PIC El valor de ICW1 es 0x11. Debe ser escrita en el registro de comandos del PIC maestro (dirección de e/s 0x20). Si existe un PIC esclavo, ICW1 se debe enviar también su registro de comandos del PIC esclavo (0xA0)
   ```
   outb(MASTER_PIC_COMMAND_PORT, 0x11);
   ```
   outb(SLAVE_PIC_COMMAND_PORT, 0x11);
2. Enviar Initialization Command Word 2 - ICW2 al PIC ICW2 debe contener un valor mayor o igual a 32 (0x20). Las primeras 8 IRQ van a ser manejadas por el PIC maestro y se mapearán a partir del numero IDT_IRQ_OFFSET (32). Las siguientes 8 interrupciones las manejará el PIC esclavo, y se mapearán a partir de la interrupcion 40 (0x28).
   ```
   outb(MASTER_PIC_DATA_PORT, IDT_IRQ_OFFSET);
   ```
   outb(SLAVE_PIC_DATA_PORT, IDT_IRQ_OFFSET + 8);
3. Enviar Initialization Control Word 3 - ICW3 al PIC Dado que en la arquitectura Intel el PIC maestro se conecta con el PIC esclavo por medio de la linea IRQ 2, el valor de ICW3 debe ser 00000100 (0x04). Para el PIC esclavo el valor de ICW3 debe ser 00000010 (0x02).
   ```
   outb(MASTER_PIC_DATA_PORT, 0x04);
   ```
   outb(SLAVE_PIC_DATA_PORT, 0x02);
4. Enviar Initialization Control Word 4 - ICW4 al PIC. El valor de ICW4 debe ser entonces 00000001, es decir, 0x01.
   ```
   outb(MASTER_PIC_DATA_PORT, 0x01);
   ```
   outb(SLAVE_PIC_DATA_PORT, 0x01);

BIOS y Arranque del Computador

Autor:

Erwin Meza Vega emezav@gmail.com

Información del Proyecto : Programación de procesadores de arquitectura IA-32 : BIOS y Arranque

En los siguientes apartados se presentan los conceptos básicos que permiten comprender la secuencia de arranque del computador, desde que se enciende o se reinicia hasta que se tiene un sistema operativo en funcionamiento.

BIOS - Basic Input-Output System

La BIOS es un componente vital dentro de los computadores personales. Es un software que se encuentra almacenado en una ROM o integrado a la Tarjeta Madre de los Computadores, y que se ejecuta automáticamente cuando se enciende o reinicia el computador.

Técnicamente, cuando se enciende o reinicia el computador el registro de segmento CS contiene el valor 0xF000 y el registro EIP contiene el valor 0xFFF0. Es decir que la primera instrucción a ejecutar se encuentra en la dirección lineal 0xFFFF0 = 1.048.560 (Cerca del límite de 1 MB de memoria). La instrucción que se encuentra en esta posición es un jmp (salto) al código de la BIOS.

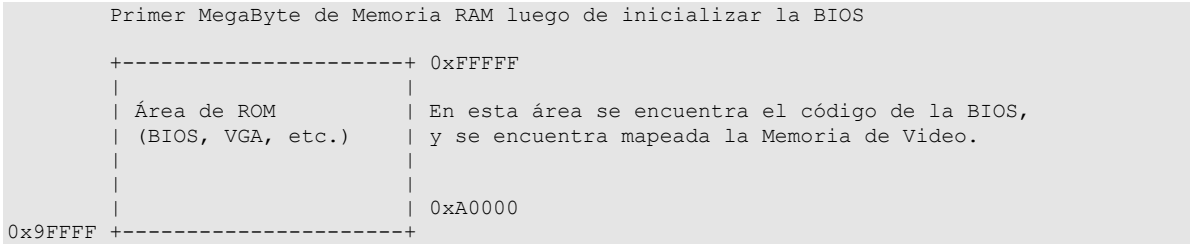
El código de la BIOS se encuentra mapeado en esta área de memoria (cercana al primer MegaByte de memoria), y luego del salto anterior recibe el control del procesador. Se debe tener en cuenta que el procesador se encuentra operando en Modo Real, por lo cual sólo se tiene acceso al primer MegaByte de memoria.

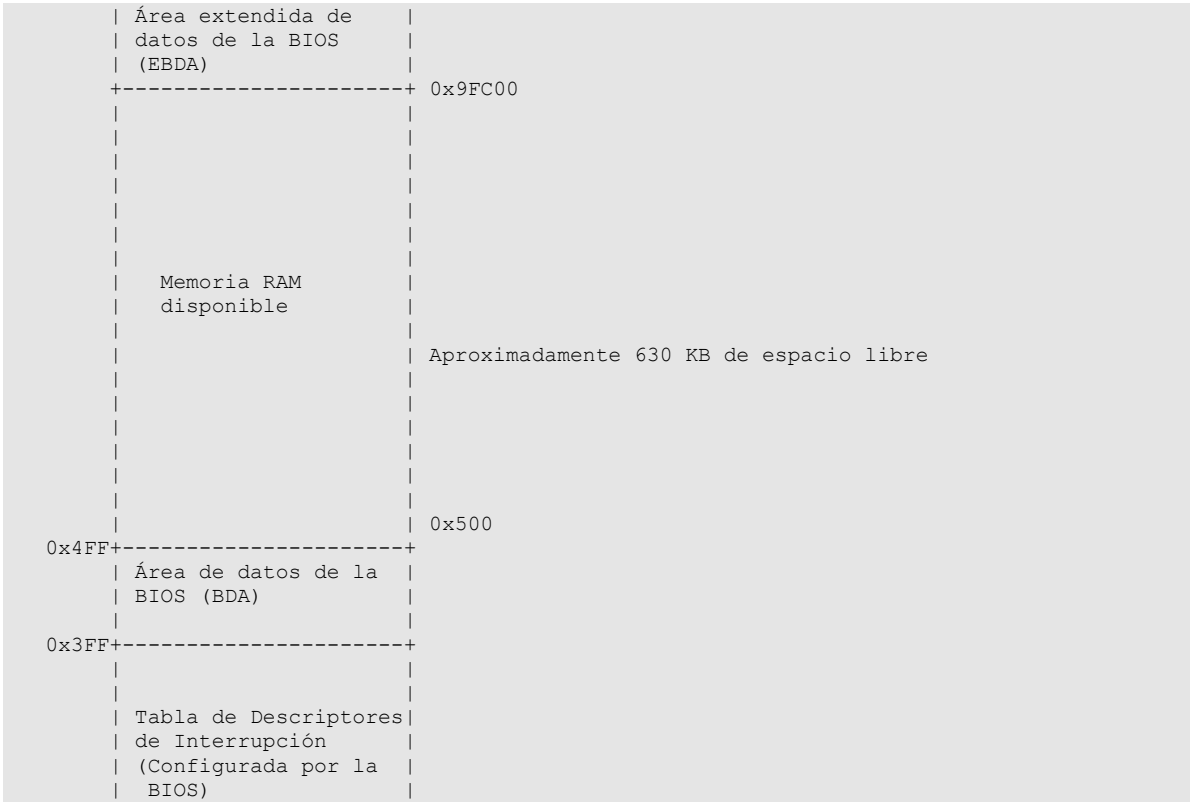
La primera tarea de la BIOS consiste en realizar un diagnóstico de sí misma y de la memoria. A este diagnóstico se le conoce como POST (Power-On Self Test). Luego realiza un diagnóstico de los dispositivos conectados al sistema (reloj, teclado, mouse, discos, unidades de red, tarjetas de video, etc.) y configura las rutinas básicas para manejar las interrupciones e interactuar con los dispositivos (De ahí su nombre, Basic Input-Output System), y configura los servicios básicos de entrada y salida como lectura /escritura de discos, manejo del teclado y la memoria de video, etc.

Las BIOS actuales ofrecen además una opción para entrar a su configuración, en la cual se puede establecer diversos parámetros, de los cuales los más importantes en la actualidad son el orden de los dispositivos de arranque y el soporte para virtualización.

Papel de la BIOS en el inicio de un Sistema Operativo

Luego del chequeo inicial, la BIOS configura el primer MegaByte de memoria con una serie de tablas y estructuras de datos necesarias para su ejecución y para la gestión básica de los dispositivos de Entrada / Salida. El siguiente esquema muestra la disposición del primer MegaByte de memoria RAM cuando la BIOS comienza su ejecución.





La disposición del área de ROM se presenta en la siguiente tabla:

Inicio	Fin	Tamaño	Descripción
0xA0000	0xAFFFF	10000 (64 KB)	Framebuffer VGA
0xB0000	0xB7FFF	8000 (32 KB)	VGA texto, monocromático
0x0xB8000	0xBFFFF	8000 (32 KB)	VGA texto, color
0xC0000	0xC7FFF	8000 (32 KB)	BIOS de video
0xF0000	0xFFFFF	10000 (64 KB)	BIOS de la board

Una de las estructuras de datos más importantes se encuentra al inicio de la memoria RAM, y se llama la Tabla de Descriptores de Interrupción (IDT). Esta estructura es un arreglo que contiene las direcciones lógicas (direcciones en formato Segmento:desplazamiento) de las rutinas genéricas que manejan las interrupciones de los dispositivos de entrada/salida y las interrupciones que ofrecen servicios de entrada y salida.

Cuando ocurre la interrupción N (invocada por software, por un dispositivo o por una condición de error del procesador), el procesador automáticamente almacena en la pila el valor actual de CS, IP y EFLAGS y le pasa el control a la rutina de manejo de interrupción cuya posición de memoria es la definida en la entrada N de esta tabla.

Las rutinas genéricas de E/S permiten realizar operaciones de básicas (leer, escribir) de los diferentes tipos de dispositivos como el disco floppy, el disco duro, el teclado, la memoria de video, etc. Se debe tener en cuenta que por su caracter genérico, estas rutinas no conocen los detalles avanzados de cada dispositivo. Por ejemplo, sólo es posible acceder a un número limitado de sectores en el disco duro, dadas las limitaciones del Modo Real.

El acceso a los servicios de la BIOS realiza por medio de interrupciones por software (lanzadas con la instrucción INT), y cada interrupción implementa un conjunto de funciones específicas. Por

ejemplo, la interrupción 0x10 permite tener acceso a una serie de servicios de video (imprimir un caracter, mover el cursor, etc) y la interrupción 0x13 permite acceder a los servicios de disco.

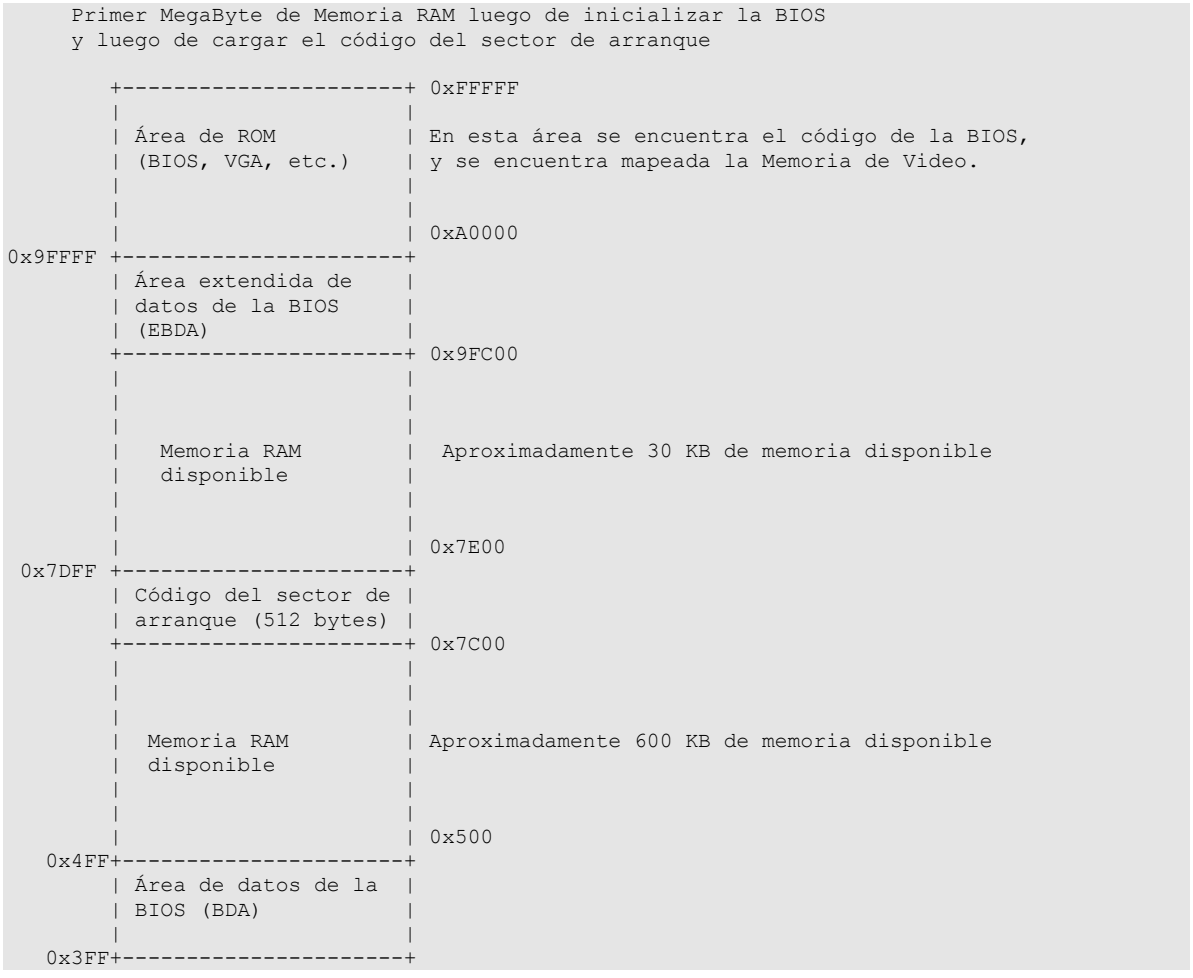
Carga del Sector de Arranque

Una vez que ha concluido su configuración inicial, la BIOS busca los dispositivos configurados en los que se presume se encuentra el sistema operativo. A estos dispositivos se les conoce como Dispositivos de Arranque. Las BIOS actuales permiten cargar e iniciar un sistema operativo desde diferentes dispositivos, que varían desde unidades floppy, CD/DVD, dispositivos USB, dispositivos de red, y por supuesto Discos Duros.

La BIOS busca y lee el primer sector (de 512 bytes) de cada dispositivo de arranque, en el orden que tenga configurado, y verifica que los últimos dos bytes de este sector contengan los valores 0x55 y 0xAA (0xAA55 en little endian). Esta es la única verificación estándar que hace la BIOS con el código del sector de arranque.

Si la BIOS no encuentra un sector de arranque válido en el primer dispositivo, continúa con el siguiente dispositivo hasta encontrar un sector de arranque válido. Si no puede encontrar ningún sector de arranque válido, imprime un mensaje de error y detiene su ejecución.

El primer sector de arranque válido leído por la BIOS se carga en la dirección de memoria 0x7C00 (31744 en decimal). El siguiente esquema muestra la posición del código del sector de arranque en memoria.



```
        |                      |
        |                      |
        | Tabla de Descriptores|
        | de Interrupción      |
        | (Configurada por la  |
        |  BIOS)               |
    0   +----------------------+
```

Luego la BIOS le pasa el control de la ejecución al código del sector de arranque, por medio de una instruccion jmp (Salto). Este salto puede tener diferentes formatos, por ejemplo (en sintaxis Intel de ensamblador):

```
ljmp 0x7C0 : 0x0000

ljmp 0x0000 : 0x7C00
```

Existen 4096 formas diferentes en que la BIOS puede realizar el salto, por lo cual se debe considerar este aspecto en el código del sector de arranque.

Además, la BIOS almacena en el registro DL un valor que indica la unidad desde la cual se cargó el sector de arranque, el cual puede ser:

- 0x01 = floppy
- 0x80 = disco duro primario.

El código del sector de arranque deberá considerar este valor para continuar con la carga del sistema operativo.

Carga del Sistema Operativo

Generalmente un sector de arranque deberá contener el código para continuar con la carga del componente central de un sistema operativo (Kernel), lo cual implica leer la tabla de particiones del disco, determinar la ubicación del kernel y cargarlo a memoria. Si el tamaño del kernel es mayor a 512 KB, se deberá pasar al modo protegido, en el cual se tiene acceso a toda la memoria RAM.

Debido a las restricciones en el tamaño del sector de arranque (ocupa exactamente 512 bytes), este debe hacer uso de los servicios ofrecidos por la BIOS para continuar con la carga del kernel. Estos servicios le permiten leer los sectores de disco en los cuales se encuentra cargado el código de inicialización del kernel.

Una vez que se ha cargado el código inicial del kernel, el sector de arranque le pasa el control. Si el sector de arranque no ha activado el modo protegido, una de las primeras tareas del kernel es habilitar este modo par tener acceso a toda la memoria y a las características avanzadas del procesador. Para ello deberá implementar los pasos requeridos para activar el modo protegido descritos en la documentación del manual de Intel.

Ver también:

Paso a Modo Protegido en Procesadores IA-32

Como mínimo se deberán implementar los pasos 1 a 4 especificados por el Manual de Intel.

El kernel continúa entonces con la carga de todo el sistema operativo, la configuración de dispositivos y muy posiblemente el inicio de una interfaz gráfica. Luego iniciará una serie de tareas que permitirán iniciar sesión e interactuar con el sistema operativo.

Uso de la BIOS en los Sistemas Operativos Actuales

Una vez cargados, los sistemas operativos modernos hacen poco o ningún uso de la BIOS. No obstante, algunos aspectos de programación del hardware sólo se pueden realizar por intermedio de la BIOS.

Los sistemas operativos modernos preservan el contenido del primer MegaByte de memoria, en el cual se encuentra mapeado el código de la BIOS. Esto les permite saltar entre el modo protegido y el modo real, para acceder a algunos servicios que implementa la BIOS y cuya implementación directa puede ser muy difícil.

Cargadores de Arranque

La mayoría de sistemas operativos actuales dejan la responsabilidad de su carga a programas especiales denominados Cargadores de Arranque. Estos permiten simplificar la tarea de cargar el kernel e interactuar con la BIOS. Los cargadores de arranque se instalan en el primer sector del disco, reemplazando en la mayoría de las ocasiones el contenido de este sector (otro código de arranque).

Las diferentes variantes de Linux y algunas variantes de UNIX al momento de su instalación también instalan un cargador de arranque genérico llamado GRUB (Grand Unified Bootlader). Este programa se instala en el primer sector del disco duro y si existe otro sistema operativo pre-instalado ofrece un menú al arranque que permite cargar tanto el Linux instalado como el sistema operativo que ya se encontraba instalado en el sistema.

Cargador de Arranque GRUB

Grub es un programa complejo, que consta de varias partes. El código inicial de GRUB se inserta en el primer sector del disco (en el sector de arranque) y es cargado por la BIOS al inicio del sistema en la posición de memoria 0x7C00. Esta parte de GRUB sólo contiene el código necesario para cargar otra porción de GRUB que se encuentra almacenada en alguna partición del disco duro (o en otro dispositivo). La segunda parte de GRUB contiene las rutinas específicas para gestionar el tipo de dispositivo en el cual se encuentra almacenado el kernel (el disco duro, un CD, una ubicación de red, etc).

Una vez que se ha cargado la segunda parte (Etapa) de GRUB, se presenta un menú que permite elegir e iniciar alguno de los sistemas operativos almacenados en el disco. Dependiendo de la selección del usuario, se carga el sistema operativo requerido.

Ver también:

Uso de los servicios de la BIOS
Paso a Modo Protegido en Procesadores IA-32
Tabla Global de Descriptores - GDT
http://www.gnu.org/software/grub/ Página oficial de GRUB (Enlace externo)

Ensamblador para procesadores IA-32

Autor:

Erwin Meza Vega emezav@gmail.com

Información del Proyecto : Programación de procesadores de arquitectura IA-32 : Ensamblador para IA-32

A continuación se presentan los fundamentos básicos del Lenguaje Ensamblador para procesadores IA-32.

Sintaxis AT&T e Intel en Lenguaje Ensamblador

Si bien la sintaxis Intel es la más utilizada en la documentación y los ejemplos, es conveniente conocer también los aspectos básicos de la sintaxis AT&T, ya que es la sintaxis por defecto del ensamblador de GNU. Las sintaxis AT&T e Intel difieren en varios aspectos, entre ellos:

- Nombre de los registros. En sintaxis AT&T los registros (de 16 bits) se denominan: %ax, %bx, %cx, %dx, %si, %di, además de los registros %sp, %bp, y los registros de segmento %cs, %ds, %es, %fs y %gs. Por su parte, en la sintaxis Intel (con la directiva '.intel_syntax noprefix') no es necesario anteponer " al nombre de los registros.
- Orden de los operandos: En la sintaxis AT&T, el orden de los operandos en las instrucciones es el siguiente:
- instrucción fuente, destino Mientras que en la sintaxis Intel, el orden de los operandos es:
- instrucción destino, fuente
- Operandos inmediatos. Los operandos inmediatos son aquellos valores que se especifican de forma explícita dentro de la instrucción. En la sintaxis AT&T, los operandos inmediatos se preceden con el signo '$' Por ejemplo, la instrucción en sintaxis AT&T:
- movb $0x05, %ah almacena en el registro ah (un byte) el valor 0x05 (0xXX) significa el número XX en formato hexadecimal). En sintaxis Intel:
- mov ah, 0x05
- Longitud de operaciones de memoria: En la sintaxis AT&T, cada instrucción debe llevar un sufijo que indica la longitud de la operación de memoria. En sintaxis Intel sólo se debe especificar el sufijo en aquellas instrucciones cuyos operandos son direcciones de memoria o cuando existe ambiguedad en el tamaño de los operandos. Por ejemplo, para mover un byte de una dirección de memoria ds:0x50, al registro AL, el código en sintaxis AT&T sería el siguiente:
- movb ds:(0x50), %al Observe el sufijo 'b' de la instrucción 'mov'. De otro lado, la sintaxis Intel sería la siguiente:
- mov al, BYTE PTR ds:[0x50] También es válida la siguiente sintaxis:
- movw al, ds:[addr] Los sufijos válidos son: b para byte, w para word, l para long (32 bits), y q para quadword (64 bits). Por otra parte, los prefijos de las direcciones de memoria en sintaxis Intel son: BYTE PTR, WORD PTR, DWORD PTR y QWORD PTR.

Referencia implícita a los registros de segmento

Recuerde que sin importar el modo de operación del procesador, siempre se usa segmentación. Así que las direcciones de memoria especificadas en una instrucción de ensamblador siempre serán relativas al inicio de un segmento. Las instrucciones de movimiento de datos (mov, lods, stos, etc) siempre harán referencia implícita al segmento de datos apuntado por el registro de segmento DS. Las instrucciones de salto siempre harán uso implícito del registro de segmento CS, y las operaciones sobre la pila (push, pop) o sobre los registros ESP y EBP siempre harán referencia al segmento apuntado por el registro SS.

En algunas instrucciones es posible cambiar el registro de segmento al cual se hace referencia de forma implícita.

Por ejemplo, la instrucción

```
mov al, [addr]
```

Es interpretada por el procesador como

```
mov al, ds:[addr]
```

Y la instrucción

```
movsw
```

Es intepretada por el procesador como:

```
  mov word ptr es:[di], word ptr ds:[si]
  inc si
  inc di
```

Así es necesario revisar la documentación de las instrucciones de ensamblador para determinar si es necesario especificar de forma explícita el registro de segmento.

Tamaño de los operandos

El uso de las instrucciones de ensamblador depende en gran medida del tipo de programa que se desea desarrollar. Si se planea desarrollar un programa para ser ejecutado en un entorno de 16 bits (Modo Real), solo se puede tener acceso a los 16 bits menos significativos de los registros, y a un conjunto limitado de instrucciones.

Por el contrario, si se planea desarrollar un programa para ser ejecutado en un entorno de modo protegido de 32 (o 64) bits, se puede tener acceso a la totalidad de bits de los registros y a un conjunto mayor de instrucciones.

La mayoría de instrucciones presentadas a continuación realizan operaciones a nivel de bytes (por ello tienen el sufijo ‘b’), y para el acceso a memoria se utiliza BYTE PTR para indicar apuntadores a bytes. También es posible operar a nivel de words (2 bytes, 16 bits), usando el sufijo ‘w’ y el modificador WORD PTR en vez de BYTE PTR para apuntadores de tipo word en memoria en sintaxis Intel.

En modo de 32 bits, es posible operar sobre doublewords (4 bytes, 32 bits) usando el sufijo ‘w’, y para instrucciones de acceso a memoria se utiliza en sintaxis Intel se debe especificar DWORD PTR en vez de BYTE PTR, o usar el sufijo ‘l’ en la instrucción mov. En operaciones de 64 bits se debe usar el sufijo ‘q’ (quadword, 8 bytes, 64 bits) para la mayoría de instrucciones y QWORD PTR para el acceso a memoria en sintaxis Intel.

Se debe recordar que el uso de los registros depende del modo de operación del procesador. Así, en modo real se puede tener acceso a los primeros de 8 bits y 16 bits de los registros de propósito general, en modo protegido se puede acceder a los 32 bits de los registros y en modo de 64 bits se puede tener acceso a los 64 bits de los registros.

Movimiento de datos

INSTRUCCIÓN mov (move)

Permite mover (copiar) datos entre dos registros, de un valor inmediato a un registro o de un valor inmediato a una posición de memoria.

En sintaxis AT&T

De inmediato a registros

```
movb $0x10, %ah /* Mover el byte (valor inmediato) 0x10 a %ah */
movb $0x20, %al /* Mover el byte 0x20 a %al */
movw $0x1020, %ax /* Mueve un word (2 bytes) a %ax */
movl $0x00001020, %eax /* Mueve un doubleword (4 bytes) a %eax */
movq $0x0000000000001020, %rax /* Mueve un quadword (8 bytes) a %rax */
```

De registros a registros

```
movb %al, %bl /* Mover un byte de %al a %bl */
movb %ah, %bh  /* Mover un byte de %ah a %bh */
movw %ax, %bx /* Equivalente a las dos instrucciones anteriores */
movl %eax, %ebx /* 32 bits */
movw %rax, %rbx /* 64 bits */
```

De registros a memoria

```
movb %al, (%si) /* Mover el byte almacenado en %al a la posición
   de memoria  apuntada por %si */
movb %al, 4(%si) /* Mover el byte almacenado en %al a la posición
de memoria apuntada por (%si + 4)*/
movb %al, -2(%si) /* Mover el byte almacenado en %al a la posición
   de memoria apuntada por (%si - 2) */
movw %ax, (%si) /* Mover el word almacenado en %ax a la posición
 de memoria  apuntada por %si */
movl %eax, (%esi) /* 32 bits */
movq %rax, (%rsi) /* 64 bits */
```

De memoria a registros

```
movb (%si), %al  /* Mover el byte de la posición de memoria
 apuntada por (%si) a %al */
movb 4($si), %al /* Mover el byte de la posición de memoria
   apuntada por (%si + 4) a %al */
movb -2($si), %al /* Mover el byte de la posición de memoria
```

apuntada por (%si - 2) a %al */

En sintaxis Intel

El orden de los operandos es instrucción destino, fuente

De inmediato a registros

```
mov ah, 0x10 /* Mover el byte (valor inmediato) 0x10 a ah */
mov al, 0x20 /* Mover el byte 0x20 a al */
mov ax, 0x1020 /* Equivalente a las dos operaciones anteriores */
mov eax, 0x00001020 /* 32 bits */
mov rax, 0x0000000000001020 /* 64 bits */
```

De registros a registros

```
mov bl, al /* Mover un byte de al a bl */
mov bh, ah  /* Mover un byte de ah a bh */
mov bx, ax /* Equivalente a las dos instrucciones anteriores */
mov ebx, eax /* 32 bits */
mov rbx, rax /* 64 bits */
```

De registros a memoria

```
mov BYTE PTR [ si ], al /* Mover el byte almacenado en al a la
    posicion de memoria apuntada por si */
movb [ si ], al      /* Equivalente a la instrucción anterior.
      Observe el sufijo 'w' en la instrucción. */

mov BYTE PTR [ si + 4 ], al /* Mover el byte almacenado en %al a
   la posición de memoria apuntada por (si + 4)*/

mov BYTE PTR [ si - 2 ], al /* Mover el byte almacenado en %al a
la posición de memoria apuntada por (si - 2) */

mov WORD PTR [ si ], ax /* Mover el word almacenado en ax a la
posición de memoria apuntada por (si) */

mov DWORD PTR [ esi ], eax /* 32 bits */
mov QWORD PTR [ rsi ], rax /* 64 bits */
```

De memoria a registros

```
mov al, BYTE PTR [ si ]   /* Mover el byte de la posición de
memoria apuntada por (si) a %al */
movw al, [ si ]   /* Equivalente a la instrucción anterior */

mov al, BYTE PTR [ si + 4] /* Mover el byte de la posición de
memoria apuntada por (si + 4) a al */

mov al, BYTE PTR [ si - 2 ] /* Mover el byte de la posición de
memoria apuntada por (si - 2) a al */

mov ax, WORD PTR [ si ]   /* Mover un word */
mov eax, DWORD PTR [ esi ]   /* 32 bits (doubleword)*/
mov rax, QWORD PTR [ rsi ]   /* 64 bits (quadword) */
```

INSTRUCCIÓN movs (move string)

Permite copiar datos entre dos posiciones de memoria. Automáticamente incrementa los dos apuntadores, luego de la copia (Ver Repeticiones).

En sintaxis AT&T

```
movsb (%si), (%di) /* Copia un byte de la posición de memoria apuntada  por el registro %si a
la posición de memoria apuntada por el registro %di */
movsw (%si), (%di) /* Copia un word de (%si) a (%di) */
movsl (%esi), (%edi) /* Copia un dobuleword de (%esi) a (%edi) */
movsq (%rsi), (%rdi) /* Copia un quadword de (%esi) a (%edi) */
```

En sintaxis Intel

```
mov BYTE PTR  [ si ], BYTE PTR [ di ] /* Mueve un byte de (si) a (di)*/
movb [ si ], [ di ] /* Equivalente a la instrucción anterior */
mov WORD PTR [ si ], WORD PTR [ di ] /* Mueve un word de (si) a (di) */
mov DWORD PTR [ esi ], DWORD PTR [ esi ] /* 32 bits */
mov QWORD PTR [ rsi ], QWORD PTR [ rdi ] /* 64 bits */
```

INSTRUCCIÓN lods (load string)

Permite copiar datos entre una posición de memoria y un registro. Automáticamente incrementa el apuntador a la memoria en el número de bytes de acuerdo con la longitud de operación.

En sintaxis AT&T

```
lodsb /* También es valido lodsb %al, (%si) */
/* Almacena un byte de la posición de memoria apuntada por (%si)en en el registro %al */

lodsw /* Almacena un word de la posición de memoria apuntada por
 (%si) en el registro %ax */

lodsl /* 32 bits */

lodsq /* 64 bits */
```

En sintaxis Intel

```
lods al, BYTE PTR [ si ]
/* Almacena un byte de la posición de memoria apuntada por (si) en  el registro al */
lodsb    /* Equivalente a la instrucción anterior.
La sintaxis abreviada también es válida */

lods ax, WORD PTR [ si ] /* Almacena un word de la posición de
memoria apuntada por (si) en el registro ax */
lodsw   /* Equivalente a la instrucción anterior. */

lodsl /* 32 bits */
```

lodsq /* 64 bits */

INSTRUCCIÓN stos (store string)

Permite copiar datos entre un registro y una posición de memoria. Incrementa automáticamente el apuntador a la memoria.

En sintaxis AT&T

```
stosb /* También es valido stosb %al, (%di) */
/* Almacena el valor de %al a la posición de memoria apuntada por (%di)  */
stosw /* Almacena el valor de %ax a la posición de memoria
 apuntada por (%di) */

stosl /* 32 bits */

stosq /* 64 bits */
```

En sintaxis Intel

```
stos BYTE PTR [ di ], al
   /* Almacena el valor de al a la posición de memoria
  apuntada por   (di) */

stosb   /* También es válida la instrucción abreviada */

stos WORD PTR [ di ], ax  /* Almacena el valor de ax a la posición
de memoria apuntada por  (di) */
stosw  /* Equivalente a la instrucción anterior. */

stosl /* 32 bits */

stosq /* 64 bits */
```

Repeticiones

Las instrucciones movs, lods y stos pueden hacer uso del prefijo 'rep' (repeat), para repetir la operación incrementando los registros ESI o EDI sea el caso y la longitud de la operación, mientras el valor del registro ECX sea mayor que cero. El valor de ECX debe ser establecido antes de invocar la instrucción.

Por ejemplo, las secuencias de instrucciones en sintaxis AT&T (modo real):

```
movw $0x100, %cx
rep stosb /* 16 bits, copiar byte a byte, incrementar %di en 1*/

movw $0x80, %cx
rep stosw /* 16 bits, copiar word a word, incrementar %di en 2 */
```

en sintaxis Intel:

```
mov cx, 0x100
rep stosb BYTE PTR [ di ], al /* 16 bits, copiar byte a byte */

mov cx, 0x80
```

rep stosw WORD PTR [di], ax /* 16 bits, copiar word a word */

Copian el valor del registro AL (un byte) o AX (dos bytes) a la posición de memoria apuntada por ES:(DI), y automáticamente incrementa el apuntador DI. Cada vez que se ejecuta la instrucción, el

registro CX se decrementa y se compara con cero. Los cuatro ejemplos realizan la misma acción en modo real: permiten copiar 256 bytes de un registro a la memoria.

En la primera forma se realizan 256 iteraciones (0x100) para copiar byte a byte, y en la segunda solo se requieren 128 iteraciones (0x80), ya que cada vez se copia un word (dos bytes) cada vez.

Dirección de incremento

Se debe tener en cuenta que las instrucciones lods, stos y movs automáticamente incrementan los registros ESI o EDI según sea el caso y la longitud de la operación (byte, word, doubleword o quadword).

Esto se puede garantizar invocando la instrucción cld (clear direction flag) con la cual se realiza el incremento automáticamente.

Por el contrario, la instrucción std (set direction flag) causa que las instrucciones decrementen automáticamente los registros ESI o EDI según sea el caso.

Saltos, bifurcaciones y ciclos

Al ejecutar el programa, el procesador simplemente recupera (fetch) la siguiente instrucción apuntada por el registro EIP, la decodifica (decode) y la ejecuta (execute). Luego continúa con la siguiente instrucción. A esto se le denomina el ciclo 'fetch-decode-execute'. No obstante, en los programas de alto nivel generalmente existen instrucciones que permiten alterar la ejecución de un programa de acuerdo con determinadas condiciones, y repetir una serie de instrucciones dentro de ciclos.

Saltos

Para evitar la linealidad de la ejecución, el procesador dispone de instrucciones que permiten 'saltar' (cambiar el valor de EIP), para poder continuar la ejecución del programa en otro sitio diferente dentro del código. A continuación se describe de forma resumida las instrucciones más usadas para realizar saltos dentro del código.

Instrucción jmp (jump)

Salto incondicional. Permite continuar la ejecución de forma incondicional en otra posición de memoria, designada generalmente por una etiqueta en el código fuente ensamblador.

Ejemplo (En sintaxis AT&T e Intel):

```
label1:
 ...
(instrucciones)
...
jmp label2
(xxxinstruccionesxxx)
...
label2:
...
(instrucciones)
...
```

En este ejemplo, las instrucciones desde jmp label2 hasta la etiqueta label2 no son ejecutadas. Esto es especialmente útil si se requiere ‘saltar’ una región del código en ensamblador.

En los programas jmp generalmente se salta dentro del mismo segmento. No obstante, si programa lo requiere, es posible saltar a otros segmentos.

Saltos entre segmentos

El formato de la instrucción JMP para salto entre segmentos es la siguiente:

En sintaxis AT&T:

```
ljmp seg, offset
```

En sintaxis Intel:

```
ljmp seg:offset
```

En estos formatos de jmp, CS adquiere el valor especificado en 'seg' y EIP adquiere el valor especificado en 'offset'.

Un ejemplo clásico consiste en el salto inicial que debe realizar un sector de arranque, para garantizar que el registro de segmento CS contenga el valor de 0x7C0. Si bien la BIOS siempre carga al sector de arranque en la Dirección lineal 0x7C00, es necesario establecer CS explícitamente en 0x7C0 para que apunte al inicio del segmento de código del sector de arranque cargado en memoria.

Esto se logra mediante la siguiente estructura de código:

```
start:
   ljmp 0x7C0 : OFFSET entry_point


entry point: /* La ejecución continúa en este punto, pero CS = 0x7C0. */
   .. demás instrucciones


   ..

   jmp finished

finished: /* Ciclo infinito dentro del código del sector de arranque */
```

jmp finished

Instrucciones de salto condicional

Instrucciones jz, jnz, je, jne, jle, jge, jc. Estas instrucciones generalmente vienen precedidas por instrucciones que realizan manipulación de registros o posiciones de memoria, y que alteran el contenido del registro EFLAGS (alguno o varios de sus bits).

La sintaxis de todas estas instrucciones es la misma:

```
 jX label
```

donde **X** es la condición que se debe cumplir para realizar el salto.

Algunos ejemplos del uso de las instrucciones de salto condicional son:

```
jz label
jnz label
je label
```

Con estas instrucciones se realiza un salto a la etiqueta **label** en el código ensamblador, dependiendo si se cumple o no la condición de acuerdo con la instrucción. Si la condición no se cumple, el procesador continúa ejecutando la siguiente instrucción que se encuentre después de la instrucción de salto condicional.

Las instrucciones y condiciones más importantes son:

- jz / je label (jump if zero / jump if equal): Saltar a la etiqueta **label** si el bit ZF del registro EFLAGS se encuentra en 1, o si en una comparación inmediatamente anterior los operandos a comparar son iguales.
- jnz / jne label (jump if not zero / jump if not equal): Contrario a jz. Saltar a la etiqueta label si el bit ZF del registro EFLAGS se encuentra en 0, o si en una comparación inmediatamente anterior los operandos a comparar no son iguales.
- jc label: Saltar a la etiqueta label si el bit CF del registro FLAGS se encuentra en 1. Este bit se activa luego de que se cumplen determinadas condiciones al ejecutar otras instrucciones, tales como sumas con números enteros. Igualmente algunos servicios de DOS o de la BIOS establecen el bit CF del registro FLAGS en determinadas circunstancias.
- jnc label: Contrario a jc. Saltar a la etiqueta si el bit CF del registro EFLAGS se encuentra en 0.

Saltos condicionales y comparaciones

Uno de los usos más comunes de las instrucciones de salto condicional es comparar si un operando es mayor o menor que otro para ejecutar código diferente de acuerdo con el resultado. Para ello, las instrucciones de salto condicional van precedidas de instrucciones de comparación:

```
cmp fuente, destino  /*(AT&T)*/
```

ó

cmp destino, fuente /* (Intel) */

Otras instrucciones de salto condicional que también se pueden utilizar para números con signo son:

- jg label: jump if greater: Saltar a la etiqueta label si la comparación con signo determinó que el operando de destino es mayor que el operando de fuente
- jl label: jump if less: Saltar a la etiqueta label si la comparación encontró que el operando de destino es menor que el operando de fuente
- jge label: Jump if greater of equal: Saltar a la etiqueta label si el operando de destino es mayor o igual que el operando de fuente
- jle: Jump if less or equal : Saltar a la etiqueta label si el operando de destino es menor o igual que el operando fuente.

EJEMPLOS

En el siguiente ejemplo se almacena el valor inmediato 100 en el registro AL y luego se realiza una serie de comparaciones con otros valores inmediatos y valores almacenados en otros registros.

En sintaxis AT&T:

```
 movb $100, %al /* A los valores inmediatos en decimal  no se les antepone '0x'
       como a los hexa*/
 cmpb $50, %al /* Esta instrucción compara el valor de %al  con 50 */
 jg es_mayor
 /* Otras instrucciones, que se ejecutan si el valor de %al no es mayor que 50
  (en este caso no se ejecutan, %al = 100 > 50 */
 jmp continuar
/* Este salto es necesario, ya que de otro modo el procesador
 ejecutará las instrucciones anteriores y las siguientes
 también, lo cual es un error de lógica*/
 es mayor:
  /*  La ejecución continua aquí si el valor de %al (100) es mayor  que 50*/
 continuar:
    /*  Fin de la comparación. El código de este punto hacia
 Abajo se ejecutará para cualquier valor de %al */
```

En sintaxis Intel:

```
   mov al, 100  /* A los valores inmediatos en decimal  no se les antepone '0x'
                como a los hexa*/
 cmpb al, 50 /* Esta instrucción compara el valor de %al  con  50*/
 jg es_mayor
```

```
  /* Otras instrucciones, que se ejecutan si el valor de %al no es mayor que 50
   (en este caso no se ejecutan, %al = 100 > 50 */
  jmp continuar
 /* Este salto es necesario, ya que de otro modo el procesador
  ejecutará las instrucciones anteriores y las siguientes
  también, lo cual es un error de lógica*/
  es_mayor:
 /*  La ejecución continua aquí si el valor de %al (100) es mayor  que 50*/
  continuar:
  /*  Fin de la comparación. El código de este punto hacia
```
Abajo se ejecutará para cualquier valor de %al */

Ciclos

Los ciclos son un componente fundamental de cualquier programa, ya que permiten repetir una serie de instrucciones un número determinado de veces. Existen varias formas de implementar los ciclos. Las dos formas más acostumbradas son:

1. Combinar un registro que sirve de variable, una comparación de este registro y una instrucción de salto condicional para terminar el ciclo
2. Usar la instrucción loop y el registro CX. Este registro no se puede modificar dentro del cuerpo del ciclo.

A continuación se ilustran los dos casos. En ciclos más avanzados, las condiciones son complejas e involucran el valor de una o más variables o registros.

El pseudocódigo es el siguiente:

```
  cx = N
+-> ciclo:
|  si cx = 0
|   goto fin_ciclo
| (Demás instrucciones del ciclo)
| decrementar cx
| goto ciclo
fin ciclo:
Instrucciones a ejecutar luego del ciclo
```

Implementación usando a cx como contador y realizando la comparación

En sintaxis AT&T:

```
movw $10, %cx /* Para el ejemplo, repetir 10 veces */
ciclo:
cmpw $0, %cx /* Comparar %cx con cero*/
je fin_ciclo /* Si %cx = 0, ir a la etiqueta fin_ciclo */
/* Demás instrucciones del ciclo */
decw %cx /* %cx = %cx - 1 */
jmp ciclo /* salto incondicional a la etiqueta ciclo*/
fin_ciclo:
/* Instrucciones a ejecutar después del ciclo */
```

En sintaxis Intel:

```
mov cx, 10 /* Para el ejemplo, repetir 10 veces */
ciclo:
cmp cx, 0 /* Comparar cx con cero*/
je fin_ciclo /* Si cx = 0, ir a la etiqueta fin_ciclo */
/* Demás instrucciones del ciclo */
dec cx /* cx = cx - 1 */
jmp ciclo /* salto incondicional a la etiqueta ciclo*/
fin_ciclo:
/* Instrucciones a ejecutar después del ciclo */
```

Implementación usando la instrucción loop y el registro cx

En sintaxis AT&T:

```
 movw $10, %cx /* Para el ejemplo, repetir 10 veces */
 ciclo:
  /* Demás instrucciones dentro del ciclo
  Importante: Recuerde que para el ciclo, se utiliza el registro %cx.
   Por esta razón no es aconsejable utilizarlo dentro del ciclo. */

 loop ciclo /* Decrementar automáticamente %cx y verificar si es
  mayor  que cero. Si %cx es mayor que cero, saltar
  a la  etiqueta 'ciclo'. En caso contrario,
  continuar la ejecución en la instrucción
```

siguiente*/

En sintaxis Intel:

```
 mov cx, 10 /* Para el ejemplo, repetir 10 veces */
 ciclo:
   /* Demás instrucciones dentro del ciclo
   Importante: Recuerde que para el ciclo, se usa el registro cx.
   Por esta razón no se debe utilizar dentro del ciclo.
   */
loop ciclo /* Decrementar automáticamente cx y verificar si es
 mayor que cero. Si cx es mayor que cero, saltar a la
 etiqueta 'ciclo' En caso contrario, continuar la
```

ejecución en la instrucción siguiente*/

Ambas estrategias son válidas. Generalmente se utiliza la instrucción 'loop' y el registro CX. Sin embargo, cuando la condición es más compleja, se utiliza la primera aproximación.

Ver también:

- Entorno de ejecución en IA-32
- Uso de la Pila en IA-32
- Creación y uso de rutinas
- Uso de los servicios de la BIOS
- Modos de Operación de procesadores IA-32
- Organización de Memoria en Procesadores IA-32
- Entorno de ejecución en IA-32
- Paso a Modo Protegido en Procesadores IA-32
- Tabla Global de Descriptores - GDT
- http://www.gnu.org/software/grub/ Página oficial de GRUB (Enlace externo)

Uso de la Pila en IA-32

Autor:

Erwin Meza Vega emezav@gmail.com

Información del Proyecto : Programación de procesadores de arquitectura IA-32 : Ensamblador para procesadores IA-32 : Uso de la Pila

La pila es un elemento muy importante dentro de la arquitectura IA-32. Se puede usar de forma explícita, y es usada de forma implícita cuando se invoca a rutinas (instrucción **call**) y cuando se gestiona una interrupción.

Organización de la Pila

Los procesadores de arquitectura IA-32 proporcionan tres registros para manipular la pila. Estos registros son:

- Registro de segmento SS (stack segment)
- Registro de propósito general ESP (stack pointer)
- Registro de propósito general EBP (base pointer).

La pila tiene la siguiente organización:

```
Disposición de la pila en IA-32

+-----------------+
|                 |
+-----------------+
|  valor          |<--+
+-----------------+   |
|  valor          |<--+-----Valores almacenados en la pila
+-----------------+   |
|  valor          |<--+
+-----------------+ <-- Tope de la pila. ESP = desplazamiento desde la
|                 |                           base de la pila
|  Espacio        |
|  disponible en  |
|  la pila        |
|                 |
|                 |
+-----------------+ <---  Base de la pila. SS apunta a la Base de la Pila
|                 |
|                 |
+-----------------+
```

Es importante resaltar que en los procesadores IA-32 la pila crece de una posición más alta a una posición más baja en la memoria. Es decir, cada vez que se almacena un byte o un word en la pila, ESP se decrementa y apunta a una dirección menor en la memoria.

Operaciones sobre la pila

Las operaciones sobre la pila dependen del modo de ejecución del procesador, de la siguiente forma:

- Modo real: En modo real (16 bits), la unidad mínima de almacenamiento en la pila es un word. Esto significa que si un programa intenta almacenar un byte en la pila (que es válido), el procesador insertará automáticamente un byte vacío para mantener una pila uniforme con unidades de almacenamiento de dos bytes (16 bits).

- Modo protegido: En este modo la unidad mínima de almacenamiento es un doubleword (4 bytes, 32 bits). Si se almacena un byte o un word, automáticamente se insertan los bytes necesarios para tener una pila uniforme con unidades de almacenamiento de cuatro bytes (32 bits). En el modo de compatibilidad de 64 bits también se usa esta unidad de almacenamiento.
- Modo de 64 bits: La unidad mínima de almacenamiento es un quadword (8 bytes, 64 bits). Tambien se pueden almacenar bytes, words o doublewords, ya que el procesador inserta los bytes necesarios en cada caso para tener unidades uniformes de 8 bytes (64 bits).

A continuación se presentan las instrucciones básicas para el manejo de la pila.

Instrucción push

La instrucción push almacena un valor inmediato (constante), o el valor de un registro en el tope de la pila. El apuntador al tope de la pila (ESP) se decrementa en el número de bytes almacenados.

La siguiente instrucción en modo real permite almacenar el valor del registro AX en la pila (un word):

```
pushw %ax
```

En sintaxis Intel, simplemente se omite el sufijo 'w':

```
push ax
```

Esta instrucción almacena en la pila el valor de AX (2 bytes), y decrementa el valor de SP en 2. La pila lucirá así:

```
      +-----------------+
      |  valor          |
      +-----------------+ <-- ESP antes de la instrucción push AX
      |  valor de AX    |  push AX Coloca el valor de AX en la pila (2 bytes)
      +-----------------+ <-- ESP = Nuevo tope de la pila luego de push AX
      |                 |
      |  Espacio        |
      |  disponible en  |
      |  la pila        |
      |                 |
      |                 |
      +-----------------+ <---  Base de la pila. SS apunta a la Base de la Pila
      |                 |
      |                 |
     +-----------------+
```

La instrucción push permite almacenar en la pila los valores de los registros del procesador y también valores inmediatos.

La siguiente instrucción en modo protegido de 32 bits permite almacenar el valor del registro EAX en la pila (un doubleword, 32 bits):

```
pushl %eax
```

En sintaxis Intel, simplemente se omite el sufijo 'l':

```
push eax
```

Esta instrucción almacena en la pila el valor de EAX (4 bytes), y decrementa el valor de ESP en 4. La pila lucirá así:

```
      +-----------------+
      |  valor          |
      +-----------------+ <-- ESP antes de la instrucción push EAX
      |  valor de EAX   | push EAX coloca el valor de EAX en la pila (4 bytes)
      +-----------------+ <-- ESP = Nuevo tope de la pila luego de push EAX
      |                 |
      |  Espacio        |
      |  disponible en  |
      |  la pila        |
      |                 |
      |                 |
      +-----------------+ <---  Base de la pila. SS apunta a la Base de la Pila
      |                 |
      |                 |
     +-----------------+
```

Instrucción pop

Por su parte, la instrucción pop retira un valor de la pila (word, doubleword o quadword según el modo de operación y el sufijo de la instrucción), lo almacena en el registro destino especificado e incrementa SP (ESP o RSP según el modo de operación) en 2, 4 o 8.

Se debe tener en cuenta que luego de sacar un valor de la pila, no se puede garantizar que el valor sacado se conserve en la pila.

Por ejemplo, para sacar un valor del tope de la pila en modo protegido de 32 bits y almacenarlo en el registro EAX, se usa la siguiente instrucción:

```
 popl %eax
```

En sintaxis Intel:

```
pop eax
```

Esta instrucción saca del tope de la pila un doubleword (cuatro bytes) y los almacena en el registro EAX, como lo muestra la siguiente figura.

```
      +-----------------+
      |  valor          |
      +-----------------+ <-- ESP después de pop EAX (ESP = ESP + 4)
      |  valor          | pop EAX : Este valor se almacena en EAX (4 bytes)
      +-----------------+ <-- ESP antes de pop EAX
      |                 |
      |  Espacio        |
      |  disponible en  |
      |  la pila        |
      |                 |
      |                 |
      +-----------------+ <---  Base de la pila. SS apunta a la Base de la Pila
      |                 |
      |                 |
     +-----------------+
```

La instrucción pop toma el valor del tope de la pila y lo almacena en el registro de destino especificado en la misma operación. Se debe tener en cuenta ue luego de extraer un valor de la pila no se garantiza que aún siga allí.

En modo real la instrucción pop retira dos bytes de la pila, y en modo de 64 bits retira 8 bytes de la pila.

Instrucción pushf

Esta instrucción toma el valor del registro EFLAGS y lo almacena en la pila.

```
      +-----------------+
      |  valor          |
      +-----------------+ <-- ESP antes de la instrucción pushf
      | valor de EFLAGS | pushf coloca el valor de EFLAGS en la pila (4 bytes)
      +-----------------+ <-- ESP = Nuevo tope de la pila luego de pushf
      |                 |
      |  Espacio        |
      |  disponible en  |
      |  la pila        |
      |                 |
      |                 |
      +-----------------+ <---  Base de la pila. SS apunta a la Base de la Pila
      |                 |
      |                 |
     +-----------------+
```

En modo real solo se puede tener acceso a los 16 bits menos significativos de EFLAGS, por lo cual solo se ocupan 2 bytes en la pila y SP se decrementa en 2. En modo de 64 bits se ocupan 8 bytes (64 bits) para el registro RFLAGS.

Instrucción popf

Esta instrucción toma el valor almacenado en el tope de la pila y lo almacena en el registro EFLAGS (32 bits), los 16 bits menos significativos de EFLAGS en modo real y RFLAGS en modo de 64 bits.

```
      +-----------------+
      |  valor          |
      +-----------------+ <-- ESP después de popf (ESP = ESP + 4)
      |  valor          | popf : Este valor se almacena en EFLAGS (4 bytes)
      +-----------------+ <-- ESP antes de popf
      |                 |
      |  Espacio        |
      |  disponible en  |
      |  la pila        |
      |                 |
      |                 |
      +-----------------+ <---  Base de la pila. SS apunta a la Base de la Pila
      |                 |
      |                 |
     +-----------------+
```

Instrucción pusha

Esta instrucción almacena el valor de los registros de propósito general en la pila. De acuerdo con el modo de operación del procesador, se almacenarán los registros en el siguiente orden:

- En modo protegido de 32 bits, se almacenan EAX, ECX, EDX, EBX, valor de ESP antes de pusha, EBP, ESI y EDI.
- En modo real (16 bits), se almacenan AX, CX, DX, BX, valor de SP antes de pusha, BP, SI y DI.
- En modo de 64 bits, se almacenan RAX, RCX, RDX, RBX, valor de RSP antes de pusha, RBP, RSI y RDI.

Así, en modo protegido de 32 bits **cada** valor almacenado en la pila tomará cuatro bytes. En modo real, tomará dos bytes y en modo de 64 bits **cada** valor tomará 8 bytes.

A continuación se presenta un diagrama del funcionamiento de la instrucción pusha en modo protegido de 32 bits.

```
+--------------------+
|  valor             |
+--------------------+ <-- ESP antes de pusha
|  valor de EAX      |     |
+--------------------+     |
|  valor de ECX      |     |
+--------------------+     |
|  valor de EDX      |     |
+--------------------+     |
|  valor de EBX      |     |
+--------------------+     |
|  ESP antes de pusha| <---+
+--------------------+
|  valor de EBP      |
+--------------------+
|  valor de ESI      |
+--------------------+
|  valor de EDI      |
+--------------------+ <-- ESP después de pusha (ESP = ESP - 32) este es
|                    |     el nuevo tope de la pila.
|  Espacio           |
|  disponible en     |
|  la pila           |
|                    |
|                    |
+--------------------+ <---  Base de la pila. SS apunta a la Base de la Pila
|                    |
|                    |
+--------------------+
```

Instrucción popa

Esta instrucción extrae de la pila ocho valores, y los almacena en los registros de propósito general, en el siguiente orden (inverso al orden de pusha):

EDI, ESI, EBP, ESP*, EBX, EDX, ECX, EAX

El valor de ESP sacado de la pila se descarta.

```
+--------------------+
|  valor             |
+--------------------+ <-- ESP después de popa ( ESP = ESP + 32)
|  valor             | ---> EAX
+--------------------+
|  valor             | ---> ECX
+--------------------+
|  valor             | ---> EDX
+--------------------+
|  valor             | ---> EBX
+--------------------+
|  valor             |  (este valor se descarta)
+--------------------+
|  valor             | ---> EBP
+--------------------+
|  valor             | ---> ESI
+--------------------+
|  valor             | ---> EDI
+--------------------+ <-- ESP andtes de pusha
|                    |
|  Espacio           |
|  disponible en     |
|  la pila           |
|                    |
|                    |
+--------------------+ <---  Base de la pila. SS apunta a la Base de la Pila
|                    |
|                    |
+--------------------+
```

Ver también:

- Ensamblador para procesadores IA-32
- Creación y uso de rutinas
- Entorno de ejecución en IA-32
- Modos de Operación de procesadores IA-32
- Organización de Memoria en Procesadores IA-32
- Entorno de ejecución en IA-32
- Paso a Modo Protegido en Procesadores IA-32
- Tabla Global de Descriptores - GDT
- http://www.gnu.org/software/grub/ Página oficial de GRUB (Enlace externo)

Creación y uso de rutinas

Autor:

Erwin Meza Vega emezav@gmail.com

Información del Proyecto : Programación de procesadores de arquitectura IA-32 : Ensamblador para procesadores IA-32 : Creación y uso de rutinas

En los lenguajes de alto nivel, además de bifurcaciones y ciclos, también se usan las rutinas como mecanismos para organizar la funcionalidad de un programa.

Para implementar rutinas, el procesador incluye entre otras las instrucciones **call** (invocar una rutina) y **ret** (retornar de la rutina).

Una rutina consta de varios componentes, entre los cuales sobresalen:

1. Nombre de la rutina: Símbolo dentro de la sección de texto que indica el inicio de la rutina.
2. Parámetros de entrada: Se utiliza la pila para pasar los parámetros a las funciones.
3. Dirección de retorno: Dirección a la cual se debe retornar una vez que se ejecuta la rutina

Por ejemplo, para definir una rutina llamada nombre_rutina, el código sería el siguiente:

```
nombre_rutina:/* Inicio de la rutina */
   /* Instrucciones de la rutina*/
   ...
   ...
   ...
   ret /* Fin de la rutina  (retornar) */
```

Es necesario notar que la definición de una etiqueta no necesariamente implica la definición de una rutina. El concepto de **"Rutina"** lo da el uso que se haga de la etiqueta. Si para saltar a una etiqueta se usa la instrucción de salto incondicional (jmp) o alguna instrucción de salto condicional (j..), esta no es una rutina. Si por el contrario, para saltar a una etiqueta se usa la instrucción call (ver explicación más adelante), y después de esa etiqueta existe una instrucción ret a la cual se llega sin importar la lógica de programación, entonces la etiqueta sí puede ser considerada una "Rutina".

Invocación a Rutinas

La llamada a una rutina se realiza por medio de la instrucción call (en sintaxis AT&T e Intel), especificando la etiqueta (el nombre de la rutina) definido en ensamblador:

```
call nombre_rutina
```

De esta forma, se estará invocando a la rutina nombre_rutina, sin pasarle parámetros.

También es posible, aunque poco común, realizar llamadas a rutinas que se encuentran en otros segmentos de memoria. En este caso se utiliza la instrucción lcall.

Parámetros de entrada de las rutinas

Si se desean pasar parámetros a una rutina, éstos se deben almacenar en la pila mediante la instrucción push, en el orden inverso en el que se van a utilizar en la rutina antes de ejecutar la instrucción call.

Por ejemplo, para invocar a una rutina y pasarle n parámetros (parámetro 1, parámetro 2, .. , parámetro n), primero se deben insertar los parámetros en orden inverso en la pila (del último al primero) antes de la instrucción call:

```
push parametro n
push parametro n-1
...
push parametro 2
push parametro 1
```

La cantidad de bytes que ocupa cada parámetro en la pila depende del modo de operación del procesador. En modo real, cada parámetro ocupa dos bytes (16 bits). En modo protegido, cuatro bytes (32 bits) y en modo de 64 bits cada parámetro ocupa ocho bytes.

Luego se utiliza la instrucción call, especificando el símbolo (la etiqueta) de la rutina que se desea ejecutar:

```
call nombre_rutina
```

La instrucción call almacena automáticamente en la pila dirección de memoria de la próxima instrucción a ejecutar luego del call (la dirección de retorno), y establece el registro EIP (instruction pointer) al desplazamiento en el segmento de código en la cual se encuentra definido el símbolo con el nombre de la rutina.

De esta forma, en el momento de llamar a una rutina, la pila se encuentra así:

```
      +--------------------+
      |  valor             |
      +--------------------+ <- ESP antes de insertar los parámetros y de
      | Parámetro N        |    invocar la rutina (call)
      +--------------------+
      | Parámetro N-1      |
      +--------------------+
      | .....              |
      +--------------------+
      | Parámetro 2        |
      +--------------------+
      | Parámetro 1        |
      +--------------------+
      | EIP                | -> Dirección de la siguiente instrucción a ejecutar
      | (Dir. de retorno)  |    al retornar de la rutina (dirección de retorno)
      +--------------------+ <-- ESP después de call
      |                    |
      |  Espacio           |
      |  disponible en     |
      |  la pila           |
      |                    |
      |                    |
      +--------------------+ <---  Base de la pila. SS apunta a la Base de la Pila
      |                    |
      |                    |
     +--------------------+
```

Retorno de una rutina

Cuando dentro de una rutina se busque retornar la ejecución al punto en el cual fue invocada, se debe usar la instrucción **ret** . Esta instrucción saca del tope de la pila la dirección de retorno, y establece el registro EIP con este valor. Se debe garantizar que el tope de la pila contiene una

dirección de retorno válida, o de lo contrario el procesador continuará su ejecución en otro punto e incluso puede llegar a causar una excepción.

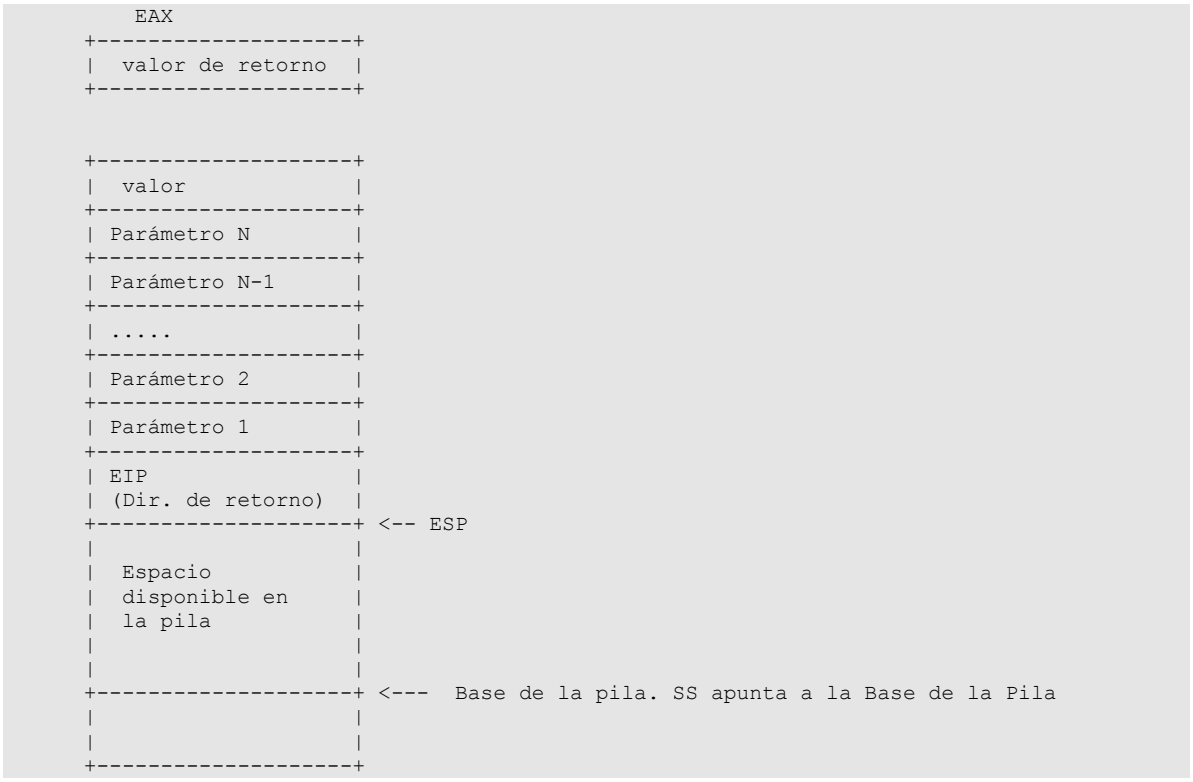
si se insertan valores en la pila dentro de la rutina, se deben extraer antes de ejecutar la instrucción ret.

Valor de retorno de las rutinas

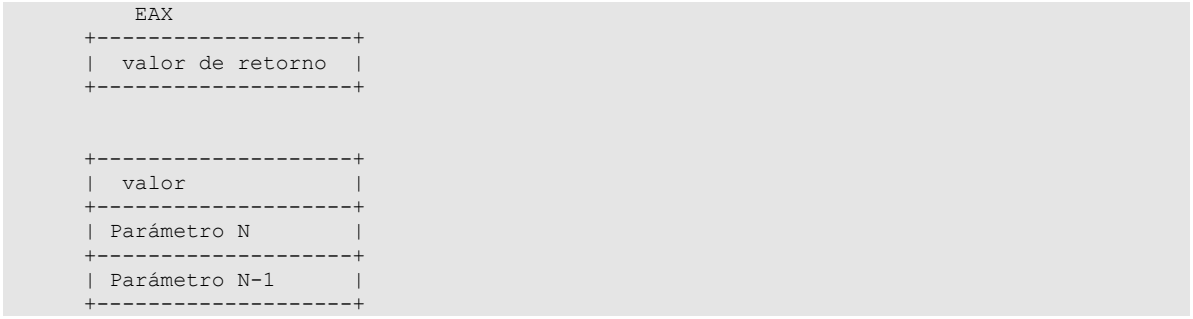
Es importante tener en cuenta que la instrucción ret difiere un poco de la instrucción return de lenguajes de alto nivel, en la cual se puede retornar un valor. En ensamblador, el valor de retorno por convención se almacena siempre en el registro AL, AX, EAX o RAX de acuerdo con el modo de operación del procesador. Así, una de las últimas instrucciones dentro de la rutina antes de ret deberá almacenar el valor de retorno en el registro EAX.

El siguiente diagrama ilustra el funcionamiento de la instrucción ret.

Antes de ejecutar la instrucción ret:



Después de ejecutar la instrucción ret



```
      | .....              |
      +--------------------+
      | Parámetro 2        |
      +--------------------+
      | Parámetro 1        |
      +--------------------+ <-- ESP
      | EIP                |
      | (Dir. de retorno)  |  --> EIP La ejecución continúa en esta dirección
      +--------------------+
      |                    |
      |  Espacio           |
      |  disponible en     |
      |  la pila           |
      |                    |
      |                    |
      +--------------------+ <---  Base de la pila. SS apunta a la Base de la Pila
      |                    |
      |                    |
     +--------------------+
```

Dado que al retornar de la rutina los parámetros aún se encuentran en la pila, es necesario avanzar ESP para que apunte a la posición de memoria en la cual se encontraba antes de insertar los parámetros. Para lograr este propósito se adiciona un número de bytes a ESP:

```
add esp, N
```

Donde N corresponde al número de bytes que se almacenaron en la pila como parámetros:

- En modo real cada parámetro ocupa dos bytes en la pila, por lo cual se deberá sumar 2 * el número de parámetros a SP.
- En modo protegido cada parámetro ocupa cuatro bytes en la pila, por lo que se deberá sumar 4 * el número de parámetros a ESP.
- En modo de 64 bits se deberá sumar 8 * el número de parámetros a RSP.

De forma general, el formato para invocar una rutina que recibe **N** parámetros es el siguiente:

```
push parametroN
push parametroN-1
...
push parametro2
push parametro1
call nombre_rutina
add esp, N
```

Ejemplo de implementación de Rutinas en Ensamblador

A continuación se muestra la implementación general de una rutina en lenguaje ensamblador. Dentro de la rutina se crea un "marco de pila", necesario para manejar correctamente las variables que fueron pasadas como parámetro en la pila y las variables locales.

El concepto de "marco de pila" se explicará tomando como base la plantilla de rutina en modo real. En este modo se usan los registros SP, BP e IP.

En los demás modos de operación del procesador el marco de pila funciona en forma similar, pero se deben expandir los registros a sus equivalentes en 32 y 64 bits, y se deberá sumar el número de bytes de acuerdo con el tamaño de los registros.

En sintaxis AT&T:

```
nombre rutina:
   pushw %bp   /*Almacenar %bp en la pila*/
   movw %sp, %bp  /*Establecer %bp con el valor de %sp*/
```

```
    /*Ya se ha creado un marco de pila*/
   ...
   (instrucciones de la rutina)
   ...

    /*Cerrar el marco de pila:*/
   movw %bp, %sp  /*Mover %bp a %sp*/
   popw %bp/*Recuperar el valor original de %bp */
```

ret /* Retornar de la rutina */

En sintaxis Intel:

```
nombre rutina:
   push bp   /*Almacenar bp en la pila*/
   mov bp, sp  /*Establecer bp con el valor de sp*/
    /*Ya se ha creado un marco de pila*/
   ...
   (instrucciones de la rutina)
   ...

    /*Cerrar el marco de pila:*/
   mov sp, bp  /*Mover bp a sp*/
   pop bp/*Recuperar el valor original de %bp */
```

ret /* Retornar de la rutina*/

Explicación de la plantilla de rutina

En esta explicación se supone que el código ha insertado los parámetros en la pila e invocó la instrucción call para ejecutar la rutina.

Con la instrucción

```
   pushw %bp   /* Sintaxis AT&T */
```

ó

```
 push bp    /* Sintaxis Intel */
```

La pila queda dispuesta de la siguiente forma:

```
      +--------------------+
      |  valor             |
      +--------------------+
      | Parámetro N        |
      +--------------------+
      | Parámetro N-1      |
      +--------------------+
      | .....              |
      +--------------------+
      | Parámetro 2        |
      +--------------------+
      | Parámetro 1        |
      +--------------------+
      |  IP                |
      | (Dir. de retorno)  |
      +--------------------+
      | Valor de  BP       | Se almacena el valor de BP en la pila
      +--------------------+ <-- SP
      |                    |
      |  Espacio           |
      |  disponible en     |
      |  la pila           |
      |                    |
```

```
      |                    |
      +--------------------+ <---  SS apunta a la base de la pila
      |                    |
      |                    |
      +--------------------+
```

Observe que SP apunta ahora a la posición de memoria en la pila en la cual se almacenó el valor que tenía BP originalmente. Esto permite modificar BP, y recuperar su valor original luego de terminadas las instrucciones de la rutina y antes de retornar al punto desde el cual se invocó la rutina.

La instrucción

```
movw %sp, %bp  /* Sintaxis AT&T */
```

ó

```
 mov bp, sp /* Sintaxis Intel */
```

Establece a EBP con el mismo valor de SP, con lo cual BP apunta a la misma dirección de memoria a la cual apunta SP:

```
      +--------------------+
      |  valor             |
      +--------------------+
      | Parámetro N        |
      +--------------------+
      | Parámetro N-1      |
      +--------------------+
      | .....              |
      +--------------------+
      | Parámetro 2        |
      +--------------------+
      | Parámetro 1        |
      +--------------------+
      |  IP                |
      | (Dir. de retorno)  |
      +--------------------+
      | Valor de  BP       | Se almacena el valor de BP en la pila
      +--------------------+ <-- SP  <-- BP
      |                    |
      |  Espacio           |
      |  disponible en     |
      |  la pila           |
      |                    |
      |                    |
      +--------------------+ <---  SS apunta a la base de la pila
      |                    |
      |                    |
      +--------------------+
```

Con esta instrucción se termina el proceso de crear el marco de pila. Ahora es totalmente seguro decrementar el valor de SP con el propósito de crear espacios para las variables locales a la rutina, o de insertar otros valores en la pila.

Por ejemplo, la instrucción

```
subw $4, %sp /* Sintaxis AT&T */
```

ó

```
sub sp, 4  /* Sintaxis Intel */
```

Crea un espacio de 4 bytes (2 words) en la pila, que ahora se encontrará así:

```
      +--------------------+
      |  valor             |
      +--------------------+
      | Parámetro N        |
      +--------------------+
      | Parámetro N-1      |
      +--------------------+
      | .....              |
      +--------------------+
      | Parámetro 2        |
      +--------------------+
      | Parámetro 1        |
      +--------------------+
      |  IP                |
      | (Dir. de retorno)  |
      +--------------------+
      | Valor de  BP       |
      +--------------------+ <-- BP
      | Espacio disponible |
      +--------------------+
      | Espacio disponible |
      +--------------------+ <-- SP
      |                    |
      |  Espacio           |
      |  disponible en     |
      |  la pila           |
      |                    |
      |                    |
      +--------------------+ <---  SS apunta a la base de la pila
      |                    |
      |                    |
      +--------------------+
```

Observe que ESP se decrementa, pero BP sigue apuntando al inicio del marco de pila. Por esta razón, el puntero BP se denomina Base Pointer (puntero base), ya que con respecto a él es posible acceder tanto los parámetros enviados a la rutina, como las variables locales creadas en ésta.

Por ejemplo, la instrucción

```
movw 4(%bp), %ax /* Sintaxis AT&T */
```

ó

```
mov ax, WORD PTR [ bp + 4 ] /* Sintaxis Intel */
```

Mueve el contenido de la memoria en la posición SS:[BP + 4] al registro AX, es decir que almacena el primer parámetro pasado a la rutina en el registro AX.

A continuación se presenta de nuevo el estado actual de la pila, para visualizar los diferentes desplazamientos a partir del registro BP.

```
      +--------------------+
      |  valor             |
      +--------------------+
      | Parámetro N        |
      +--------------------+ <-- BP + 2*N
      | Parámetro N-1      |
      +--------------------+ ...
      | .....              |
      +--------------------+ <-- BP + 8
      | Parámetro 2        |
      +--------------------+ <-- BP + 6
      | Parámetro 1        |
      +--------------------+ <-- BP + 4
      |  IP                |
      | (Dir. de retorno)  |
      +--------------------+ <-- BP + 2
      | Valor de  BP       |
```

```
      +--------------------+ <-- BP
      | Espacio disponible |
      +--------------------+ <-- BP - 1
      | Espacio disponible |
      +--------------------+ <-- SP  <-- BP - 4
      |                    |
      |  Espacio           |
      |  disponible en     |
      |  la pila           |
      |                    |
      |                    |
      +--------------------+ <---  SS apunta a la base de la pila
      |                    |
      |                    |
      +--------------------+
```

De esta forma, la instrucción

```
movw %ax, -2(%bp) /* Sintaxis AT&T */
```

ó

```
mov WORD PTR [ bp - 2 ], ax /* Sintaxis Intel */
```

Almacena el valor del registro AX en el primer word de espacio de la pila.

Se puede observar que si se crea un marco de pila estándar con las instrucciones mencionadas, siempre el primer parámetro que se paso a la rutina se encontrará en (BP + 4), el segundo en (BP + 6) y así sucesivamente.

Se debe tener en cuenta que si dentro de la rutina se modifica el valor del registro BP, se deberá almacenar su valor con anterioridad en la pila o en otro registro.

Cerrar el marco de Pila

Al finalizar la rutina se deberá cerrar el marco de pila creado. La instrucción

```
 movw %bp, %sp /* Sintaxis AT&T */
```

ó

```
mov sp, bp  /* Sintaxis Intel */
```

Cierra el espacio creado para las variables locales, al apuntar SP a la misma dirección de memoria en la pila a la que BP. Luego de esta instrucción la pila lucirá así:

```
      +--------------------+
      |  valor             |
      +--------------------+
      | Parámetro N        |
      +--------------------+
      | Parámetro N-1      |
      +--------------------+
      | .....              |
      +--------------------+
      | Parámetro 2        |
      +--------------------+
      | Parámetro 1        |
      +--------------------+
      |  IP                |
      | (Dir. de retorno)  |
      +--------------------+
      | Valor de  BP       |
      +--------------------+ <-- BP  <-- SP
      |                    |
      +--------------------+
      |                    |
```

```
      +--------------------+ <-- SP antes de cerrar el marco de pila
      |                    |
      |  Espacio           |
      |  disponible en     |
      |  la pila           |
      |                    |
      |                    |
      +--------------------+ <---  SS apunta a la base de la pila
      |                    |
      |                    |
      +--------------------+
```

En este momento ya no es seguro acceder a los valores almacenados en el espacio para variables locales.

Ahora se deberá recuperar el valor original de BP antes de crear el marco de pila:

```
popw %bp /* Sintaxis AT&T */
```

ó

```
pop bp /* Sintaxis Intel */
```

Con ello la pila se encontrará en el siguiente estado:

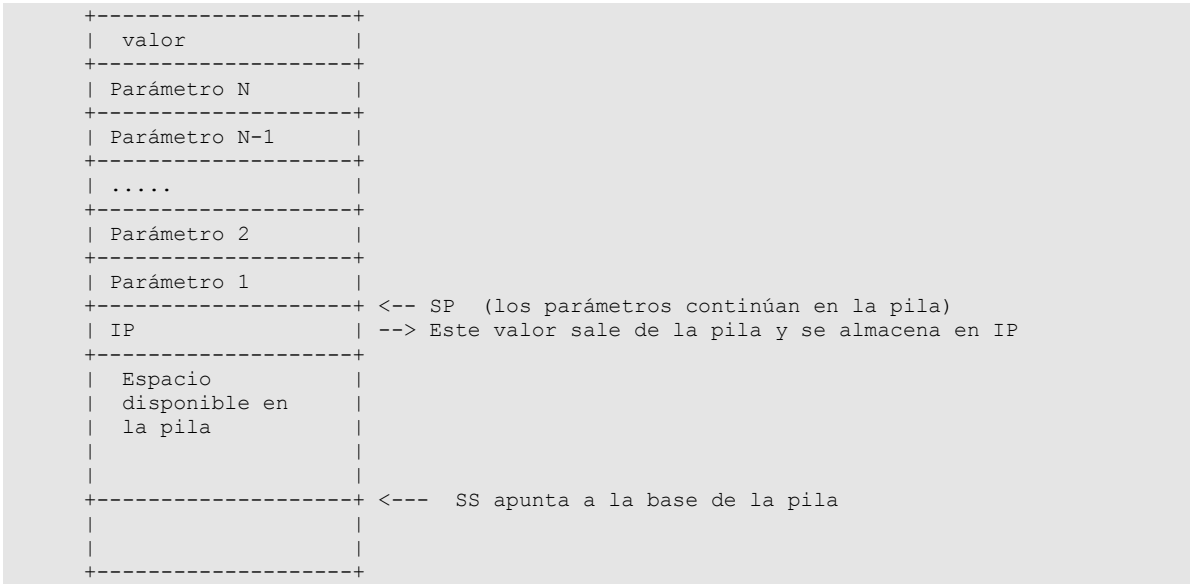
```
      +--------------------+
      |  valor             |
      +--------------------+
      | Parámetro N        |
      +--------------------+
      | Parámetro N-1      |
      +--------------------+
      | .....              |
      +--------------------+
      | Parámetro 2        |
      +--------------------+
      | Parámetro 1        |
      +--------------------+
      |  IP                |
      | (Dir. de retorno)  |
      +--------------------+ <-- SP
      | Valor de  BP       | --> Este valor sale de la pila y se almacena en BP
      +--------------------+
      |                    |
      |  Espacio           |
      |  disponible en     |
      |  la pila           |
      |                    |
      |                    |
      +--------------------+ <---  SS apunta a la base de la pila
      |                    |
      |                    |
      +--------------------+
```

Ahora SP apunta a la dirección de retorno de la rutina (donde debe continuar la ejecución). La instrucción

```
ret
```

Toma de la pila la dirección de retorno (la dirección de memoria de la instrucción siguiente a la cual se llamó la rutina mediante call), y realiza un jmp a esa dirección.

Note que luego de retornar de la rutina, la pila se encontrará en el siguiente estado:



Por esta razón es necesario avanzar SS en un valor igual al número de bytes que se enviaron como parámetro a la rutina. Si se enviaron N parámetros a la pila, el numero de bytes que se deberán sumar a sp son 2 * N (En modo real cada parámetro ocupa un word = 2 bytes).

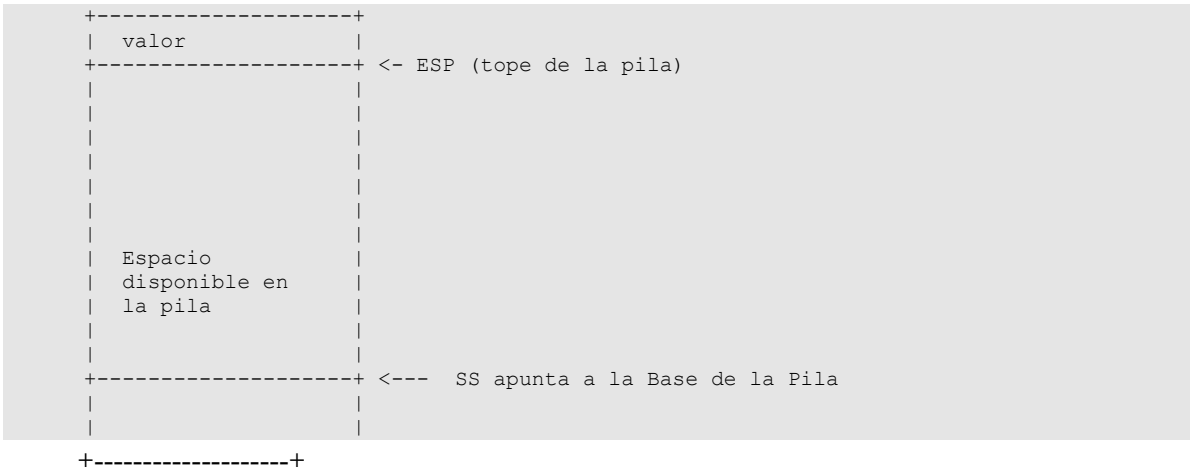
```
addw $M, %sp /* Sintaxis AT&T */
```

ó

```
add sp, M /* Sintaxis Intel */
```

Donde M representa el número de bytes a desplazar SP.

Con estas instrucciones la pila se encontrará en el mismo estado que antes de invocar la rutina:



Plantilla de Rutina en Modo Protegido de 32 bits

La plantilla de una rutina en modo protegido de 32 bits es muy similar a la de modo real. La principal diferencia entre las dos consiste en el tamaño de los registros, que se expanden de 16 a 32 bits (BP se expande a EBP y SP se expande a ESP).

También es importante recordar que cada parámetro almacenado en la pila ocupa 4 bytes (32 bits), por lo cual el valor que se debe sumar a ESP después de retornar de la rutina es 4 * el número de parámetros insertados.

En sintaxis AT&T

```
nombre_rutina:
   pushl %ebp   /*Almacenar %ebp en la pila*/
   movw %esp, %ebp  /*Establecer %ebp con el valor de %esp*/
    /*Ya se ha creado un marco de pila*/
   ...
   (instrucciones de la rutina)
   Por ejemplo, para obtener el primer parámetro de la pila sería:
   mov 8(%ebp), %eax
   ...

    /*Cerrar el marco de pila:*/
   movw %ebp, %esp  /*Mover %ebp a %esp*/
   popw %ebp/*Recuperar el valor original de %ebp */
```

ret /* Retornar de la rutina */

En sintaxis Intel

```
nombre rutina:
   push ebp   /*Almacenar ebp en la pila*/
   mov ebp, esp  /*Establecer ebp con el valor de esp*/
    /*Ya se ha creado un marco de pila*/
   ...
   (instrucciones de la rutina)
   Por ejemplo, para obtener el primer parámetro de la pila sería:
   mov eax, [ebp + 8]
   ...

    /*Cerrar el marco de pila:*/
   mov esp, ebp  /*Mover ebp a esp*/
   pop ebp/*Recuperar el valor original de ebp */
```

ret /* Retornar de la rutina*/

Ver también:

- Uso de la Pila en IA-32
- Ensamblador para procesadores IA-32
- Entorno de ejecución en IA-32
- Modos de Operación de procesadores IA-32
- Organización de Memoria en Procesadores IA-32
- Entorno de ejecución en IA-32
- Paso a Modo Protegido en Procesadores IA-32
- Tabla Global de Descriptores - GDT
- http://www.gnu.org/software/grub/ Página oficial de GRUB (Enlace externo)

Uso de los servicios de la BIOS

Autor:

Erwin Meza Vega emezav@gmail.com

Información del Proyecto : Programación de procesadores de arquitectura IA-32 : Ensamblador para procesadores IA-32 : Servicios de la BIOS

Breve reseña de la BIOS

La BIOS (Basic Input-Output System) ofrece una serie de servicios, en forma de interrupciones por software que pueden ser invocadas desde el código de un sector de arranque o de un sistema operativo que se ejecuta en Modo Real. Este documento presenta los conceptos básicos relacionados con el uso de los servicios proporcionados por la BIOS.

Cuando se enciende el computador o se pulsa el boton RESET, el procesador pasa el control a la BIOS. La BIOS contiene una serie de rutinas encargadas de establecer un entorno inicial para la ejecución del procesador en modo real. Sus principales tareas son:

Realizar un chequeo inicial del hardware (procesador, memoria RAM y dispositivos de entrada/salida)

Mapear algunos dispositivos de entrada / salida en memoria (por ejemplo, la memoria de video de modo texto a color en la direccion 0xB8000

Establecer las rutinas básicas de manejo de interrupción para los dispositivos que ha detectado, las rutinas para el manejo de las excepciones del procesador y las rutinas para los servicios proporcionados por la BIOS.

En la arquitectura IA-32, las interrupciones pueden ser de tres tipos: interrupciones lanzadas por el hardware (por ejemplo interrupciones del teclado, del mouse, etc), interrupciones generadas por software (mediante la instrucción int **N** , donde N es el número de la interrupción), y condiciones de error del procesador, conocidas como excepciones (por ejemplo, división por cero).

La BIOS configura la Tabla de Descriptores de Interrupción (Interrupt Descriptor Table, IDT) al inicio de la memoria RAM. Esta tabla contiene exactamente 256 entradas de 4 bytes, las cuales contienen apuntadores a rutinas de manejo de interrupción. Cada apuntador consta de dos valores, que representan los valores de deben tomar **CS** e **IP** para apuntar a la rutina específica que maneja la interrupción.

Manejo de Interrupciones

Cuando ocurre la interrupcion N, el procesador automáticamente almacena en la pila el valor actual de CS e EI (es decir la dirección de la siguiente instrucción que se debe ejecutar luego de completar la rutina de manejo de interrupción). Luego, busca la N-ésima entrada de la IDT, que se encuentra exactamente a N * 4 bytes desde su inicio (dado que cada entrada ocupa exactamente 4 bytes). Esta entrada contiene los valores que debe tomar CS e IP para ejecutar la rutina para manejar la interrupción N.

Cuando la rutina de manejo de interrupción termina,el procesador obtiene de la pila los valores de CS e IP que permiten retornar al punto de ejecución en el cual se encontraba el procesador en el momento en que ocurrió la interrupción.

En la actualidad, los servicios de la BIOS sólo se utilizan mientras el código del sector de arranque ejecuta la funcionalidad necesaria para cargar el kernel y pasar a modo protegido de 32 bits. Una

vez en modo protegido, kernel completa la carga del sistema operativo, en el cual se incluye el código específico para acceder a los dispositivos de entrada y salida sin necesidad de usar los servicios de la BIOS.

Uso básico de los servicios de la BIOS

Para usar los servicios de la BIOS en modo real, el código debe establecer primero los registros del procesador en unos valores específicos para cada servicio. Luego se debe invocar una interrupción por software, por medio de la instrucción int N (donde N es el numero de la interrupción).

A continuación se presentan los servicios mas comunes que se usan en modo real. Para cada uno de ellos, se especifica el número de la interrupción que se debe invocar, y los valores de los registros del procesador que deben ser establecidos antes de invocar la interrupcion y los valores que retorna la BIOS.

(int 0x10)

La BIOS ofrece una serie de servicios de video por medio de la interrupción 0x10. A continuación se presentan los servicios de video más importantes:

Establecer el modo de video

```
AH = 0x00
AL = modo de video
```

Algunos modos de video válidos para cualquier tarjeta de video VGA son:

```
   AL = 0x00  Modo texto de 25 filas, 40 columnas, blanco y negro
      = 0x01  Modo texto de 25 filas, 40 columnas, 16 colores
      = 0x02  Modo texto de 25 filas, 80 columas, escala de grises
      = 0x03  Modo texto de 25 filas, 80 columnas, 16 colores
      = 0x04  Modo grafico de 320x200 pixels, 4 colores por pixel
      = 0x0D  Modo grafico de 320x200 pixels, 16 colores por pixel
      = 0x0E  Modo grafico de 640x200 pixels, 16 colores por pixel
      = 0x11  Modo grafico de 640x480 pixels, blanco y negro
      = 0x12  Modo grafico de 640x480 pixels, 16 colores por pixel
      = 0x13  Modo grafico de 320x200 pixels, 256 colores por pixel
```

Estos modos dependen del tipo de tarjeta grafica (CGA, EGA, VGA). El modo de video por defecto es Modo Texto, 25 filas, 80 columnas, 16 colores.

Leer la posición actual del cursor

```
ah  = 0x03
```

La BIOS almacena la fila actual del cursor en el registro DH, y la columna actual en el registro DL.

Establecer la posicion del cursor

```
AH = 02
DH = fila
DL = columna
```

Esta rutina establece la posicion del cursor a los valores especificados. Todas las posiciones son relativas a 0,0, que representa la esquina superior de la pantalla. Si no se configura de otra forma, la pantalla de video de solo texto ocupa 25 filas por 80 columnas (fila 0 a 24, columna 0 a 79).

Escribir un caracter en modo 'terminal' (modo texto)

```
    AH = 0x0E
    AL = Caracter ascii que se desea escribir
```

Esta rutina imprime el caracter en la pantalla. Los caracteres Backspace (0x08), Fin de línea (0x0A) y Retorno de Carro (0x0D) se tratan de forma consistente. Igualmente, se actualiza la posición actual del cursor.

Servicios de teclado(int 0x16)

La BIOS ofrece una serie de servicios de teclado por medio de la interrupción 0x16. A continuación se presentan los servicios de teclado mas importantes:

Leer un caracter de teclado

```
    AH = 0x00
```

La BIOS almacena el caracter ASCII leido en el registro AL, y el código de escaneo (Scan Code) en el registro AH. Si se presionó una tecla de funcion especial (shift, esc, F1, etc), AL contiene 0x00.

Leer el estado del teclado

```
    AH = 0x01
```

Si no existe una tecla presionada, AX toma el valor de cero. Si existe una tecla presionada, AL contiene el código ASCII de la tecla, y AH contiene el código de escaneo (Scan Code). Tambien si existe una tecla presionada, el bit ZF (Zero Flag) del registro FLAGS se establece en 0.

Leer el estado de 'shift'

```
    AH = 0x02
```

```
La BIOS establece los bits del registro AL con el siguiente formato:
```

```
    ¦7¦6¦5¦4¦3¦2¦1¦0¦
    ¦ ¦ ¦ ¦ ¦ ¦ ¦ +---- right shift está presionado
    ¦ ¦ ¦ ¦ ¦ ¦ +----- left shift está presionado
    ¦ ¦ ¦ ¦ ¦ +------ CTRL está presionado
    ¦ ¦ ¦ ¦ +------- ALT está presionado
    ¦ ¦ ¦ +-------- Scroll Lock está activo
    ¦ ¦ +--------- Num Lock está activo
    ¦ +---------- Caps Lock está activo
```

```
   +----------- Insert está activo
```

Servicios de disco (int 0x13)

Por medio de la interrupción 0x13, la BIOS ofrece una serie de servicios para los discos. A continuación solo se presenta el servicio que permite leer sectores del disco.

Leer sectores de disco

```
AH = 0x02
AL = Numero de sectores a leer
CH = numero de la pista / cilindro  * ver nota
CL = numero del sector * ver nota
DH = numero de la cabeza
DL = numero del drive (0x00=floppy A:, 0x01=floppy B:, 80h=disco primario,
                       81h=disco  secundario)
```

ES:BX = Apuntador a la posición de memoria en la cual se leen los datos

Nota:

El formato del registro CX (CH:CL) es el siguiente:

```
¦F¦E¦D¦C¦B¦A¦9¦8¦7¦6¦5-0¦  CX
¦ ¦ ¦ ¦ ¦ ¦ ¦ ¦ ¦ ¦  +-----  Numero del sector (6 bits)
¦ ¦ ¦ ¦ ¦ ¦ ¦ ¦ +---------  2 bits mas significativos de la pista / cilindro
+-----------------------  8 bits menos significativos de la pista / cilindro
```

La BIOS lee el número de sectores especificados en el registro AL, comenzando en el sector, el cilindro y la cabeza especificados en los registros CL, CH y DH del disco especificado en DL a la posicion de memoria apuntada por ES:BX. La BIOS entonces almacena el AH el estado de la lectura, en AL el nÚmero de lectores que se pudieron leer, y establece el bit CF (Carry Flag) del registro FLAGS en 0 si la lectura fue exitosa o 1 si ocurriÓ un error.

Las lecturas de disco se deberían re-intentar al menos tres veces, para permitir que el controlador del disco se ubique en los sectores que se desean leer. Debido a que en estos ejemplos se usa una imagen de disco, se supone que la lectura nunca falla. En caso que la lectura falle, el código entra en un ciclo infinito.

Ejemplo de uso de servicios de la BIOS

Los siguientes ejemplos asumen que el código se está ejecutando en modo real (por ejemplo este código es ejecutado por un sector de arranque cargado por la BIOS en la dirección 0x7C00).

Imprimir el carácter '@' por pantalla:

```
mov ah, 0x0E /* Servicio a usar */
mov al, 0x40 /* Caracter '@' */
int 0x10 /* Invocar la interrupción 0x10 : servicios de video de la BIOS */
```

Otra versión del código para imprimir un carácter por pantalla:

```
mov ax, 0x0E40 /* AH = 0x0E, AL = 0x40 */
int 0x10 /* Invocar la interrupción 0x10 : servicios de video de la BIOS */
```

Leer un carácter del teclado:

```
mov AH, 0x00 /* Servicio de teclado : leer un caracter */
int 0x16 /* Invocar la interrupción 0x16: servicios de teclado de la BIOS */
```

Leer el segundo sector del floppy (cilindro 0, pista 0, sector 2) a la dirección de memoria 0x1000

```
 mov AX, 0x100
 mov ES, AX /* ES = 0x100*/
 mov BX, 0x0000  /* BX = 0, por lo tanto ES:BX apunta a 0x1000 */

 mov AH, 0x2   /*  Servicio de disco: leer un sector */
 mov AL, 1 /* Número de sectores a leer: 1 */

 mov CH, 0x00 /* Pista / cilindro =  0 */
 mov DH, 0x00 /* Cabeza = 0*/
 mov CL, 0x02 /* Sector número 2 */

 mov DL, 0x00 /* Drive 0x00 = floppy, 0x80 = disco primario */
```

int 0x13

Ver también:

- Ensamblador para procesadores IA-32
- Uso de la Pila en IA-32
- Creación y uso de rutinas
- Entorno de ejecución en IA-32
- Modos de Operación de procesadores IA-32
- Organización de Memoria en Procesadores IA-32
- Entorno de ejecución en IA-32
- Paso a Modo Protegido en Procesadores IA-32
- Tabla Global de Descriptores - GDT
- http://www.gnu.org/software/grub/ Página oficial de GRUB (Enlace externo)
- http://docs.huihoo.com/help-pc/int.html Documentación de los servicios de la BIOS

Carga y Ejecución del Kernel compatible con Multiboot

Autor:

Erwin Meza Vega emezav@gmail.com

Información del Proyecto : Carga y Ejecución del Kernel

El kernel creado en este proyecto es compatible con la Especificación Multiboot. Todos los archivos que lo componen se compilan en un único archivo en formato ELF (el formato ejecutable de Linux) y se almacena en una imagen de disco duro. Esta imagen de disco ya tiene integrado a GRUB.

GRUB es un cargador de arranque que cumple con la Especificación Multiboot. De esta forma, si se desarrolla un kernel compatible con Multiboot, podrá ser cargado por GRUB sin ningún inconveniente.

El archivo ejecutable del kernel se organiza para que sea cargado por GRUB a la dirección de memoria 0x100000, y para que su ejecución comience en la etiqueta **start** del archivo src/start.S.

Imagen de disco del proyecto

La imagen de disco ya contiene al cargador de arranque GRUB en su sector de arranque maestro, y tiene una sola partición. Dentro de esta partición se encuentran los archivos necesarios para iniciar GRUB, y el código del kernel compilado.

La partición se encuentra formateada con el sistema de archivos ext2 (linux), y su contenido se muestra en el siguiente esquema:

```
/ -+    <-- Directorio raíz de la única partición en la imagen de disco
   |
   boot <-- Almacena los archivos de GRUB y el kernel compilado
        |
        |
        grub  <-- Almacena los archivos de GRUB
        |   |
        |   |
        | e2fs_stage_1_5  <-- Etapa 1.5 de GRUB. cargado por la etapa 1 de GRUB
        |   |                 Contiene el código para manejar el sistema de
        |   |                 archivos de la partición (ext2). Este archivo
        |   |                 es opcional, ya que al instalar GRUB este archivo
        |   |                 se copió en el sector adyacente al sector de
        |   |                 arranque.
        |   |
        | menu.lst    <-- Archivo de configuración leido por GRUB al arranque.
        |   |             especifica la configuración del menú que despliega
        |   |             GRUB al arranque y ubicación del kernel en
        |   |             (la imagen de) disco.
        | stage1      <-- Etapa 1 de GRUB. Este archivo es opcional, ya que se
        |   |             copió en el sector de arranque del disco al instalar
        |   |             GRUB.
        |   |             Carga la etapa 1.5 de GRUB. Después carga la
        |   |             etapa 2 de GRUB desde el disco y le pasa el control.
        |   |             Este archivo es opcional.
        |   |
        | stage2      <-- Etapa 2 de GRUB. Cargada por la etapa 1 de GRUB.
        |                 Configura el sistema y presenta el menú que
        |                 permite cargar el kernel.
        |                 Este archivo es obligatorio.
        |                 Cuando el usuario selecciona la única opción
```

```
|                 disponible: cargar y pasar el control el archivo kernel
|                 que se encuentra en el directorio /boot de la imagen
|                 de disco
|                 El kernel se carga a la dirección de memoria 0x100000
|                 (1 MB)
|
kernel   <-- Archivo que contiene el código compilado del kernel.
```

Papel de GRUB en la carga del Kernel

Al iniciar el sistema, la BIOS carga el código de GRUB almacenado en el sector de arranque de la imagen de disco. Cuando GRUB se instala en la imagen de disco, el contenido del archivo stage_1 se incluye dentro del sector de arranque.

El código del archivo stage1 cargado en memoria recibe el control. Este código a su vez carga en memoria la etapa 1.5 de GRUB, que ha sido copiada en la imagen del disco a continuación del sector de arranque (por esta razón el archivo e2fs_stage_1_5 también es opcional). La etapa 1.5 de GRUB contiene el código para acceder a particiones de disco con formato ext2.

El código de la etapa 1 y la etapa 1.5 de GRUB cargan la etapa 2 de GRUB (el archivo stage_2, que es obligatorio), y luego le pasan el control.

El código de la etapa 2 lee el archivo menu.lst, que contiene las opciones de la interfaz del menú que se presenta al usuario y las diferentes opciones de sistemas operativos a cargar. El contenido de este archivo se reproduce a continuación:

```
default 0
timeout 10
color cyan/blue white/blue

title Aprendiendo Sistemas Operativos
root (hd0,0)
kernel /boot/kernel
```

Básicamente se establece un menú en el cual la primera (y única) opción de arranque se define como opción por defecto. El parámetro **timeout** permite establecer el tiempo que espera GRUB antes de cargar automáticamente la opción de arranque definida por defecto.

Es posible definir múltiples opciones de arranque, si en (la imagen de) el disco se tienen instalados varios sistemas operativos. (Por ejemplo, Linux y Windows).

La única opción de arranque configurada consta de tres elementos:

- **title** permite especificar el texto de la opción de menú que presentará GRUB
- **root** especifica la partición en la cual se encuentra el kernel (hd0,0). Esta corresponde a el disco duro primario (hd0), primera (y única) partición (0).
- **kernel** Especifica la ruta dentro de la partición en la cual se encuentra el kernel del sistema operativo (el código compilado de este proyecto).

Con esta configuración, al arranque del sistema se presenta el familiar menú de GRUB con la opción "Aprendiendo Sistemas Operativos". Al seleccionar esta opción, GRUB carga el archivo /boot/kernel de la imagen de disco a la posición de memoria 0x100000 (1MB) y le pasa el control de la ejecución.

Muchas distribuciones actuales de Linux usan a GRUB como cargador de arranque, mediante un sistema similar (con algunos aspectos complejos adicionales) al que se presenta en este proyecto.

Carga de Módulos del Kernel

La Especificación Multiboot además establece que es posible cargar otros archivos necesarios para la inicialización del kernel. Estos archivos pueden ser ejecutables, imágenes, archivos comprimidos, etc. Si se desea agregar un módulo para que sea cargado por GRUB al inicio del sistema, se deberá copiar el archivo dentro de la imagen de disco (Generalmente en el mismo directorio en el cual se encuentra copiado el archivo del kernel), y se deberá adicionar para cada módulo una línea en la opción correspondiente en el archivo de configuración:

```
default 0
timeout 10
color cyan/blue white/blue

title Aprendiendo Sistemas Operativos
root (hd0,0)
kernel /boot/kernel
module /boot/MODULO_1 param1 param2 .. paramN
module /boot/MODULO 2 param1 param3 .. paramN

title Otro Sistema Operativo
root (hd0,0)
kernel /boot/vmlinuz
```

En este caso para la opción "Aprendiendo Sistemas Operativos" se adicionaron dos módulos (MODULO_1 y MODULO_2) que deben ser cargados por GRUB junto con el kernel. Los módulos no necesariamente deben encontrarse en el mismo directorio que el archivo del kernel, pero se considera una buena práctica que se encuentren en el mismo directorio.

Carga de un Ramdisk

GRUB ofrece además la posibilidad de cargar una imagen de disco en memoria junto con el kernel. A esta imagen de disco se le denomina Initial Ramdisk o initrd.

Esta posibilidad es aprovechada por Linux, que usa el initrd para cargar una imagen de disco comprimida que contiene código adicional para la inicialización del sistema. Al ser una imagen de disco en memoria, el kernel no necesita aún conocer la estructura ni el formato del disco duro en el cual se encuentra el resto de su código. Solo requiere conocer la posición de memoria en la cual se cargó la imagen de disco.

Una vez que el código y los datos del Ramdisk han sido usados, el kernel puede descartarlo y liberar el espacio de memoria en el que se cargó.

Consulte la opción initrd de GRUB para más detalles.

Carga del kernel por GRUB

El código del kernel es cargado por GRUB a la dirección de memoria 0x100000 (1 MB), ya que dentro del encabezado multiboot del kernel se especificó esta como la dirección en la que se deseaba cargar el kernel (ver campo kernel_start dentro del Encabezado Multiboot, la etiqueta multiboot_header en el archivo start.S).

Debido a que GRUB es un cargador de arranque compatible con la especificación Multiboot, al pasar el control al kernel se tiene el siguiente entorno de ejecución (extractado de la Especificación Multiboot, sección 3.2 Machine state):

- La línea de direcciones A20 se encuentra habilitada, por lo cual se tiene acceso a los 32 bits de los registros del procesador y a un espacio lineal de memoria de hasta 4 GB.
- Existe una Tabla Global de Descriptores (GDT) temporal configurada por GRUB. En la documentación se insiste en que el kernel deberá crear y cargar su propia GDT tan pronto como sea posible.

- GRUB ha obtenido información del sistema, y ha configurado en la memoria una estructura de datos que recopila esta información. Incluye la cantidad de memoria disponible, los módulos del kernel cargados (ninguno en este proyecto), entre otros.
- El registro EAX contiene el valor 0x2BADB002, que indica al código del kernel que se usó un cargador compatible con la especificación Multiboot para iniciarlo.
- El registro EBX contiene la dirección física (un apuntador de 32 bits) en la cual se encuentra la estructura de datos con la información recopilada por GRUB que puede ser usada por el kernel para conocer información del sistema (memoria disponible, módulos cargados, discos y dispositivos floppy, modos gráficos, etc).
- En el registro de control CR0 el bit PG (paginación) se encuentra en 0, por lo cual la paginación se encuentra deshabilitada. El bit PE (Protection Enable) se encuentra en 1, por lo cual el procesador se encuentra operando en modo protegido de 32 bits.
- El registro de segmento de código (CS) contiene un selector que referencia un descriptor de segmento válido dentro de la GDT temporal configurada por GRUB. El segmento descrito se ha configurado como segmento de **código** en modo plano (flat), es decir que tiene base 0 y límite 4 GB.
- Los registros de segmento de datos y pila (DS, ES, FS, GS y SS) contienen un selector que referencia un descriptor de segmento válido dentro de la GDT temporal configurada por GRUB. El segmento descrito se ha configurado como un segmento de **datos** en modo plano (flat), es decir que tiene base 0 y límite 4 GB.
- El registro apuntador al tope de la pila (ESP) debe ser configurado por el kernel tan pronto como sea posible.
- El kernel debe deshabilitar las interrupciones, hasta que configure y cargue una tabla de descriptores de interrupción (IDT) válida. Dado a que el procesador ya se encuentra en modo protegido, no se puede usar la IDT que la BIOS configura al inicio del sistema.

Estructura de la Información Multiboot recopilada por GRUB

La estructura de datos que GRUB construye y cuyo apuntador se almacena en el registro EAX es definida por la Especificación Multiboot con el formato que se presenta a continuación. El desplazamiento se encuentra definido en bytes, es decir que cada campo ocupa 4 bytes (32 bits).

```
        +--------------------+
      0 | flags              | (required)| Permite identificar cuales de los
        |                    | siguientes campos se encuentran definidos:
        +--------------------+
      4 | mem_lower          | (presente si flags[0] = 1)
      8 | mem upper          | (presente si flags[0] = 1)
        +--------------------+
     12 | boot_device        | (presente si flags[1] = 1)
        +--------------------+
     16 | cmdline            | (presente si flags[2] = 1)
        +--------------------+
     20 | mods_count         | (presente si flags[3] = 1)
     24 | mods addr          | (presente si flags[3] = 1)
        +--------------------+
 28 - 40 | syms               | (presente si flags[4] or
        |                    | flags[5] = 1)
        +--------------------+
     44 | mmap_length        | (presente si flags[6] = 1)
     48 | mmap addr          | (presente si flags[6] = 1)
        +--------------------+
     52 | drives_length      | (presente si flags[7] = 1)
     56 | drives_addr        | (presente si flags[7] = 1)
        +--------------------+
     60 | config_table       | (presente si flags[8] = 1)
        +--------------------+
     64 | boot_loader_name   | (presente si flags[9] = 1)
        +--------------------+
     68 | apm_table          | (presente si flags[10] = 1)
        +--------------------+
     72 | vbe_control_info   | (presente si flags[11] = 1)
```

```
     76 | vbe_mode_info     |
     80 | vbe_mode          |
     82 | vbe_interface_seg |
     84 | vbe_interface_off |
     86 | vbe_interface_len |
        +-------------------+
```

Consulte la Especificación Multiboot para obtener más detalles acerca de esta estructura.

Ejecución del Kernel

La ejecución del kernel se divide en dos partes: ejecución del código inicial (programado en lenguaje ensamblador) y ejecución del código en C.

Ejecución del código inicial del Kernel

El código del kernel se encuentra organizado de forma que primero se ejecuta el código del archivo start.S. Esta organización del archivo ejecutable se define en el archivo link.ld.

El código de start.S define el encabezado multiboot, necesario para que el kernel sea reconocido por GRUB como compatible con la especificación multiboot. Este encabezado es leído e interpretado por GRUB al momento de cargar el kernel, de acuerdo con lo establecido en la especificación.

La ejecución inicia en la etiqueta start, en la cual se realiza un salto a la etiqueta entry_point. Se debe realizar este salto porque de no hacerlo el procesador asumiría que el encabezado multiboot (una estructura de datos insertada dentro del código) contiene instrucciones válidas y trataría de ejecutarlas.

El código que se encuentra luego de la etiqueta entry_point implementa las recomendaciones de la Especificación Multiboot:

- Deshabilitar las interrupciones
- Configurar la pila del kernel: En este caso el tope de la pila se establece en la dirección física 0x9FC00.
- Restablecer el registro EFLAGS

Luego, el código en ensamblador almacena en la pila (recién configurada) los parámetros que enviará a la función cmain (kernel.c). Estos parámetros son:

- Valor de EBX: Apuntador a la dirección de memoria en la cual GRUB ha creado la estructura de información Multiboot.
- Valor de EAX: Número mágico, que permite al código en C verificar si el kernel fué cargado por un cargador de arranque compatible con la Especificación Multiboot.

Después de insertar los parámetros en la pila, el código en ensamblador invoca la función cmain, definida en el archivo kernel.c. Esto se logra mediante el siguiente código definido en start.S:

```
/* Enviar los parámetros al kernel */
push ebx /* dirección fisica de memoria en la cual se encuentra
      la estructura de informacion multiboot. Esta puede
      ser utilizada por el kernel para obtener la informacion
      suministrada por el cargador. */
push %eax /* Número mágico del cargador de arranque = 0x2BADB002.
      En el kernel se puede validar si se recibió este valor. En caso
      afirmativo, el kernel fué cargado por un cargador que cumple
      con la especificacion multiboot. */

call cmain /* Pasar el control a la rutina 'cmain' en elarchivo kernel.c */
```

Ejecución del código en C del Kernel

Se debe recordar que una función en C recibe los parámetros de forma inversa a como se insertaron en la pila en lenguaje ensamblador. Por este motivo la definición de la función cmain es la siguiente:

```
void cmain(unsigned int magic, void * multiboot_info)
```

Dentro de la función cmain se continúa con la inicialización del kernel. En esta función se debe incluir la llamada a otras rutinas que permitirán tener un kernel, y (en un futuro) un sistema operativo funcional.

Cuando la función cmain() termina, el control se cede de nuevo al código del archivo start.S. Este código retira los parámetros almacenados en la pila, y entra en un ciclo infinito para mantener ocupado el procesador y evitar que éste se reinicie:

```
  /* La función cmain() retorna a este punto. Se debe entrar en un ciclo
  infinito, para que el procesador no siga ejecutando instrucciones al finalizar
  la ejecución del kernel. */

loop: hlt
 jmp loop /* Ciclo infinito */
```

Ver también:

http://www.gnu.org/software/grub/manual/multiboot/multiboot.html Especificación Multiboot
http://www.skyfree.org/linux/references/ELF_Format.pdf Especificación ELF
http://www.gnu.org/software/grub/ Página oficial de GRUB (Enlace externo)

Gestión de Interrupciones, Excepciones e IRQ

Autor:

Erwin Meza Vega emezav@gmail.com

Información del Proyecto : Gestión de Interrupciones, Excepciones e IRQ

En la arquitectura IA-32 se definen tres fuentes básicas de interrupción:

- Excepciones: Son condiciones de error que se presentan en la ejecución. Por ejemplo, al realizar una división por cero se lanza la excepción Division By Zero. Es una interrupción generada internamente por el procesador.
- Interrupciones de los dispositivos de hardware (discos, teclado, floppy, etc). Los dispositivos de hardware realizan solicitudes de interrupción (Interrupt Request - IRQ). Cada IRQ tiene asociado un número de interrupción predefinido, pero es posible cambiarlo por programación.
- Interrupciones por software, generadas mediante la instrucción

```
int N
```

- donde N es el número de interrupción.

La arquitectura IA-32 soporta 256 interrupciones. De estas, las 32 primeras (número 0 a 31) se asignan por defecto a las excepciones del procesador.

A continuación se muestra una descripción de las interrupciones para IA-32.

```
Número de    | Descripción
Interrupción |
(dec/hex)    |
-------------|-------------------------------------------------
  0  0x00    | Divide error:
             | Ocurre durante una instrucción DIV, cuando el divisor
             | es cero o cuando ocurre un desbordamiento del cociente.
             | Esta excepción no genera código de error.

  1  0x01    | (Reservada)
             | Esta excepción no genera código de error.

  2  0x02    | Nonmaskable interrupt:
             | Ocurre debido a una interrupcion de hardware que no se puede
             |  enmascarar.
             | Esta excepción no genera código de error.

  3  0x03    | Breakpoint:
             | Ocurre cuando el procesador encuentra una instrucción INT 3
             | Esta excepción no genera código de error.

  4  0x04    | Overflow:
             | Ocurre cuando el procesador encuentra una instrucción INTO
             | y el bit OF (Overflow) del registro EFLAGS se encuentra activo.
             | Esta excepción no genera código de error.

  5  0x05    | Bounds check (BOUND instruction):
             | Ocurre cuando el procesador, mientras ejecuta una instrucción
             | BOUND, encuentra que el operando excede el límite especificado.
             | Esta excepción no genera código de error.

  6  0x06    | Invalid opcode:
             | Ocurre cuando se detecta un código de operación inválido.
             | Esta excepción no genera código de error.

  7  0x07    | Device Not Available (No Math Coprocessor)
             | Ocurre para alguna de las dos condiciones:
             | - El procesador encuentra una instrucción ESC (Escape)
             |   y el bit EM (emulate) bit de CR0 (control register zero)
             |   se encuentra activo.
```

```
            | - El procesador encuentra una instrucción WAIT o una instrucción
            |   ESC y los bits MP (monitor coprocessor) y TS (task switched)
            |   del registro CR0 se encuentran activos.
            | Esta excepción no genera código de error.

 8  0x08    | Double fault:
            | Ocurre cuando el procesador detecta una excepción mientras trata
            | de invocar el manejador de una excepción anterior.
            | Esta excepción genera un código de error.

 9  0x09    | Coprocessor segment overrun:
            | Ocurre cuando se detecta una violación de página o segmento
            | mientras se transfiere la porción media de un operando de
            | coprocesador al NPX.
            | Esta excepción no genera código de error.

10  0xA     | Invalid TSS:
            | Ocurre si el TSS es inválido al tratar de cambiar de tarea (Task
            |  switch).
            | Esta excepción genera código de error.

11  0xB     | Segment not present:
            | Ocurre cuando el procesador detecta que el bit P (presente) de un
            | descriptor de segmento es cero.
            | Esta excepción genera código de error.

12  0xC     | Stack exception:
            | Ocurre para las siguientes condiciones:
            | - Como resultado de una violación de límite en cualquier operación
            |   que se refiere al registro de segmento de pila (SS)
            | - Cuando se trata de establecer SS con un selector cuyo descriptor
            |   asociado se encuentra marcado como no presente, pero es válido
            | Esta excepción genera código de error.

13  0xD     | General protection violation (GP):
            | Cada violación de protección que no causa otra excepción
            | causa una GP.
            | - Exceder el límite de segmento para CS, DS, ES, FS, o GS
            | - Exceder el límite de segmento cuando se referencia una tabla de
            |   descriptores
            | - Transferir el control a un segmento que no es ejecutable
            | - Escribir en un segmento de datos de sólo lectura o en un
            |   segmento de código
            | - Leer de un segmento marcado como sólo de ejecución
            | - Cargar en SS un selector que referencia a un segmento de sólo
            |   lectura
            | - Cargar SS, DS, ES, FS, o GS con un selector que referencia a un
            |   descriptor de tipo "sistema"
            | - Cargar DS, ES, FS, o GS con un selector que referencia a un
            |   descriptor de segmento marcado como ejecutable que además no se
            |   puede leer
            | - Cargar en SS un selector que referencia un descriptor  de
            |   segmento ejecutable
            | - Acceder a la memoria por medio de DS, ES, FS, o GS cuando estos
            |   registros de segmento contienen un selector nulo
            | - Pasar la ejecución (task switch) a una tarea marcada como "Busy"
            | - Violar las reglas de privilegios
            | - Cargar CR0 con los bits PG=1 y PE=0 (habilitar la paginación y
            |      no habilitar el modo protegido)
            | - Lanzar una interrupción o una excepción a través de un trap
            |   gate desde Modo Virtual 8086 a un privilegio (DPL) diferente
            |   de cero
            | Esta excepción genera código de error.

14  0xE     | Page fault:
            | Ocurre cuando la paginación está habilitada (PG = 1) en CR0
            | y el procesador detecta alguna de las siguientes condiciones
            | cuando trata de traducir una dirección lineal a física:
            | - El directorio de tabla de páginas o la tabla de páginas requerido
            |   para realizar la traducción tiene 0 en su bit de presente (P)
            | - El procedimiento actual no tiene los suficientes privilegios para
```

```
            |   acceder la página indicada.
            | Esta excepción genera código de error.

15  0xF     | (Reservada)
            | Esta excepción no genera código de error.

16  0x10    | x87 FPU Floating-Point Error (Math Fault)
            | Ocurre cuando el procesador detecta una señal del coprocesador en el
            | pin de entrada ERROR#.

17  0x11    | Alignment Check
            | Ocurre cuando se realiza una referencia de datos en la memoria a
            | una región no alineada.
            | Esta excepción genera código de error.

18  0x12    | Machine Check
            | Depende del modelo y las características del procesador.
            | Esta excepción no genera código de error.

19  0x23    | SIMD Floating-Point Exception
            | Ocurre cuando existe un error en las instrucciones SSE/SSE2/SSE3.
            | Esta excepción no genera código de error.


20  0x24    | Reservadas por Intel.
 hasta      |
31  0x1F    | Estas excepciones no generan código de error.
            |


32  0x20    | Interrupción externa o interrupción invocada mediante la
 hasta      | instrucción INT N
```
255 0xFF | Estas interrupciones no generan código de error

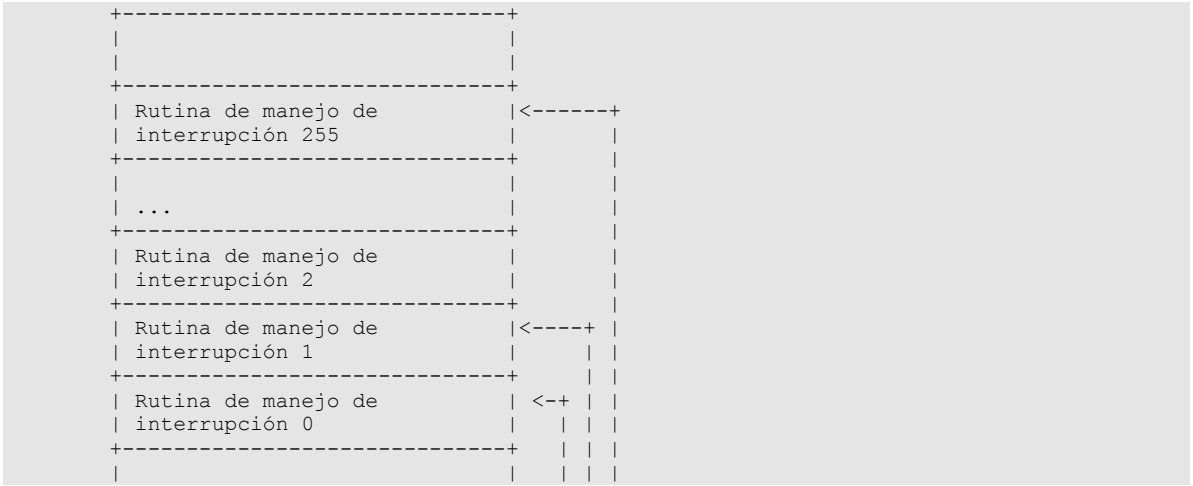
Nota:

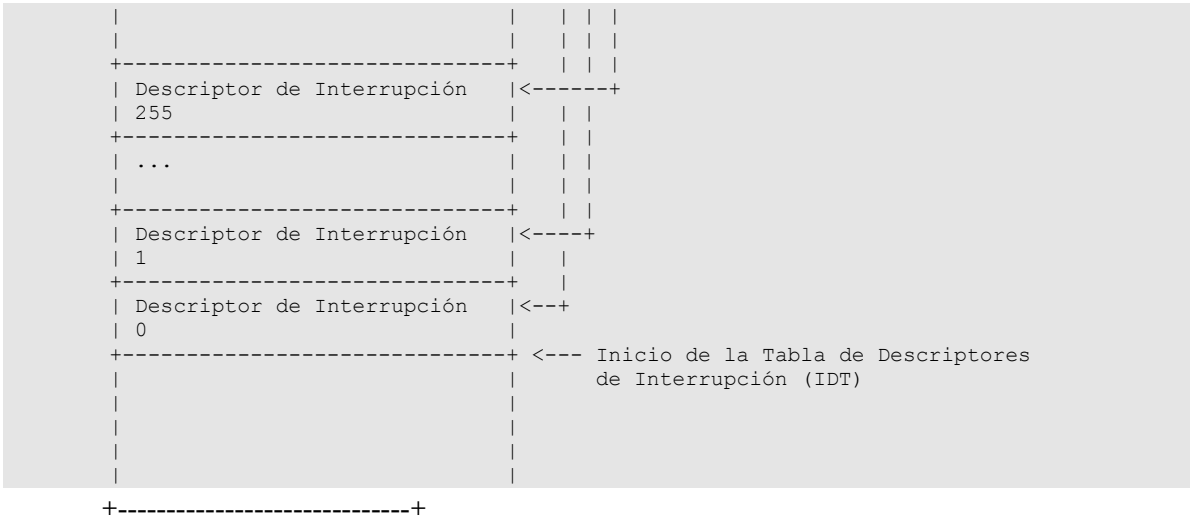
Tabla adaptada de Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1. Sección 5.3.1.

Tabla de Descriptores de Interrupción (IDT)

La IDT es una estructura de datos que usa el procesador en el momento en que ocurre la interrupción, y que debe estar configurada antes de habilitar las interrupciones. Es una tabla que contiene una serie de entradas denominadas "descriptores", que definen entre otros parámetros la dirección de memoria en la cual se encuentra cada rutina de manejo de interrupción.

El siguiente esquema muestra la IDT y las rutinas de manejo de interrupción en memoria:

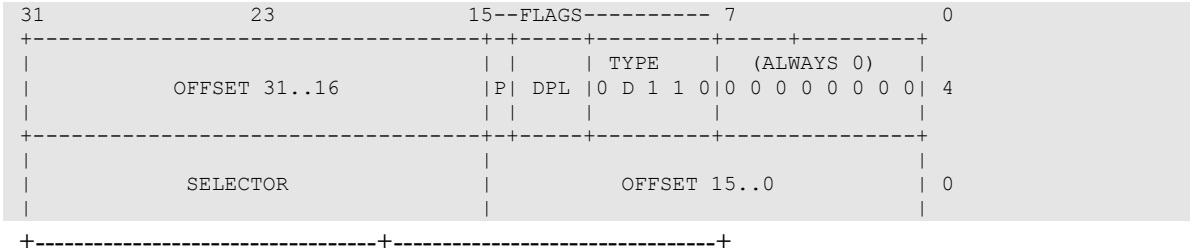




La IDT está conformada por 256 descriptores, uno para cada interrupción. Cada descriptor ocupa 8 bytes, y puede ser de uno de los siguientes tipos:

- Task-Gate
- Interrupt-Gate
- Trap-Gate

Cada entrada tiene el siguiente formato:



En donde:

- Offset: Desplazamiento (offset) en el cual se encuentra la rutina de manejo de interrupción (la dirección de memoria de la rutina) dentro de un segmento de código.
- Selector: Selector que referencia al descriptor de segmento de código en la GDT dentro del cual se encuentra la rutina de manejo de interrupción.
- D : Tipo de descriptor : (0=16 bits), (1=32 bits)
- FLAGS : compuesto por los bits P (1 bit), DPL (2 bits) y TYPE (5 bits). Para un interrupt gate, el valor de FLAGS es 0x8E = 10001110 (P = 1, DPL = 0, D = 1)

La dirección lógica segmento : offset que se obtiene del descriptor se traduce a una dirección lineal. Si la paginación se encuentra deshabilitada (por defecto), la dirección lineal es la misma dirección física en la cual se encuentra la rutina que atenderá la interrupción.

Carga de la IDT

Para cargar la IDT se utiliza la instrucción de ensamblador

```
lidt ptr_addr
```

La instrucción idt toma el puntero y lo carga en el registro IDTR del procesador. ptr_addr corresponde a la dirección de memoria en la cual se encuentra una estructura de datos que describe la IDT. Esta estructura de datos se denomina 'puntero a la IDT', 'IDT Pointer'.

El puntero al IDT tiene el siguiente formato:

```
 47                  15              0
 +-----------------------------------+
 |      base         |    límite     |
 +-------------------------------+ en donde: base = dirección lineal de la IDT, que corresponde a la dirección de memoria de IDT.
```

límite = tamaño de la IDT en Bytes. Si la IDT tiene 256 entradas y el tamaño de cada entrada es de 8 bytes, el tamaño total de la IDT es de 2048 bytes (2 KB).

Gestión de Interrupciones en IA-32

Cuando ocurre una interrupción (0 a 255), el procesador busca en la IDT la rutina que la atenderá. La función setup_idt() ha configurado la IDT para que se ejecute la rutina isrN cuando ocurre la interrupción N. Estas rutinas se definen en ensamblador en el archivo isr.S.

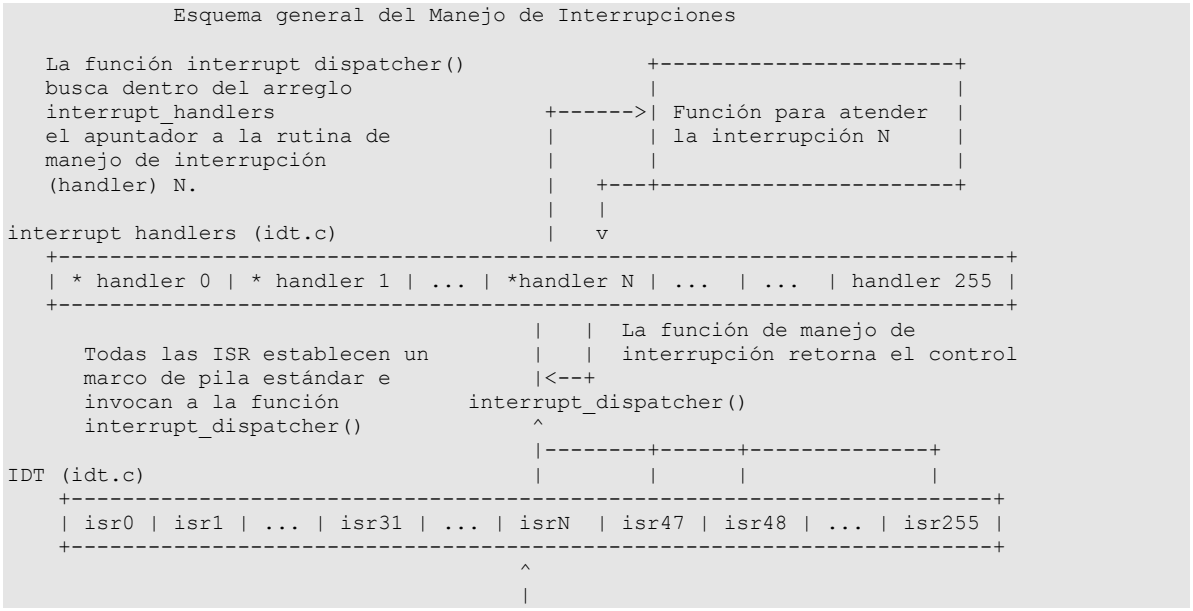
Las rutinas isr0 ... isr255 almacenan el estado actual del procesador, y también el número de la interrupción que ocurrió. Estas rutinas guardan el apuntador al tope de la pila en el cual se encuentra el estado del procesador en las variables globales current_ss y current_esp, y luego invocan a la función interrupt_dispatcher() implementada en idt.c.

La rutina interrupt_dispatcher toma el marco de pila y extrae el número de la interrupción que ocurrió. Con este valor busca dentro de la tabla de manejadores de interrupción (la variable interrupt_handlers) el apuntador a la rutina que manejará la interrupción correspondiente.

Si encuentra un apuntador válido, invoca la rutina de manejo de interrupción. Cuando esta rutina termina, retorna de nuevo el control a interrupt_dispatcher().

A su vez, interrupt_dispatcher() retorna a la función isrN que la invocó.

Luego, la función isrN invoca a la función return_from_interrupt() (isr.S), la cual recupera el estado del procesador y retorna de la interrupción al sitio en el cual la ejecución fue interrumpida.



Interrupción N -+

Instalar y Desinstalar Manejadores de Interrupción

Para instalar un nuevo manejador de interrupción, se debe invocar a la función install_interrupt_handler() definida en el archivo idt.c. Esta función recibe como parámetro el número de interrupción, y el apuntador a la rutina que manejará la interrupción. Este apuntador se almacena en la posición correspondiente en el arreglo (la tabla) interrupt_handlers.

Para desinstalar un manejador de interrupción, se debe especificar el número de interrupción. Esta función elimina el apuntador correspondiente en el arreglo (la tabla) interrupt_handlers.

Gestión de Excepciones

Para gestionar las excepciones se agrega una capa de abstracción adicional. Luego de configurar las 255 rutinas manejadoras de interrupción (setup_idt()) el código del kernel en cmain() invoca a la función setup_exceptions().

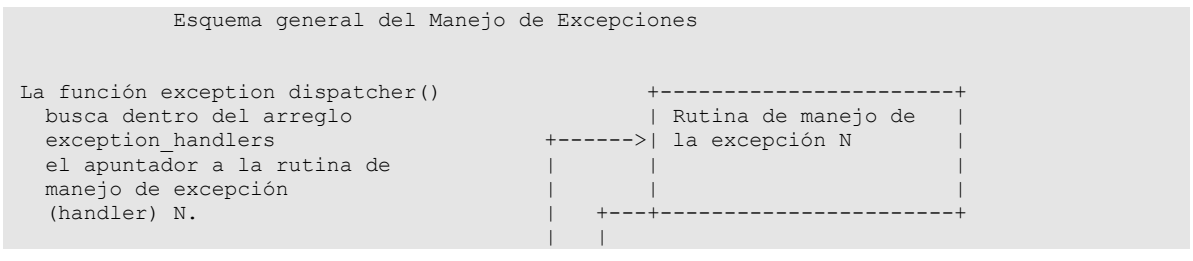
Esta función configura las entradas 0 a 31 de la tabla interrupt_handlers con apuntadores a la función exception_dispatcher(). Esta función permite centralizar la gestión de excepciones.

De forma similar al arreglo interrupt_handlers de idt.c que permite almacenar los apuntadores a las funciones que manejarán las 255 interrupciones IA-32, en el archivo exception.c se define el arreglo (la tabla) exception_handlers que almacena los apuntadores a las funciones que manejarán las 32 excepciones IA-32.

Proceso de gestión de una excepción

A continuación se describe el proceso para gestionar una excepción.

1. Cuando ocurre una excepción N (interrupción 0 a 31), se ejecuta la rutina isrN (0 <= N <= 31) correspondiente (configurada en setup_idt()).
2. La rutina isrN crea el marco de pila en el cual almacena el estado del procesador, guarda el apuntador actual a la pila (current_ss y current_esp) e invoca a la función interrupt_dispatcher().
3. La función interrupt_dispatcher() busca dentro de la tabla interrupt_handlers el apuntador a la rutina de manejo de interrupción. Debido a que la función setup_exceptions() ha configurado las entradas 0 a 31 para que apunten a la función exception_dispatcher() (exception.c), se invoca esta función.
4. La función exception_dispatcher() busca dentro de la tabla exception_handlers el apuntador a la función que manejará la excepción correspondiente (que se configura con la función install_exception_handler() en el archivo exception.c). Si existe un manejador de excepción, se invoca. En caso contrario, imprime un error y entra en un ciclo infinito.
5. Cuando la rutina de manejo de excepción termina, retorna el control a la función exception_dispatcher(). Esta a su vez retorna el control a interrupt_dispatcher(), la cual retorna a la función isrN definida en isr.S.
6. Luego de recibir el control de interrupt_dispatcher(), la rutina isrN invoca a la función return_from_interrupt() definida en isr.S. Esta función recupera el estado del procesador y retorna al sitio en el cual el procesador fue interrumpido.



Instalar y Desinstalar Manejadores de Excepción

Para instalar un nuevo manejador de excepción, se debe invocar a la función install_exception_handler(() definida en el archivo exception.c. Esta función recibe como parámetro el número de excepción, y el apuntador a la rutina que manejará la excepción. Este apuntador se almacena en la posición correspondiente en el arreglo (la tabla) exception_handlers.

Para desinstalar un manejador de excepción, se debe especificar el número de excepción. Esta función elimina el apuntador correspondiente en el arreglo (la tabla) exception_handlers.

Gestión de Solicitudes de Interrupción - IRQ

Cuando un dispositivo de Entrada / Salida requiere atención, lanza una Solicitud de Interrupción (Interrupt Request - IRQ). Estas IRQ son recibidas por un dispositivo llamado el PIC PIC (Programmable Interrupt Controller). El trabajo del PIC consiste en recibir y priorizar las IRQ recibidas, y enviar una señal de interrupción a la CPU.

En la arquitectura IA-32 el sistema cuenta con dos controladores PIC, uno llamado "Maestro" y otro "Esclavo", que se encuentra conectado en cascada al PIC Maestro. Cada PIC puede atender 8 líneas de IRQ, por lo tanto se pueden atender hasta 16 solicitudes.

Al arranque del sistema, las líneas de interrupción IRQ0 a IRQ 5 se encuentran mapeadas a las interrupciones numero 0x8 a 0xF. Las líneas de interrupción IRQ8 a IRQ 15 se encuentran mapeadas a las interrupciones 0x70 a 0x77.

Lista de Solicitudes de Interrupción

La lista de IRQ es la siguiente:

```
              Número de
   IRQ        Interrupción   Descripción
   ----      -------------   ---------------------------------------
   IRQ0         0x08         Timer
   IRQ1         0x09         Teclado
   IRQ2         0x0A         Cascade para el PIC esclavo
   IRQ3         0x0B         Puerto serial 2
   IRQ4         0x0C         Puerto serial 1
   IRQ5         0x0D         AT: Puerto paralelo2  PS/2 : reservado
   IRQ6         0x0E         Diskette
   IRQ7         0x0F         Puerto paralelo 1
   IRQ8/IRQ0    0x70         Reloj de tiempo real del CMOS
   IRQ9/IRQ1    0x71         Refresco vertical de CGA
   IRQ10/IRQ2   0x72         Reservado
   IRQ11/IRQ3   0x73         Reservado
   IRQ12/IRQ4   0x74         AT: reservado. PS/2: disp. auxiliar
   IRQ13/IRQ5   0x75         FPU (Unidad de Punto Flotante)
   IRQ14/IRQ6   0x76         Controlador de disco duro
   IRQ15/IRQ7   0x77         Reservado
```

Al observar la tabla anterior, se hace evidente que existe un problema: las interrupciones 0x8 a 0x0F también son utilizadas para las excepciones de la arquitectura IA-32, ya que éstas siempre ocupan las interrupciones 0-31.

Por esta razón, es necesario reprogramar el PIC para que las interrupciones de entrada/salida se mapeen después de las excepciones de IA-32, es decir desde la interrupción numero 32 en adelante. A la IRQ 0 (Timer) le corresponderá la interrupción número 32, y asi sucesivamente.

A continuación se presenta el proceso, que se implementa en la función irq_remap() del archivo irq.c. Esta función es invocada por setup_irq() en el momento de configurar la IRQ.

```
  /* Initialization Command Word 1 - ICW1
Esta es la palabra primaria para inicializar el PIC.
Para inicializar el PIC se requiere que los bits 0 y 4 de ICW1 esten en
1 y los demas en 0. Esto significa que el valor de ICW1 es 0x11.
ICW1 debe ser escrita en el  registro de comandos del PIC maestro
(dirección de e/s 0x20). Si existe un PIC esclavo, ICW1 se debe enviar
tambien su registro de comandos del PIC esclavo (0xA0)
*/

outb(MASTER_PIC_COMMAND_PORT, 0x11);
outb(SLAVE_PIC_COMMAND_PORT, 0x11);

/* Initialization Command Word 2 - ICW2
Esta palabra permite definir la dirección base (inicial) en la tabla de
descriptores de interrupcion que el PIC va a utilizar.

Debido a que las primeras 32 entradas estan reservadas para las
excepciones en la arquitectura IA-32, ICW2 debe contener un valor mayor o
igual a 32 (0x20). Los valores  de ICW2 representan el numero de IRQ
base que manejara el PIC

Al utilizar los PIC en cascada, se debe enviar ICW2 a los dos
controladores en su registro de datos (0x21 y 0xA1 para maestro y
esclavo respectivamente), indicando la dirección en la IDT que va a ser
utilizada por cada uno de ellos.

Las primeras 8 IRQ van a ser manejadas por el PIC maestro y se mapearan
a partir del numero 32 (0x20). Las siguientes 8 interrupciones las manejara
el PIC esclavo, y se mapearan a partir de la interrupcion 40 (0x28).
*/

outb(MASTER_PIC_DATA_PORT, IDT_IRQ_OFFSET);
outb(SLAVE_PIC_DATA_PORT, IDT_IRQ_OFFSET + 8);
```

```
/* Initialization Control Word 3 - ICW3
Esta palabra permite definir cuales lineas de IRQ van a ser compartidas
por los PIC maestro y esclavo. Al igual que ICW2, ICW3 tambien se
escribe en los  registros de datos de los PIC (0x21 y 0xA1 para el PIC
maestro y esclavo,respectivamente).

Dado que en la arquitectura Intel el PIC maestro se conecta con el PIC
esclavo por medio de la linea IRQ 2, el valor de ICW3 debe ser 00000100
(0x04), que define el bit 3 (correspondiente a la linea IRQ2) en 1.

Para el PIC esclavo, el numero de la linea se debe representar en
notacion binaria. Por lo tanto, 000 corresponde a la linea de IRQ 0,
001 corresponde a la linea de IRQ 1, 010 corresponde a la linea de
IRQ 2, y asi sucesivamente.
Debido a que se va a utilizar la linea de IRQ 2, el valor de ICW3
para el PIC esclavo debe ser 00000010, (0x02).
*/

outb(MASTER_PIC_DATA_PORT, 0x04);
outb(SLAVE PIC DATA PORT, 0x02);

/* Initialization Control Word 4 - ICW4
Para ICW4 solo es necesario establecer su bit 0 (x86 mode) y escribirla
en los registros de datos del PIC maestro y esclavo (0x21 y 0xA1).
El valor de ICW4 debe ser entonces 00000001, es decir, 0x01.
*/
outb(MASTER PIC DATA PORT, 0x01);
outb(SLAVE_PIC_DATA_PORT, 0x01);

/* Se han mapeado las IRQ!.
 * Las IRQ 0-7 seran atendidas por el PIC maestro, y las IRQ 8-15
 * por el PIC esclavo. Las IRQ0-15 estaran mapeadas en la IDT a partir
 * de la entrada 32 hasta la 47.*/
```

Rutina para la gestión de las IRQ

Luego de re-configurar el PIC para que las 16 IRQ generen las interrupciones 32 a 47, la función setup_irq() configura estas entradas en la tabla interrupt_handlers para que apunten a la función irq_dispatcher() definida en el archivo irq.c. Esta rutina, similar a exception_dispatcher() permite centralizar el manejo de las IRQ.

Instalar y Desinstalar Manejadores de IRQ

Para instalar un nuevo manejador de IRQ, se debe invocar a la función install_irq_handler() definida en el archivo irq.c. Esta función recibe como parámetro el número de la IRQ (0 a 15), y el apuntador a la rutina que manejará la IRQ. Este apuntador se almacena en la posición correspondiente en el arreglo (la tabla) irq_handlers.

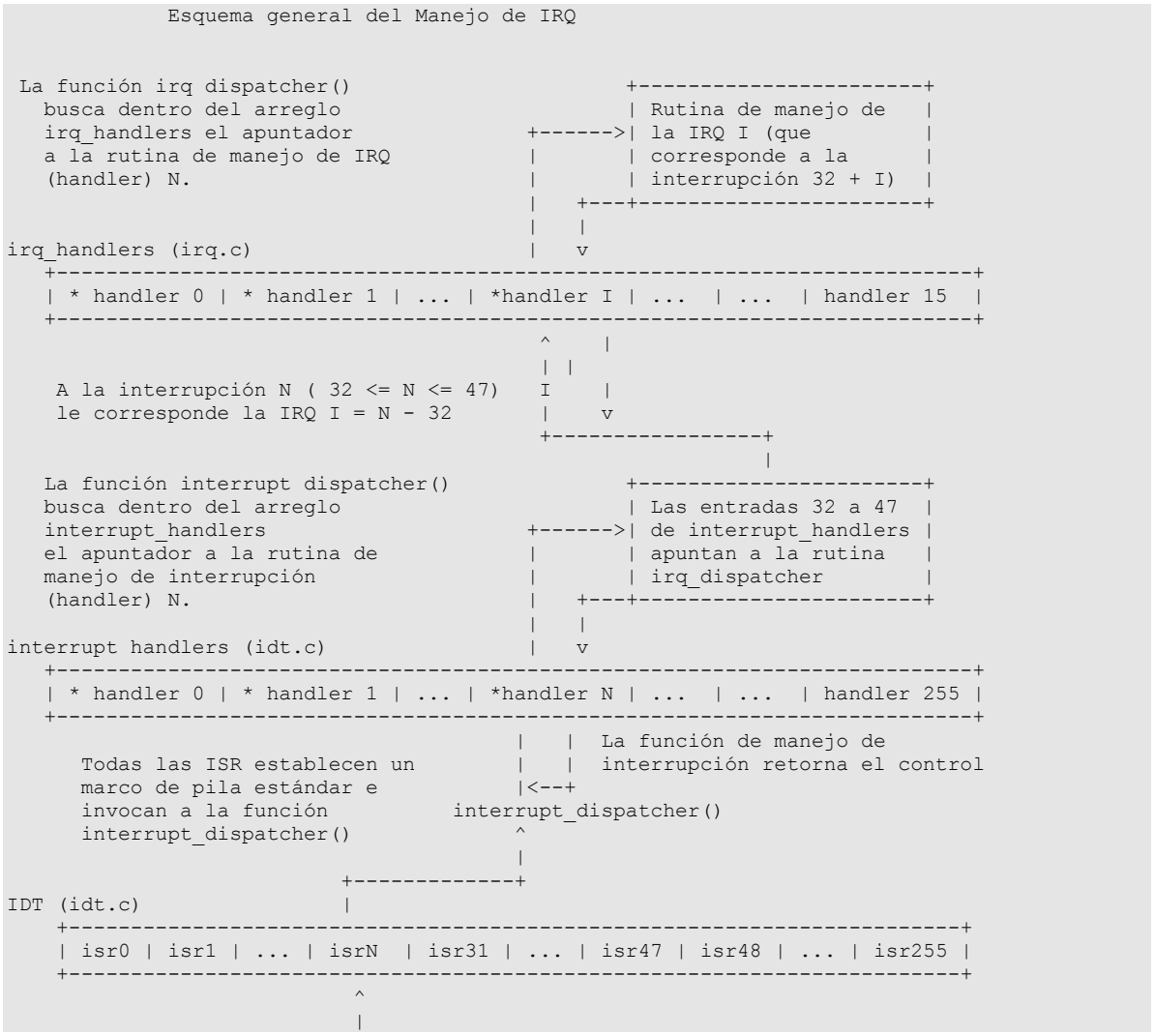
Para desinstalar un manejador de IRQ, se debe especificar el número de IRQ. Esta función elimina el apuntador correspondiente en el arreglo (la tabla) irq_handlers.

Proceso de gestión de una IRQ

A continuación se describe el proceso para gestionar una excepción.

1. Cuando ocurre una IRQ N (interrupción 32 a 47), se ejecuta la rutina isrN (32 <= N <= 74) correspondiente (configurada en setup_idt()).
2. La rutina isrN crea el marco de pila en el cual almacena el estado del procesador, guarda el apuntador actual a la pila (current_ss y current_esp) e invoca a la función interrupt_dispatcher().

3. La función interrupt_dispatcher() busca dentro de la tabla interrupt_handlers el apuntador a la rutina de manejo de interrupción. Debido a que la función setup_irq() ha configurado las entradas 32 a 47 para que apunten a la función irq_dispatcher() (irq.c), se invoca esta función.
4. La función irq_dispatcher() busca dentro de la tabla irq_handlers el apuntador a la función que manejará la IRQ correspondiente (que se configura con la función install_irq_handler() en el archivo irq.c). Si existe un manejador de IRQ, se invoca. En caso contrario, la función irq_dispatcher() retorna.
5. Cuando la rutina de manejo de IRQ termina, retorna el control a la función irq_dispatcher(). Esta a su vez retorna el control a interrupt_dispatcher(), la cual retorna a la función isrN definida en isr.S.
6. Luego de recibir el control de interrupt_dispatcher(), la rutina isrN invoca a la función return_from_interrupt() definida en isr.S. Esta función recupera el estado del procesador y retorna al sitio en el cual el procesador fue interrumpido.



Ver también:

http://www.gnu.org/software/grub/manual/multiboot/multiboot.html Especificación Multiboot
http://www.skyfree.org/linux/references/ELF_Format.pdf Especificación ELF
http://www.gnu.org/software/grub/ Página oficial de GRUB (Enlace externo)

Gestión de Memoria Física con un Mapa de Bits

Autor:

Erwin Meza Vega emezav@gmail.com

Información del Proyecto : Gestión de Memoria Física con un Mapa de Bits

La gestión de memoria es el mecanismo de asignar y liberar unidades de memoria de forma dinámica. Para ofrecer esta funcionalidad, es necesario contar con una estructura de datos que permita llevar un registro de la memoria que se encuentra asignada y la memoria libre.

Los mapas de bits son un mecanismo para gestionar memoria que se basan en un principio simple: Usando un bit (cuyo valor puede ser cero o uno) se puede determinar si un byte o una región de memoria se encuentra disponible o no. Esto ofrece una posibilidad sencilla para gestionar memoria, pero se puede ver limitada por el tamaño del mapa de bits en sí.

Por ejemplo, si se desea gestionar una memoria de 4 GB (2^32 bytes) y se usa un bit por cada byte de memoria (tomando la unidad básica de asignación como un byte), el mapa de bits ocuparía 2^29 bits, es decir 512 MB.

Una estrategia para solucionar este inconveniente consiste en usar unidades de asignación mayores a un byte. Por ejemplo, si se crea un mapa de bits en el cual cada bit representa una región de memoria (unidad de asignación) de 4 KB (2^12 bytes), el mapa de bits correspondiente para una memoria de 4 GB ´ ocuparía exactamente 128 KB. Este tamaño es aceptable, pero causa que no se puedan asignar unidades de memoria menores a 4 KB.

A continuación se presenta una descripción gráfica del uso de un mapa de bits.

```
                Esquema del Mapa de Bits

   +-----------------------------------+        Cada bit en el mapa de bits
   | 1| 0| 1| 0| 1| 0|..|..|..|..| 0| 1|        representa una unidad de
   +-----------------------------------+        asignación de memoria

+-------------------------------------------------------------------------+
|libre |usada|libre|usada|libre|usada|...  |     |     |     |usada|libre |
|      |     |     |     |     |     |     |     |     |     |     |      |
+-------------------------------------------------------------------------+
```

Creación del mapa de bits

El mapa de bits inicialmente se llena de ceros, para indicar todo el espacio de memoria como no disponible. Luego a partir de la información de la memoria disponible se "habilitan" las entradas correspondientes.

Asignación de Memoria

La asignación de memoria se puede realizar de dos formas:

- Asignar una unidad de memoria: Se recorre el mapa de bits buscando un bit que se encuentre en 1 (región disponible). Si se encuentra este bit, se obtiene el inicio de la dirección de memoria que éste representa y se retorna.

```
                  Asignar una unidad de memoria

     +------------- Esta entrada (bit) en el mapa de bits se encuentra en 1.
     |              Esto significa que la región asociada a este bit está
     v              disponible.
   +-----------------------------------+        Cada bit en el mapa de bits
   | 1| 0| 1| 0| 1| 0|..|..|..|..| 0| 1|        representa una unidad de
   +-----------------------------------+        asignación de memoria
```

```
+-------------------------------------------------------------------------+
|libre |usada|libre|usada|libre|usada|...  |     |     |     |usada|libre |
|      |     |     |     |     |     |     |     |     |     |     |      |
+-------------------------------------------------------------------------+
  ^
  |
  +---------- Región de memoria representada por el primer bit. Se debe
              retornar la dirección de memoria de inicio de la región.


     +------------- La entrada se marca como "no disponible"
     |
     v
   +-----------------------------------+      Mapa de bits actualizado
   | 0| 0| 1| 0| 1| 0|..|..|..|..| 0| 1|
   +-----------------------------------+
```

- Asignar una región de memoria de N bytes: Primero se redondea el tamaño solicitado a un múltiplo del tamaño de una unidad de asignación. Luego se busca dentro del mapa de bits un número consecutivo de bits que sumen la cantidad de memoria solicitada. Si se encuentra, se marcan todos los bits como no disponibles y Se retorna la dirección física que le corresponde al primer bit en el mapa de bits.

```
                  Asignar una región de memoria

          +-------------  Este es el inicio de la región de memoria
          |               disponible
          v
   +-----------------------------------+        Cada bit en el mapa de bits
   | 1| 0| 1| 1| 1| 1|..|..|..|..| 0| 1|        representa una unidad de
   +-----------------------------------+        asignación de memoria

+-------------------------------------------------------------------------+
|libre |usada|libre|libre|libre|libre|...  |     |     |     |usada|libre |
|      |     |     |     |     |     |     |     |     |     |     |      |
+-------------------------------------------------------------------------+
              ^
              |
              +---------- Inicio de la región de memoria. Se retorna la
                          dirección que le corresponde al primer bit del mapa.


          +-------------  La región de memoria se marca como no disponible
          |
          v
   +-----------------------------------+        Se deben marcar los bits
   | 1| 0| 0| 0| 0| 0|..|..|..|..| 0| 1|        correspondientes como
   +-----------------------------------+        "no disponible"
```

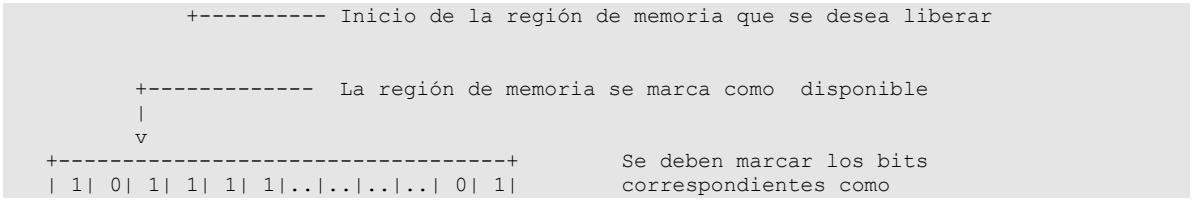
Liberado de Memoria

Para liberar memoria se puede simplemente establecer en 1 (disponible) el bit correspondiente a la unidad o la región a liberar.

```
                  Liberar una región de memoria

          +-------------  Este es el inicio de la región de memoria
          |               asignada
          v
   +-----------------------------------+        Cada bit en el mapa de bits
   | 1| 0| 0| 0| 0| 0|..|..|..|..| 0| 1|        representa una unidad de
   +-----------------------------------+        asignación de memoria

+-------------------------------------------------------------------------+
|libre |usada|usada|usada|usada|usada|...  |     |     |     |usada|libre |
|      |     |     |     |     |     |     |     |     |     |     |      |
+-------------------------------------------------------------------------+
              ^
              |
```



Ver también:

http://www.gnu.org/software/grub/manual/multiboot/multiboot.html Especificación Multiboot
http://www.skyfree.org/linux/references/ELF_Format.pdf Especificación ELF
http://www.gnu.org/software/grub/ Página oficial de GRUB (Enlace externo)

Creación de Imágenes de Disco con GRUB preinstalado

Autor:

Erwin Meza Vega emezav@gmail.com

Información del Proyecto : Creación de Imágenes de Disco con GRUB preinstalado

A continuación se presenta el proceso para crear una imagen de disco e instalar GRUB en su sector de arranque.

Requisitos del sistema

Se debe cumplir con los siguientes requisitos:

1. Utilidades para gestión de discos (fdisk, mke2fs)
2. Grub
3. Utilidad dd
4. Utilidad e2fsimage

Estos requisitos son cumplidos por la mayoría de instalaciones de Linux. La utilidad e2fsimage no se encuentra instalada por defecto, así que deberá ser instalada. En Ubuntu se puede usar la herramienta apt-get o el gestor de paquetes synaptic para instalarla. En otros sistemas operativos Linux, se deberá instalar primero las dependencias de e2fsimage (generalmente se encuentran en un paquete llamado e2fsprogs), y luego descargar e instalar desde el código fuente e2fsimage.

En Windows es posible cumplir estos requisitos en un entorno Cygwin o MinGW/Msys.

Breve Descripción de la Geometría C/H/S de (la imagen de) un Disco Duro

La ubicación de cualquier sector dentro de un disco se describe mediante tres parámetros:

- C : Cilindro (pista) en el cual se encuentra el sector.
- H : Cabeza, que permite determinar el "plato" en el cual se encuentra la pista.
- S : Permite ubicar un sector dentro de una pista.

Para más detalles acerca de C/H/S, consulte la documentación disponible en la siguiente página y sus páginas relacionadas:

http://www.storagereview.com/guide/tracksDifference.html

En un disco duro se supone que todas las pistas contienen el mismo número de sectores. Si bien esto no es correcto desde el punto de vista físico (ya que las pistas son concéntricas) la lógica del dispositivo realiza los ajustes correspondientes para que el software pueda referenciar la misma cantidad de sectores dentro de todas las pistas.

La fórmula para calcular la capacidad aproximada de un disco viene dada por sus valores máximos de C, H y S. El cálculo se ilustra a continuación:

```
                                                    Capacidad
             =     Número de sectores del disco     * de un sector
                                                    (512 bytes)

Capacidad        Número de pistas del     Sectores por
del          =   disco                *   pista          * 512
disco            ____________________
en bytes         Número de   Número de    Sectores por   * 512
             =    cilindros   Cabezas      Pista
```

= C * H * S * 512

Donde C, H y S son el número de cilindros, cabezas y sectores por pista que tiene el disco.

Si se fijan los parámetros H (cabezas) y S (sectores por pista), el tamaño del disco dependerá del número de cilindros. Así tomando H = 16 y S = 63, el número de cilindros para un disco con capacidad de **X** MB se puede calcular mediante la siguiente fórmula:

```
            Capacidad del disco en bytes
     C =   ------------------------------
                  H * S * 512
```

De esta forma, para un disco con capacidad de 5 MB, se tiene:

```
                 5242880   (5 MB)
     C =   ----------------------  =  10
               16 * 63 * 512
```

Se debe tener en cuenta que la capacidad del disco no es exactamente 5 MB, en realidad es un poco menos. Esto se debe a que no se pueden tener valores decimales para C, H y S.

La siguiente tabla ilustra este aspecto:

C	H	S	Bytes por Sector	Capacidad en bytes	Capacidad en KB	Capacidad en MB	Número de Sectores
10	16	63	512	5160960	5040	4,92	10080
20	16	63	512	10321920	10080	9,84	20160
30	16	63	512	15482880	15120	14,77	30240
100	16	63	512	51609600	50400	49,22	100800

Creación del archivo de la imagen de disco

Conociendo los valores de C, H y S y el número de sectores que determinan la capacidad X deseada, es posible crear una imagen de disco mediante el comando dd:

```
   dd if=/dev/zero of=disk_template bs=512 count=10080
```

El comando dd copia a nivel de bloques de un archivo de entrada a un archivo de salida. Los parámetros de este comando son:

- if Archivo de entrada (input file). El archivo /dev/zero es un archivo especial que permite generar tantos '0' como sean necesarios.
- of Archivo de salida (output file). El archivo resultante se llamará disk_template
- bs Tamaño de un bloque. Por defecto 512 bytes. Si la imagen es muy grande, el tamaño de bloque se puede especificar en MB (bs=1MiB).
- count: Número de bloques a copiar. 10080 bloques de 512 bytes, aprox. 4.9 MB.

Creación de particiones dentro de la imagen de disco

Para crear las particiones dentro de la imagen de disco, se usa el comando fdisk. Se debe pasar como parámetro el número de cilindros, sectores y cabezas de acuerdo con la geometría del disco.

```
fdisk -C 10 -H 16 -S 63 disk_template
```

La primera vez que se ejecuta el comando, fdisk advierte que la imagen de disco no contiene una tabla de particiones válida. Esto es cierto, debido a que el archivo disk_image sólo contiene ceros. El mensaje es el siguiente:

```
El dispositivo no contiene una tabla de particiones DOS válida ni una etiqueta de disco Sun o
SGI o OSF
Se está creando una nueva etiqueta de disco DOS con el identificador 0x221c4d50.
Los cambios sólo permanecerán en la memoria, hasta que decida escribirlos.
Tras esa operación, el contenido anterior no se podrá recuperar.

Atención: el indicador 0x0000 inválido de la tabla de particiones 4 se corregirá mediante w(rite)

AVISO: El modo de compatibilidad DOS es obsoleto. Se recomienda fuertemente
         apagar el modo (orden «c») y cambiar mostrar unidades a
         sectores (orden «u»).

Orden (m para obtener ayuda):
```

Al escribir el comando **w** , fdisk corrige este error y termina su ejecución.

```
El dispositivo no contiene una tabla de particiones DOS válida ni una etiqueta de disco Sun o
SGI o OSF
Se está creando una nueva etiqueta de disco DOS con el identificador 0x221c4d50.
Los cambios sólo permanecerán en la memoria, hasta que decida escribirlos.
Tras esa operación, el contenido anterior no se podrá recuperar.

Atención: el indicador 0x0000 inválido de la tabla de particiones 4 se corregirá mediante w(rite)

AVISO: El modo de compatibilidad DOS es obsoleto. Se recomienda fuertemente
         apagar el modo (orden «c») y cambiar mostrar unidades a
         sectores (orden «u»).

Orden (m para obtener ayuda): w
¡Se ha modificado la tabla de particiones!

Se están sincronizando los discos.
```

Se debe iniciar de nuevo fdisk:

```
fdisk -C 10 -H 16 -S 63 disk_template
```

Esta vez no se muestra el error anterior.

Se puede invocar el comando **m** para obtener ayuda de las opciones de fdisk:

```
Orden (m para obtener ayuda): m
Orden  Acción
   a   Conmuta el indicador de iniciable
   b   Modifica la etiqueta de disco bsd
   c   Conmuta el indicador de compatibilidad con DOS
   d   Suprime una partición
   l   Lista los tipos de particiones conocidos
   m   Imprime este menú
   n   Añade una nueva partición
   o   Crea una nueva tabla de particiones DOS vacía
   p   Imprime la tabla de particiones
   q   Sale sin guardar los cambios
   s   Crea una nueva etiqueta de disco Sun
   t   Cambia el identificador de sistema de una partición
```

```
   u   Cambia las unidades de visualización/entrada
   v   Verifica la tabla de particiones
   w   Escribe la tabla en el disco y sale
   x   Funciones adicionales (sólo para usuarios avanzados)

Orden (m para obtener ayuda):
```

Los comandos a usar son:

- n Añade una nueva partición. Se debe elegir partición de tipo primaria, número de partición 1 (es la primera partición), primer cilindro = 1, último cilindro = 10 (la partición ocupa todo el disco).

```
Orden (m para obtener ayuda): n
Acción de la orden
   e   Partición extendida
   p   Partición primaria (1-4)
p
Número de partición (1-4): 1
Primer cilindro (1-10, valor predeterminado 1): 1
Último cilindro, +cilindros o +tamaño{K,M,G} (1-10, valor predeterminado 10): 10
```

- Orden (m para obtener ayuda):
- p Permite listar la tabla de particiones. Deberá aparecer la partición creada en el paso anterior.

```
Orden (m para obtener ayuda): p

Disco disk_template: 0 MB, 0 bytes
16 cabezas, 63 sectores/pista, 10 cilindros
Unidades = cilindros de 1008 * 512 = 516096 bytes
Tamaño de sector (lógico / físico): 512 bytes / 512 bytes
Tamaño E/S (mínimo/óptimo): 512 bytes / 512 bytes
Identificador de disco: 0x221c4d50

   Dispositivo Inicio    Comienzo      Fin      Bloques  Id  Sistema
disk_template1               1          10        5008+  83  Linux
```

- Orden (m para obtener ayuda):
- **Nota:**

  Es posible que (como en la gráfica) fdisk muestre 0 MB como el tamaño del disco. Esto es normal, debido a que no se está usando un disco real sino una imagen de disco.

- a Permite activar una partición (definirla como la partición de arranque del disco.

```
Orden (m para obtener ayuda): a
Número de partición (1-4): 1

Orden (m para obtener ayuda): p

Disco disk template: 0 MB, 0 bytes
16 cabezas, 63 sectores/pista, 10 cilindros
Unidades = cilindros de 1008 * 512 = 516096 bytes
Tamaño de sector (lógico / físico): 512 bytes / 512 bytes
Tamaño E/S (mínimo/óptimo): 512 bytes / 512 bytes
Identificador de disco: 0x221c4d50

   Dispositivo Inicio    Comienzo      Fin      Bloques  Id  Sistema
disk_template1   *           1          10        5008+  83  Linux
```

- Orden (m para obtener ayuda):
- w Guarda los cambios realizados en la imagen de disco y cierra fdisk.

```
Orden (m para obtener ayuda): w
¡Se ha modificado la tabla de particiones!

Se están sincronizando los discos.
```

Ahora se debe verificar que la partición quedó creada y activa dentro de la imagen de disco. Además es importante saber exactamente el sector a partir del cual comienza la partición definida. Esto se logra mediante el comando:

```
fdisk -C 10 -H 16 -S 63 -u -l disk_template
```

El resultado es el siguiente:

```
Disco disk_template: 0 MB, 0 bytes
16 cabezas, 63 sectores/pista, 10 cilindros, 0 sectores en total
Unidades = sectores de 1 * 512 = 512 bytes
Tamaño de sector (lógico / físico): 512 bytes / 512 bytes
Tamaño E/S (mínimo/óptimo): 512 bytes / 512 bytes
Identificador de disco: 0x221c4d50

   Dispositivo Inicio    Comienzo      Fin      Bloques  Id  Sistema
disk_template1   *          63       10079        5008+  83  Linux
```

Se puede ver que la partición comienza en el sector 63 del disco.

Formateo de la Partición en el Disco

Para dar formato a la partición dentro del disco, se debe hacer uso de la utilidad mke2fs. Si bien es posible "montar" la partición que se encuentra dentro de la imagen de disco, es más recomendable extraer esta partición, formatearla de forma independiente y luego llevarla de nuevo dentro de la imagen de disco. A continuación se describe el proceso para extraer la partición de la imagen de disco, formatearla e insertarla de nuevo dentro de la imagen de disco.

1. Extraer la partición de la imagen de disco. El comando

   ```
   fdisk -C 10 -H 16 -S 63 -u -l disk_template
   ```

   permite encontra el sector desde el cual comienza la primera partición:

   ```
      Dispositivo Inicio    Comienzo      Fin      Bloques  Id  Sistema
   disk_template1   *          63       10079        5008+  83  Linux
   ```

   Para extraer la partición de la imagen de disco, se puede usar de nuevo el comando **dd:**

   ```
   dd if=disk_template of=first_partition bs=512 skip=63
   ```

   El parámetro skip=63 le especifica a **dd** que debe descartar los primeros 63 sectores (sector 0 al 62) de la entrada. Esto permite extraer únicamente la primera partición. No es necesario especificar el número de bloques a sacar, en este caso solo existe una partición. El resultado de este comando es el siguiente:

   ```
   10177+0 registros de entrada
   10177+0 registros de salida
   5210624 bytes (5,2 MB) copiados, 0,0709376 s, 73,5 MB/s
   ```

2. Formatear la partición extraida. Esto se logra mediante el comando:

   ```
   mke2fs first_partition
   ```

   La respuesta de mke2fs es la siguiente:

   ```
   mke2fs 1.41.14 (22-Dec-2010)
   first_partition no es un dispositivo especial de bloques.
   ¿Continuar de todas formas? (s,n) s
   Etiqueta del sistema de ficheros=
   Tipo de SO: Linux
   Tamaño del bloque=1024 (bitácora=0)
   Tamaño del fragmento=1024 (bitácora=0)
   Stride=0 blocks, Stripe width=0 blocks
   1272 nodos-i, 5088 bloques
   254 bloques (4.99%) reservados para el superusuario
   Primer bloque de datos=1
   Número máximo de bloques del sistema de ficheros=5242880
   1 bloque de grupo
   8192 bloques por grupo, 8192 fragmentos por grupo
   1272 nodos-i por grupo

   Escribiendo las tablas de nodos-i: hecho
   Escribiendo superbloques y la información contable del sistema de ficheros: hecho

   Este sistema de ficheros se revisará automáticamente cada 27 montajes o
   180 días, lo que suceda primero.  Utilice tune2fs -c o -i para cambiarlo.
   ```

Copia de archivos dentro de la partición formateada

Ya se tiene el archivo first_partition, que contiene una partición formateada. Ahora dentro de este archivo se deberá crear la estructura de directorios que contiene los archivos de GRUB, para su instalación.

Este proceso se puede realizar usando el comando **mount** para montar la partición, luego copiar los archivos y finalmente usar el comando **umount** para desmontar la partición.

Otra estrategia consiste en utilizar el programa e2fsimage, que permite agregar archivos y directorios directamente a un archivo de partición sin necesidad de montarlo o desmontarlo. La única restricción consiste en que no se pueden sobreescribir archivos o directorios que ya existan dentro de la partición.

El proceso de usar e2fsimage se describe a continuación:

1. Crear una estructura de directorios que contenga los archivos de GRUB y los archivos de configuración. Para el ejemplo se creará un directorio llamado filesys, el cual se reproduce la siguiente estructura:

```
filesys
|
 boot
     |
     |
     grub  <-- Almacena los archivos de GRUB
     |   |
     |   |
     | e2fs_stage_1_5
     |   |
     | stage1
     |   |
```

| stage2 Los archivos e2fs_stage_1_5, stage1 y stage2 se pueden copiar del directorio /boot/grub. La copia de estos archivos se realiza con los siguientes pasos:

```
mkdir filesys
mkdir -p filesys/boot/grub
cp /boot/grub/stage1 filesys/boot/grub
cp /boot/grub/e2fs stage1 5 filesys/boot/grub
cp /boot/grub/stage2 filesys/boot/grub
```

2. Actualizar el contenido de la primera partición de la imagen de disco a partir del directorio creado en el paso anterior. Para ello se usa el comando e2fsimage:

```
e2fsimage -f first_partition -d filesys -n
```

El resultado del comando es el siguiente:

```
e2fsimage - Version: 0.2.2
Copied 2 Directorys, 3 regular files, 0 symlinks
0 hard links and 0 special files - total 5
```

El archivo de partición ahora contiene los archivos requeridos para instalar GRUB. Ahora se deberá tomar de nuevo el archivo first_partition (la partición con los archivos, e insertarlo de nuevo dentro de la imagen de disco. Esto se logra de nuevo con el comando **dd:**

```
dd if=first_partition of=disk_image bs=512 seek=63
```

Este comando permite tomar el archivo de entrada (la partición actualizada) e insertarlo en el archivo de salida (la imagen de disco), omitiendo 63 bloques del archivo de salida (seek=63). Con esto se inserta la partición dentro de la imagen del disco saltando los sectores 0 a 62.

El resultado de este comando es el siguiente:

```
10177+0 registros de entrada
10177+0 registros de salida
5210624 bytes (5,2 MB) copiados, 0,0718208 s, 72,6 MB/s
```

Instalar GRUB dentro de la imagen de disco

Ahora se debe instalar GRUB dentro de la imagen de disco, que ya cuenta con los archivos requeridos dentro de su primera partición. Esto se logra mediante la siguiente secuencia:

1. Ejecutar GRUB para instalarlo en el archivo disk_image (la imagen de disco)

```
grub
```

Se inicia el proceso de instalación de GRUB:

```
    GNU GRUB  version 0.97  (640K lower / 3072K upper memory)

 [ Minimal BASH-like line editing is supported.  For the first word, TAB
   lists possible command completions.  Anywhere else TAB lists the possible
   completions of a device/filename. ]

grub>
```

2. Realizar la secuencia de instalación de GRUB, especificando la partición en la cual se encuentra el kernel (la primera partición = 0) y el archivo del kernel.

```
grub> device (hd0) disk_template

grub> geometry (hd0) 10 16 63
drive 0x80: C/H/S = 10/16/63, The number of sectors = 10080, disk_template
   Partition num: 0,  Filesystem type is ext2fs, partition type 0x83

grub> root (hd0,0)
 Filesystem type is ext2fs, partition type 0x83

grub> setup (hd0)
 Checking if "/boot/grub/stage1" exists... yes
 Checking if "/boot/grub/stage2" exists... yes
 Checking if "/boot/grub/e2fs_stage1_5" exists... yes
 Running "embed /boot/grub/e2fs stage1 5 (hd0)"...  17 sectors are embedded.
succeeded
 Running "install /boot/grub/stage1 (hd0) (hd0)1+17 p (hd0,0)/boot/grub/stage2
/boot/grub/menu.lst"... succeeded
Done.

grub>
```

Los comandos de esta secuencia de instalación son los siguientes:

- device (hd0) disk_template : Especifica a GRUB que el dispositivo (disco) sobre el cual será instalado es el archivo disk_template.
- geometry (hd0) 10 16 63 : Especifica la geometría de (la imagen de) disco del dispositivo.
- root (hd0, 0) : Define la partición raíz en la cual se encuentra el kernel que GRUB debe cargar se encuentra en la primera partición del dispositivo (las particiones se numeran desde cero, así que es la primera partición)
- setup (hd0) : Instalar GRUB en el sector maestro de (la imagen de) disco. También es posible instalar GRUB en el sector de arranque de la partición, mediante el comando setup (hd0,0).

Con esta secuencia GRUB se encuentra instalado dentro de la imagen de disco.

Debido a que la plantilla de imagen de disco sólo contiene a GRUB, si se intenta arrancar presentará un prompt y no cargará ningún kernel (ya que no existe).

Para arrancar un kernel, se deberán incluir dos archivos más en la imagen de disco:

- El archivo boot/grub/menu.lst, que contiene la configuración de GRUB y las opciones de arranque.
- Un kernel.

Por ejemplo, se puede crear el siguiente archivo menu.lst:

```
default 0
timeout 10
color cyan/blue white/blue
```

```
title Aprendiendo Sistemas Operativos
root (hd0,0)
```
kernel /boot/kernel

En este caso se está definiendo que el kernel a cargar es un archivo llamado kernel dentro del directorio boot. Se deberán insertar los dos archivos dentro de la imagen de disco (boot/grub/menu.lst y boot/kernel).

Ver también:

http://www.storagereview.com/guide/index_geom.html Geometría de Disco Duro
http://sourceforge.net/projects/e2fsimage/ Página de la utilidad e2fsimage
http://www.gnu.org/software/grub/manual/multiboot/multiboot.html Especificación Multiboot
http://www.skyfree.org/linux/references/ELF_Format.pdf Especificación ELF
http://www.gnu.org/software/grub/ Página oficial de GRUB (Enlace externo)

Imagen de Disco usada en el Proyecto

Autor:

Erwin Meza Vega emezav@gmail.com

Información del Proyecto : Imagen de Disco usada en el Proyecto

En este proyecto se usa imagen de disco duro, en la cual se encuentra pre-instalado GRUB. Esta imagen de disco se actualiza dinámicamente para incluir el archivo del kernel compilado.

Copia del Kernel dentro de la Imagen de Disco

Cuando se ejecuta alguno de los Make Targets definidos dentro del proyecto (all, bochs, bochsdbg o qemu), automáticamente se dispara la regla **all** . La regla **all** tiene como pre-requisito el kernel compilado, así que dispara las reglas que permiten compilar el código del kernel y obtener el archivo en formato ELF llamado kernel.

Dentro de la regla **all** se realiza el siguiente proceso:

1. Verificar si existe el archivo disk_template.gz (plantilla de la imagen de disco, comprimida). Si existe, se descomprime el archivo .gz para obtener la plantillallamada disk_template.

```
   -if test -f disk_template.gz; then \
    gunzip disk_template.gz; \
```

   else true; fi
2. El archivo disk_template es una imagen de disco duro, que ocupa aproximadamente 5 MB. Dentro de este archivo ya se encuentra instalado GRUB La imagen de disco tiene una geometría de C/H/S de 10/16/63, con una capacidad aproximada de 4.9 MB.
3. Copiar el archivo del kernel recién generado dentro del directorio filesys/boot. El directorio filesys contiene la estructura de directorios que se copiará dentro de la única partición de la imagen de disco:

```
/ -+    <-- Directorio raíz de la única partición en la imagen de disco
   |
   boot <-- Almacena los archivos de GRUB y el kernel compilado
       |
       |
       grub  <-- Almacena los archivos de GRUB
       |   |
       |   |
       | e2fs_stage_1_5  <-- Etapa 1.5 de GRUB. cargado por la etapa 1 de GRUB
       |   |                 Contiene el código para manejar el sistema de
       |   |                 archivos de la partición (ext2). Este archivo
       |   |                 es opcional, ya que al instalar GRUB este archivo
       |   |                 se copió en el sector adyacente al sector de
       |   |                 arranque.
       |   |
       | menu.lst    <-- Archivo de configuración leido por GRUB al arranque.
       |   |             especifica la configuración del menú que despliega
       |   |             GRUB al arranque y ubicación del kernel en
       |   |             (la imagen de) disco.
       | stage1      <-- Etapa 1 de GRUB. Este archivo es opcional, ya que se
       |   |             copió en el sector de arranque del disco al instalar
       |   |             GRUB.
       |   |             Carga la etapa 1.5 de GRUB. Después carga la
       |   |             etapa 2 de GRUB desde el disco y le pasa el control.
       |   |             Este archivo es opcional.
       |   |
       | stage2      <-- Etapa 2 de GRUB. Cargada por la etapa 1 de GRUB.
       |                 Configura el sistema y presenta el menú que
       |                 permite cargar el kernel.
       |                 Este archivo es obligatorio.
       |                 Cuando el usuario selecciona la única opción
       |                 disponible: cargar y pasar el control el archivo kernel
       |                 que se encuentra en el directorio /boot de la imagen
       |                 de disco
```

```
        |                 El kernel se carga a la dirección de memoria 0x100000
        |                 (1 MB)
        |
        kernel   <-- Archivo que contiene el código compilado del kernel.
```

Este proceso se realiza con el comando:

```
   #Copiar el kernel al directorio filesys/kernel
```

cp -f kernel filesys/boot/kernel

4. Crear una copia de la plantilla de la imagen de disco disk_template a un archivo llamado disk_image. Esto se realiza para tener siempre una plantilla vacía en el archivo disk_template. El archivo disk_image será la imagen de disco cargada por los emuladores bochs y grub.

```
    #Copiar la plantilla a la imagen de disco
```

cp -f disk_template disk_image

5. Extraer la primera partición de la imagen de disco. La primera partición empieza en el sector 63 de la imagen de disco.

```
 #Extraer la particion inicial
```

dd if=disk_template of=first_partition bs=512 skip=63

6. Actualizar el contenido de la primera partición (en el archivo first_partition) con el contenido del directorio filesys. Esto se realiza con la utilidad e2fsimage.

```
 #Copiar el kernel al sistema de archivos
```

e2fsimage -f first_partition -d filesys -n

7. Luego se debe copiar la partición actualizada de nuevo a la imagen de disco, y borrar el archivo first_partition.

```
 #Copiar la particion actualizada a la imagen de disco
 dd if=first_partition of=disk_image bs=512 seek=63
 #Borrar la particion actualizada
```

rm -f first_partition

Con este proceso la imagen de disco se actualiza automáticamente cada vez que se compila el kernel.

A continuación se reproduce la regla 'all' definida en el archivo Makefile.

```
all: kernel
 -if test -f disk_template.gz; then \
    gunzip disk template.gz; \
    else true; fi
 #Copiar el kernel al directorio filesys/kernel
 cp -f kernel filesys/boot/kernel
 #Copiar la plantilla a la imagen de disco
 cp -f disk_template disk_image
 #Extraer la particion inicial
 dd if=disk_template of=first_partition bs=512 skip=63
 #Copiar el kernel al sistema de archivos
 e2fsimage -f first_partition -d filesys -n
 #Copiar la particion actualizada a la imagen de disco
 dd if=first partition of=disk image bs=512 seek=63
 #Borrar la particion actualizada
 rm -f first_partition
```

Ver también:

Creación de Imágenes de Disco con GRUB preinstalado
http://www.gnu.org/software/grub/manual/multiboot/multiboot.html Especificación Multiboot
http://www.skyfree.org/linux/references/ELF_Format.pdf Especificación ELF
http://www.gnu.org/software/grub/ Página oficial de GRUB (Enlace externo)

Memoria de Video en Modo Texto

Autor:

Erwin Meza Vega emezav@gmail.com

Información del Proyecto : Memoria de Video en Modo Texto

La memoria de video en modo texto a color (25 líneas de 80 caracteres cada una) se encuentra mapeada en memoria a la dirección 0xB8000, y ocupa 32 KB de memoria. Cada carácter en que se muestra en la pantalla ocupa 2 bytes en la memoria de video (un word): un byte contiene el código ASCII del carácter, y el otro byte contiene los atributos de color de texto y color de fondo del carácter. A su vez este byte se subdivide en:

```
 7  6  5  4  3  2  1  0
+-----------------------+
|I |B |B |B |I |F |F |F |
+-----------------------+
Los bits F correspondel al color del texto (Foreground).
Los bits B corresponden al color de fondo (Background).
El bit I corresponde a la intensidad del color de fondo (0 = oscuro,
```

1 = claro) o del color del texto.

De esta forma, para mostrar un carácter en la esquina superior de la pantalla (línea 0, carácter 0) se deberá escribir un word (2 bytes) en la dirección de memoria 0xB8000. El primer byte de este word será el código ascii a mostrar, y los siguientes bytes representarán el color de texto y de fondo del carácter.

El siguiente word (ubicado en la dirección de memoria 0xB8002) corresponde al segundo carácter en la pantalla, y así sucesivamente.

Los colores válidos se muestran en la siguiente tabla :

```
Valor  Color   Valor   Color
0      black   8       dark gray
1      blue    9       bright blue
2      green   10      bright green
3      cyan    11      bright cyan
4      red     12      pink
5      magenta 13      bright magenta
6      brown   14      yellow
```

7 white 15 bright white

Los colores 0-15 son válidos para el color de texto. Sin embargo para el fondo solo es posible utilizar los colores del 0 al 7,. Los colores de fondo 8 al 15 en algunas tarjetas causan el efecto 'blink' (parpadeo) del texto.

Por ejemplo, para imprimir el carácter 'A', (al cual le corresponde el código ASCII 65, 0x41 en hexadecimal) con color blanco sobre fondo negro en la esquina superior de la pantalla, se deberá copiar el word 0x0F41 en la dirección de memoria 0xB8000. El byte de atributos 0x0F indica texto blanco (dígito hexa F) sobre fondo negro (dígito hexa 0).

Se puede observar la posición XY de un carácter en la pantalla se puede obtener de la siguiente forma:

Pos_XY = 0xB8000 + ((80*Y) + X) *2

En donde 0xB8000 es la dirección base de la memoria de video (esquina superior izquierda), Y representa la fila, X representa la columna. Se debe multiplicar por 2 debido a que cada carácter en

la pantalla en realidad ocupa 2 bytes, uno para el código ascii y otro para los atributos de color de texto y fondo.

Actualización de la Posición del Cursor

El controlador CRT (controlador de video) cuenta con los siguientes puertos de Entrada y Salida:

```
Dirección   Descripción
  0x3D4    registro de índice
```

0x3D5 registro de control

Para programar el microcontrolador CRT, primero se debe escribir en el registro de índice. El valor escrito en este puerto le indica al controlador el parámetro que se desea configurar. Luego se debe escribir en el registro de control el dato requerido.

En el caso de la configuración de la posición del cursor, se debe escribir en los puertos del CRT la posición lineal Pos_XY. Debido a que esta posición requiere 16 bits y a que los puertos de E/S del CRT son de 8 bits (1 byte), se deben escribir primero los 8 bits menos significativos de la posición lineal y luego los 8 bits más significativos. Este proceso se describe a continuaión:

1. Escribir el valor 0x0F (Cursor Location Low) en el registro de índice del CRT (puerto de E/S 0x3D4).
2. Escribir los 8 bits menos significativos de la posición lineal del cursor al registro de control del CRT (puerto 0x3D5).
3. Escribir el valor 0x0E (Cursor Location High) en el registro de índice del CRT (puerto de E/S 0x3D4).
4. Escribir los 8 bits más significativos de la posición lineal del cursor al registro de control del CRT (puerto 0x3D5).

Ver también:

http://www.osdever.net/FreeVGA/vga/crtcreg.htm Puertos (registros) del controlador CRT

Indice de módulos

Módulos

Lista de todos los módulos:

Índice de clases

Lista de clases

Lista de las clases, estructuras, uniones e interfaces con una breve descripción:

Indice de archivos

Lista de archivos

Lista de todos los archivos con descripciones breves:

Documentación de módulos

Código del Kernel

Archivos

- archivo asm.h
- *Rutinas para la ejecución de código ensamblador para la arquitectura IA-32.* archivo exception.h
- *En este archivo se definen los tipos y las funciones relacionados con la gestion de excepciones en la arquitectura IA-32.* archivo idt.h
- *Contiene las definiciones globales requeridas para el manejo de interrupciones en la arquitectura IA-32.* archivo irq.h
- *Este archivo define las rutinas publicas para la gestion de solicitudes de interrupcion (IRQ) de los dispositivos de Entrada / Salida.* archivo multiboot.h
- *Contiene algunas constantes necesarias para el kernel relacionadas con la especificación Multiboot.* archivo physmem.h
- *Contiene las definiciones relacionadas con las gestión de memoria del kernel.* archivo pm.h
- *Contiene las definiciones relacionadas con el Modo Protegido IA-32.* archivo stdio.h
- *Contiene las primitivas basicas para entrada / salida.* archivo stdlib.h
- *Contiene las definiciones de algunas funciones de utilidad.* archivo exception.c
- *Este archivo implementa las primitivas necesarias para el manejo de excepciones en la arquitectura IA-32.* archivo idt.c
- *Este archivo implementa las primitivas para el manejo de interrupciones en la arquitectura IA-32.* archivo irq.c
- *Contiene la implementacion de las rutinas necesarias para manejar las solicitudes de interrupcion (IRQ) de los dispositivos de entrada / salida.* archivo isr.S
- *Contiene la definicion y la implementacion de las rutinas de servicio de interrupcion para las 255 interrupciones que se pueden generar en un procesador IA-32. Todas las rutinas establecen un marco de pila uniforme, y luego invocan a la rutina interrupt_dispatcher().* archivo kernel.c
- *Código de inicialización del kernel en C. Este código recibe el control de start.S y continúa con la ejecución.* archivo physmem.c
- *Contiene la implementación de las rutinas relacionadas con las gestión de memoria física. La memoria se gestiona en unidades de 4096 bytes.* archivo pm.c
- *Contiene la implementación de las rutinas para la gestión del modo protegido IA-32 y de la Tabla Global de Descriptores (GDT)* archivo start.S
- *Punto de entrada del kernel.* archivo stdio.c
- *Contiene las primitivas basicas para entrada / salida.* archivo stdlib.c

Contiene las implementaciones de algunas funciones de utilidad.

Descripción detallada

Conjunto de archivos usados por el kernel

Documentación de las clases

Referencia de la Estructura aout_symbol_table

Tabla de símbolos usadas en el formato a.out.
`#include <multiboot.h>`

Atributos públicos

- unsigned int tabsize
- unsigned int strsize
- unsigned int addr
- unsigned int reserved

Descripción detallada

Tabla de símbolos usadas en el formato a.out.

Definición en la línea 67 del archivo multiboot.h.

Documentación de los datos miembro

unsigned int addr

Definición en la línea 71 del archivo multiboot.h.

unsigned int reserved

Definición en la línea 72 del archivo multiboot.h.

unsigned int strsize

Definición en la línea 70 del archivo multiboot.h.

unsigned int tabsize

Definición en la línea 69 del archivo multiboot.h.

La documentación para esta estructura fue generada a partir del siguiente fichero:

- include/multiboot.h

Referencia de la Estructura elf_section_header_table

Tabla de encabezados de sección del kernel en el formato elf.
`#include <multiboot.h>`

Atributos públicos

- unsigned int num
- unsigned int size
- unsigned int addr
- unsigned int shndx

Descripción detallada

Tabla de encabezados de sección del kernel en el formato elf.

Definición en la línea 76 del archivo multiboot.h.

Documentación de los datos miembro

unsigned int addr

Definición en la línea 80 del archivo multiboot.h.

unsigned int num

Definición en la línea 78 del archivo multiboot.h.

unsigned int shndx

Definición en la línea 81 del archivo multiboot.h.

unsigned int size

Definición en la línea 79 del archivo multiboot.h.

La documentación para esta estructura fue generada a partir del siguiente fichero:

- include/multiboot.h

Referencia de la Estructura gdt_descriptor

Estructura de datos para un descriptor de segmento.
`#include <pm.h>`

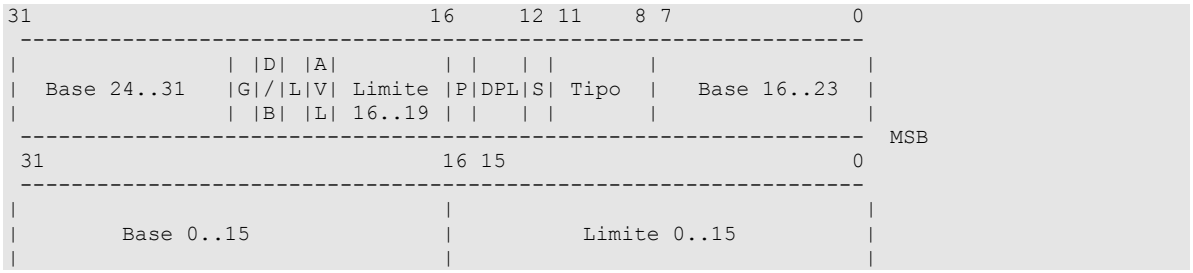
Atributos públicos

- unsigned int low: 32
  Bits menos significativos del descriptor. Agrupan Limite 0..15 y Base 0..15 del descriptor.
- unsigned int high: 32

Descripción detallada

Estructura de datos para un descriptor de segmento.

De acuerdo con el manual de Intel, un descriptor de segmento en modo protegido de 32 bits es una estructura de datos que ocupa 8 bytes (64 bits), con el siguiente formato (Ver Cap. 3, Seccion 3.4.5, en Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1)



La distribución de los bits dentro del descriptor se explica a continuación:

- Los bits Base 0..31 permiten definir la base del segmento en el espacio lineal de 32 bits.
- Los bits Limite 0..19 permiten definir el tamaño del segmento. Este límite también se relaciona con el bit G (Granularidad) de la siguiente forma:
  - Si G es 0, el tamaño del segmento en bytes es igual al valor almacenado en Limite
  - Si G = 1, el tamaño del segmento es el valor almacenado en Limite multiplicado por 4096.
- El bit D/B funciona diferente para para segmentos de codigo y datos. Para modo protegido de 32 bits este bit debe tener valor de 1. Consulte el manual de Intel para más detalles.
- El bit L debe ser 0 para modo protegido de 32 bits.
- El bit AVL puede ser tener como valor 1 o 0. Por defecto se toma 0.
- El bit P es 1 si el segmento está presente, 0 en caso contrario.
- El bit DPL define el nivel de privilegios del descriptor. Por tener 2 bits puede almacenar tres valores: 0, 1 y 2. 0 es el mayor privilegio.
- El bit S para descriptores de segmento de codigo o datos debe ser 1.
- Los bits correspondientes a Tipo definen el tipo de segmento. Vea la sección 3.5.1 del manual de Intel mencionado.
  - Para segmentos de código, Tipo tiene el valor binario de 1010 = 0xA
  - Para segmentos de datos, Tipo tiene el valor binario de 0010 = 0x2Estructura de datos para un descriptor de segmento
-

Definición en la línea 92 del archivo pm.h.

Documentación de los datos miembro

unsigned int high

bits más significativos del descriptor. Agrupan Base 16..23, Tipo, S, DPL, P, Límite, AVL, L, D/B, G y Base 24..31

Definición en la línea 98 del archivo pm.h.

unsigned int low

Bits menos significativos del descriptor. Agrupan Limite 0..15 y Base 0..15 del descriptor.

Definición en la línea 95 del archivo pm.h.

La documentación para esta estructura fue generada a partir del siguiente fichero:

- include/pm.h

Referencia de la Estructura gdt_pointer_t

Estructura de datos para el registro GDTR (puntero a la GDT)
`#include <pm.h>`

Atributos públicos

- unsigned short limit
- unsigned int base

Descripción detallada

Estructura de datos para el registro GDTR (puntero a la GDT)

Definición en la línea 105 del archivo pm.h.

Documentación de los datos miembro

unsigned int base

Definición en la línea 107 del archivo pm.h.

unsigned short limit

Definición en la línea 106 del archivo pm.h.

La documentación para esta estructura fue generada a partir del siguiente fichero:

- include/pm.h

Referencia de la Estructura idt_descriptor

Definición de la estructura de datos para un descriptor de interrupción.
`#include <idt.h>`

Atributos públicos

- unsigned short offset_low: 16
- unsigned short selector: 16
- unsigned short type: 16
- unsigned int offset_high: 16

Descripción detallada

Definición de la estructura de datos para un descriptor de interrupción.

Definición en la línea 26 del archivo idt.h.

Documentación de los datos miembro

unsigned int offset_high

Bits más significativos del desplazamiento dentro del segmento de código en el cual se encuentra la rutina de manejo de interrupción

Definición en la línea 37 del archivo idt.h.

unsigned short offset_low

Bits menos significativos del desplazamiento dentro del segmento de código en el cual se encuentra la rutina de manejo de interrupción

Definición en la línea 29 del archivo idt.h.

unsigned short selector

Selector del segmento de código en el cual se encuentra la rutina de manejo de interrupción

Definición en la línea 32 del archivo idt.h.

unsigned short type

Tipo del descriptor

Definición en la línea 34 del archivo idt.h.

La documentación para esta estructura fue generada a partir del siguiente fichero:

- include/idt.h

Referencia de la Estructura idt_pointer_t

Estructura de datos para el registro IDTR (puntero a la IDT)
`#include <idt.h>`

Atributos públicos

- unsigned short limit
  Tamaño de la IDT.
- unsigned int base
  dirección lineal del inicio de la IDT

Descripción detallada

Estructura de datos para el registro IDTR (puntero a la IDT)

Definición en la línea 44 del archivo idt.h.

Documentación de los datos miembro

unsigned int base

dirección lineal del inicio de la IDT

Definición en la línea 48 del archivo idt.h.

unsigned short limit

Tamaño de la IDT.

Definición en la línea 46 del archivo idt.h.

La documentación para esta estructura fue generada a partir del siguiente fichero:

- include/idt.h

Referencia de la Estructura interrupt_state

Estructura que define el estado del procesador al recibir una interrupción o una excepción.
`#include <idt.h>`

Atributos públicos

- unsigned int gs
  Valor del selector GS (Tope de la pila)
- unsigned int fs
  Valor del selector FS.
- unsigned int es
  Valor del selector ES.
- unsigned int ds
  Valor del selector DS.
- unsigned int edi
  Valor del registro EDI.
- unsigned int esi
  Valor del registro ESI.
- unsigned int ebp
  Valor del registro EBP.
- unsigned int esp
  Valor del registro ESP.
- unsigned int ebx
  Valor del registro EBX.
- unsigned int edx
  Valor del registro EDX.
- unsigned int ecx
  Valor del registro ECX.
- unsigned int eax
  Valor del registro EAX.
- unsigned int number
  Número de la interrupción (o excepción)
- unsigned int error_code
  Código de error. Cero para las excepciones que no generan código de error y para las interrupciones.
- unsigned int old_eip
  Valor de EIP en el momento en que ocurrió la interrupción (almacenado automáticamente por el procesador)
- unsigned int old_cs
  Valor de CS en el momento en que ocurrió la interrupción (almacenado automáticamente por el procesador)
- unsigned int old_eflags
  Valor de EFLAGS en el momento en que ocurrió la interrupción (almacenado automáticamente por el procesador)
- unsigned int old_esp

Valor de ESP en el momento en que ocurrió la interrupción (almacenado automáticamente por el procesador). Sólo se almacena cuando la interrupción o excepción ocurrió cuado una tarea de privilegio mayor a cero se estaba ejecutando.

- unsigned int old_ss
  Valor de SS en el momento en que ocurrió la interrupción (almacenado automáticamente por el procesador). Sólo se almacena cuando la interrupción o excepción ocurrió cuado una tarea de privilegio mayor a cero se estaba ejecutando.

Descripción detallada

Estructura que define el estado del procesador al recibir una interrupción o una excepción.

Al recibir una interrupción, el procesador automáticamente almacena en la pila el valor de CS, EIP y EFLAGS. Si la interrupción ocurrió en un nivel de privilegios diferente de cero, antes de almacenar CS, EIP y EFLAGS se almacena el valor de SS y ESP. El control lo recibe el código del archivo isr.S, en el cual almacena (en orden inverso) el estado del procesador contenido en esta estructura.

Definición en la línea 70 del archivo idt.h.

Documentación de los datos miembro

unsigned int ds

Valor del selector DS.

Definición en la línea 78 del archivo idt.h.

unsigned int eax

Valor del registro EAX.

Definición en la línea 94 del archivo idt.h.

unsigned int ebp

Valor del registro EBP.

Definición en la línea 84 del archivo idt.h.

unsigned int ebx

Valor del registro EBX.

Definición en la línea 88 del archivo idt.h.

unsigned int ecx

Valor del registro ECX.

Definición en la línea 92 del archivo idt.h.

unsigned int edi

Valor del registro EDI.
Definición en la línea 80 del archivo idt.h.

unsigned int edx

Valor del registro EDX.
Definición en la línea 90 del archivo idt.h.

unsigned int error_code

Código de error. Cero para las excepciones que no generan código de error y para las interrupciones.
Definición en la línea 99 del archivo idt.h.

unsigned int es

Valor del selector ES.
Definición en la línea 76 del archivo idt.h.

unsigned int esi

Valor del registro ESI.
Definición en la línea 82 del archivo idt.h.

unsigned int esp

Valor del registro ESP.
Definición en la línea 86 del archivo idt.h.

unsigned int fs

Valor del selector FS.
Definición en la línea 74 del archivo idt.h.

unsigned int gs

Valor del selector GS (Tope de la pila)
Definición en la línea 72 del archivo idt.h.

unsigned int number

Número de la interrupción (o excepción)

Definición en la línea 96 del archivo idt.h.

unsigned int old_cs

Valor de CS en el momento en que ocurrió la interrupción (almacenado automáticamente por el procesador)

Definición en la línea 105 del archivo idt.h.

unsigned int old_eflags

Valor de EFLAGS en el momento en que ocurrió la interrupción (almacenado automáticamente por el procesador)

Definición en la línea 108 del archivo idt.h.

unsigned int old_eip

Valor de EIP en el momento en que ocurrió la interrupción (almacenado automáticamente por el procesador)

Definición en la línea 102 del archivo idt.h.

unsigned int old_esp

Valor de ESP en el momento en que ocurrió la interrupción (almacenado automáticamente por el procesador). Sólo se almacena cuando la interrupción o excepción ocurrió cuado una tarea de privilegio mayor a cero se estaba ejecutando.

Definición en la línea 113 del archivo idt.h.

unsigned int old_ss

Valor de SS en el momento en que ocurrió la interrupción (almacenado automáticamente por el procesador). Sólo se almacena cuando la interrupción o excepción ocurrió cuado una tarea de privilegio mayor a cero se estaba ejecutando.

Definición en la línea 118 del archivo idt.h.

La documentación para esta estructura fue generada a partir del siguiente fichero:

- include/idt.h

Referencia de la Estructura memory_map

Estructura de datos que almacena la información de una región de memoria dentro del mapa de memoria proporcionado por GRUB.

`#include <multiboot.h>`

Atributos públicos

- unsigned int entry_size
  Campo no usado.
- unsigned int base_addr_low
  32 bits menos significativos de la base de la región de memoria
- unsigned int base_addr_high
  32 bits más significativos de la base de la región de memoria
- unsigned int length_low
  32 bits menos significativos del tamaño de la región de memoria
- unsigned int length_high
  32 bits más significativos del tamaño de la región de memoria
- unsigned int type
  Tipo de área de memoria 1 = disponible, 2 = reservada.

Descripción detallada

Estructura de datos que almacena la información de una región de memoria dentro del mapa de memoria proporcionado por GRUB.

Definición en la línea 86 del archivo multiboot.h.

Documentación de los datos miembro

unsigned int base_addr_high

32 bits más significativos de la base de la región de memoria

Definición en la línea 93 del archivo multiboot.h.

unsigned int base_addr_low

32 bits menos significativos de la base de la región de memoria

Definición en la línea 91 del archivo multiboot.h.

unsigned int entry_size

Campo no usado.

Definición en la línea 89 del archivo multiboot.h.

unsigned int length_high

32 bits más significativos del tamaño de la región de memoria
Definición en la línea 97 del archivo multiboot.h.

unsigned int length_low

32 bits menos significativos del tamaño de la región de memoria
Definición en la línea 95 del archivo multiboot.h.

unsigned int type

Tipo de área de memoria 1 = disponible, 2 = reservada.
Definición en la línea 99 del archivo multiboot.h.

La documentación para esta estructura fue generada a partir del siguiente fichero:

- include/multiboot.h

Referencia de la Estructura mod_info

Estructura de datos que almacena la información de un módulo cargado por GRUB.
`#include <multiboot.h>`

Atributos públicos

- unsigned int mod_start
  Dirección en la cual se cargó el módulo.
- unsigned int mod_end
  Tamaño en bytes del módulo cargado.
- char * string
  32 Comando usado para cargar el módulo
- unsigned int always0

Descripción detallada

Estructura de datos que almacena la información de un módulo cargado por GRUB.

Definición en la línea 104 del archivo multiboot.h.

Documentación de los datos miembro

unsigned int always0

Definición en la línea 111 del archivo multiboot.h.

unsigned int mod_end

Tamaño en bytes del módulo cargado.

Definición en la línea 108 del archivo multiboot.h.

unsigned int mod_start

Dirección en la cual se cargó el módulo.

Definición en la línea 106 del archivo multiboot.h.

char* string

32 Comando usado para cargar el módulo

Definición en la línea 110 del archivo multiboot.h.

La documentación para esta estructura fue generada a partir del siguiente fichero:

- include/multiboot.h

Referencia de la Estructura multiboot_header_struct

Definición del tipo de datos del Encabezado Multiboot.
`#include <multiboot.h>`

Atributos públicos

- unsigned int magic
  Número mágico.
- unsigned int flags
  Flags pasadas a GRUB para solicitar información / servicios.
- unsigned int checksum
  Número de chequeo. Este valor debe ser igual a la suma de magic y flags.
- unsigned int header_addr
  Desplazamiento (Dirección) en el cual se encuentra el encabezado dentro del archivo del kernel.
- unsigned int kernel_start
  Dirección en la cual se debe cargar el kernel.
- unsigned int data_end
  Dirección del final de la sección de datos del kernel.
- unsigned int bss_end
  Dirección del final de la sección BSS del kernel.
- unsigned int entry_point
  Dirección en la cual se debe pasar el control al kernel.

Descripción detallada

Definición del tipo de datos del Encabezado Multiboot.

Definición en la línea 45 del archivo multiboot.h.

Documentación de los datos miembro

unsigned int bss_end

Dirección del final de la sección BSS del kernel.

Definición en la línea 61 del archivo multiboot.h.

unsigned int checksum

Número de chequeo. Este valor debe ser igual a la suma de magic y flags.

Definición en la línea 52 del archivo multiboot.h.

unsigned int data_end

Dirección del final de la sección de datos del kernel.
Definición en la línea 59 del archivo multiboot.h.

unsigned int entry_point

Dirección en la cual se debe pasar el control al kernel.
Definición en la línea 63 del archivo multiboot.h.

unsigned int flags

Flags pasadas a GRUB para solicitar información / servicios.
Definición en la línea 49 del archivo multiboot.h.

unsigned int header_addr

Desplazamiento (Dirección) en el cual se encuentra el encabezado dentro del archivo del kernel.
Definición en la línea 55 del archivo multiboot.h.

unsigned int kernel_start

Dirección en la cual se debe cargar el kernel.
Definición en la línea 57 del archivo multiboot.h.

unsigned int magic

Número mágico.
Definición en la línea 47 del archivo multiboot.h.

La documentación para esta estructura fue generada a partir del siguiente fichero:

- include/multiboot.h

Referencia de la Estructura multiboot_info_t

Estructura de información Multiboot. Al cargar el kernel, GRUB almacena en el registro EBX un apuntador a la dirección de memoria en la que se encuentra esta estructura.

```
#include <multiboot.h>
```

Atributos públicos

- unsigned int flags
  Versión e información de Multiboot. El kernel deberá comprobar sus bits para verificar si GRUB le pasó la información solicitada.
- unsigned int mem_lower
  Presente si flags[0] = 1 Memoria baja reportada por la BIOS.
- unsigned int mem_upper
  Presente si flags[0] = 1 Memoria alta reportada por la BIOS.
- unsigned int boot_device
  Presente si flags[1] = 1 Dispositivo desde el cual se cargó el kernel.
- unsigned int cmdline
  Presente si flags[2] = 1 Línea de comandos usada para cargar el kernel.
- unsigned int mods_count
  Presente si flags[3] = 1 Número de módulos cargados junto con el kernel.
- unsigned int mods_addr
  Presente si flags[3] = 1 Dirección de memoria en la cual se encuentra la información de los módulos cargados por el kernel.
- union {
- aout_symbol_table_t aout_symbol_table
- elf_section_header_table_t elf_section_table
- } syms
- unsigned int mmap_length
  Presente si flags[6] = 1. Tamaño del mapa de memoria creado por GRUB.
- unsigned int mmap_addr
  Presente si flags[6] = 1. Dirección física de la ubicación del mapa de memoria creado por GRUB.
- unsigned int drives_length
  Presente si flags[7] = 1. Especifica el tamaño total de la estructura que describe los drives reportados por la BIOS.
- unsigned int drives_addr
  Presente si flags[7] = 1. Especifica la dirección de memoria en la que se encuentra la estructura que describe los drivers reportados por la BIOS.
- unsigned int config_table
  Presente si flags[8] = 1. Especifica la dirección de la tabla de configuración de la BIOS.
- unsigned int boot_loader_name
  Presente si flags[9] = 1. Contiene la dirección de memoria en la cual se encuentra una cadena de caracteres con el nombre del cargador de arranque.
- unsigned int apm_table
  Presente si flags[10] = 1. Especifica la localización en la memoria de la tabla APM.
- unsigned int vbe_control_info
  Presente si flags[11] = 1. Contiene la información de control retornada por la función vbe 00.

- unsigned int vbe_mode_info
  Presente si flags[11] = 1. Contiene la información de modo retornada por la función vbe 00.
- unsigned int vbe_mode
  Presente si flags[11] = 1.
- unsigned short vbe_interface_seg
  Presente si flags[11] = 1.
- unsigned short vbe_interface_off
  Presente si flags[11] = 1.
- unsigned short vbe_interface_len
  Presente si flags[11] = 1.

Descripción detallada

Estructura de información Multiboot. Al cargar el kernel, GRUB almacena en el registro EBX un apuntador a la dirección de memoria en la que se encuentra esta estructura.

Definición en la línea 117 del archivo multiboot.h.

Documentación de los datos miembro

aout_symbol_table_t aout_symbol_table

Definición en la línea 141 del archivo multiboot.h.

unsigned int apm_table

Presente si flags[10] = 1. Especifica la localización en la memoria de la tabla APM.

Definición en la línea 171 del archivo multiboot.h.

unsigned int boot_device

Presente si flags[1] = 1 Dispositivo desde el cual se cargó el kernel.

Definición en la línea 127 del archivo multiboot.h.

unsigned int boot_loader_name

Presente si flags[9] = 1. Contiene la dirección de memoria en la cual se encuentra una cadena de caracteres con el nombre del cargador de arranque.

Definición en la línea 167 del archivo multiboot.h.

unsigned int cmdline

Presente si flags[2] = 1 Línea de comandos usada para cargar el kernel.

Definición en la línea 130 del archivo multiboot.h.

unsigned int config_table

Presente si flags[8] = 1. Especifica la dirección de la tabla de configuración de la BIOS.

Definición en la línea 163 del archivo multiboot.h.

unsigned int drives_addr

Presente si flags[7] = 1. Especifica la dirección de memoria en la que se encuentra la estructura que describe los drivers reportados por la BIOS.

Definición en la línea 160 del archivo multiboot.h.

unsigned int drives_length

Presente si flags[7] = 1. Especifica el tamaño total de la estructura que describe los drives reportados por la BIOS.

Definición en la línea 155 del archivo multiboot.h.

elf_section_header_table_t elf_section_table

Definición en la línea 142 del archivo multiboot.h.

unsigned int flags

Versión e información de Multiboot. El kernel deberá comprobar sus bits para verificar si GRUB le pasó la información solicitada.

Definición en la línea 120 del archivo multiboot.h.

unsigned int mem_lower

Presente si flags[0] = 1 Memoria baja reportada por la BIOS.

Definición en la línea 122 del archivo multiboot.h.

unsigned int mem_upper

Presente si flags[0] = 1 Memoria alta reportada por la BIOS.

Definición en la línea 124 del archivo multiboot.h.

unsigned int mmap_addr

Presente si flags[6] = 1. Dirección física de la ubicación del mapa de memoria creado por GRUB.

Definición en la línea 150 del archivo multiboot.h.

unsigned int mmap_length

Presente si flags[6] = 1. Tamaño del mapa de memoria creado por GRUB.

Definición en la línea 147 del archivo multiboot.h.

unsigned int mods_addr

Presente si flags[3] = 1 Dirección de memoria en la cual se encuentra la información de los módulos cargados por el kernel.

Definición en la línea 136 del archivo multiboot.h.

unsigned int mods_count

Presente si flags[3] = 1 Número de módulos cargados junto con el kernel.

Definición en la línea 133 del archivo multiboot.h.

union { ... } syms

unsigned int vbe_control_info

Presente si flags[11] = 1. Contiene la información de control retornada por la función vbe 00.

Definición en la línea 176 del archivo multiboot.h.

unsigned short vbe_interface_len

Presente si flags[11] = 1.

Definición en la línea 190 del archivo multiboot.h.

unsigned short vbe_interface_off

Presente si flags[11] = 1.

Definición en la línea 188 del archivo multiboot.h.

unsigned short vbe_interface_seg

Presente si flags[11] = 1.

Definición en la línea 186 del archivo multiboot.h.

unsigned int vbe_mode

Presente si flags[11] = 1.

Definición en la línea 183 del archivo multiboot.h.

unsigned int vbe_mode_info

Presente si flags[11] = 1. Contiene la información de modo retornada por la función vbe 00.

Definición en la línea 180 del archivo multiboot.h.

La documentación para esta estructura fue generada a partir del siguiente fichero:

- include/multiboot.h

Documentación de archivos

Referencia del Archivo dox/00_0_development_environment.dox

Referencia del Archivo dox/00_ia32_intro.dox

Referencia del Archivo dox/01_ia32_operation_modes.dox

Referencia del Archivo dox/02_ia32_memory_organization.dox

Referencia del Archivo dox/03_ia32_execution_environment.dox

Referencia del Archivo dox/04_ia32_protected_mode.dox

Referencia del Archivo dox/05_gdt.dox

Referencia del Archivo dox/05_idt_and_interrupts.dox

Referencia del Archivo dox/05_irq_and_pic.dox

Referencia del Archivo dox/06_bios_and_booting.dox

Referencia del Archivo dox/07_ia32_assembly_basics.dox

Referencia del Archivo dox/08_ia32_using_the_stack.dox

Referencia del Archivo dox/09_ia32_using_routines.dox

Referencia del Archivo dox/10_ia32_using_bios_services.dox

Referencia del Archivo dox/11_multiboot_kernel_loading.dox

Referencia del Archivo dox/12_interrupts_exceptions_and_irq.dox

Referencia del Archivo dox/13_memory_management_bitmaps.dox

Referencia del Archivo dox/19_disk_image_creation.dox

Referencia del Archivo dox/19_disk_image_description.dox

Referencia del Archivo dox/20_text_video_memory.dox

Referencia del Archivo dox/settings.dox

Referencia del Archivo include/asm.h

Rutinas para la ejecución de código ensamblador para la arquitectura IA-32.

'defines'

- #define inline_assembly(code...) __asm__ __volatile__(code)
  Alias para incluir codigo ensamblador dentro del código en C.

Descripción detallada

Rutinas para la ejecución de código ensamblador para la arquitectura IA-32.

Autor:

Erwin Meza emezav@gmail.com

Copyright:

GNU Public License.

Dado que algunas instrucciones sólo se pueden ejecutar directamente en lenguaje ensamblador, se hace uso de inline assembly (assembler en-linea) para incluir código en ensamblador directamente dentro del código en C.

```
asm     volatile  (" instrucciones asm"
    : operandos de salida
    : operandos de entrada
    : registros a invalidar
    );
```

Ver también:

http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html HOWTO de assembler en línea.

Definición en el archivo asm.h.

Documentación de los 'defines'

#define inline_assembly(code...) __asm__ __volatile__(code)

Alias para incluir codigo ensamblador dentro del código en C.

Definición en la línea 34 del archivo asm.h.

Referencia del Archivo include/exception.h

En este archivo se definen los tipos y las funciones relacionados con la gestion de excepciones en la arquitectura IA-32.

```
#include <idt.h>
```

'defines'

- #define MAX_EXCEPTIONS 32

'typedefs'

- typedef interrupt_handler exception_handler

Funciones

- void setup_exceptions (void)
  Esta rutina crea un manejador de interrupcion para las 32 excepciones x86 e inicializa la tabla de manejadores de excepcion.

Descripción detallada

En este archivo se definen los tipos y las funciones relacionados con la gestion de excepciones en la arquitectura IA-32.

Autor:

Erwin Meza emezav@gmail.com

Copyright:

GNU Public License.

Definición en el archivo exception.h.

Documentación de los 'defines'

#define MAX_EXCEPTIONS 32

Importar la definicion del tipo manejador de interrupcion En la arquitectura IA-32 se definen 32 excepciones.

Definición en la línea 19 del archivo exception.h.

Documentación de los 'typedefs'

typedef interrupt_handler exception_handler

Alias para las rutinas de manejo de excepcion

Definición en la línea 22 del archivo exception.h.

Documentación de las funciones

void setup_exceptions (void)

Esta rutina crea un manejador de interrupcion para las 32 excepciones x86 e inicializa la tabla de manejadores de excepcion.

Definición en la línea 71 del archivo exception.c.

Referencia del Archivo include/idt.h

Contiene las definiciones globales requeridas para el manejo de interrupciones en la arquitectura IA-32.

Clases

- struct idt_descriptor
- *Definición de la estructura de datos para un descriptor de interrupción.* struct idt_pointer_t
- *Estructura de datos para el registro IDTR (puntero a la IDT)* struct interrupt_state

Estructura que define el estado del procesador al recibir una interrupción o una excepción. 'defines'

- #define MAX_IDT_ENTRIES 256
  Número de entradas en la IDT: 256 en la arquitectura IA-32.
- #define INTERRUPT_GATE_TYPE 0x0E
  Constante para el tipo de descriptor 'interrupt_gate'.
- #define IF_ENABLE 0x202
  Valor de el registro EFLAGS, con el bit IF = 1. El bit 1 siempre debe ser 1.
- #define NULL_INTERRUPT_HANDLER (interrupt_handler)0
  Constante para definir un manejador de interrupcion vacio.

'typedefs'

- typedef struct idt_descriptor idt_descriptor
  Definición del tipo de datos para el descriptor de segmento.
- typedef struct idt_pointer_t idt_ptr
  Definición del tipo de datos para el apuntador a la GDT.
- typedef struct interrupt_state interrupt_state
  Estructura que define el estado del procesador al recibir una interrupción o una excepción.
- typedef void(* interrupt_handler)(interrupt_state *)
  Definición de tipo para las rutinas de manejo de interrupcion.

Funciones

- struct idt_descriptor __attribute__ ((packed))
- void setup_idt (void)
  Esta rutina se encarga de cargar la IDT.

Variables

- unsigned short offset_low
- unsigned short selector
- unsigned short type
- unsigned int offset_high
- unsigned short limit
  Tamaño de la IDT.
- unsigned int base
  dirección lineal del inicio de la IDT
- unsigned int isr_table []

Referencia a la tabla de rutinas de servicio de interrupcion. Esta tabla se encuentra definida en el archivo isr.S.

- idt_descriptor idt []
  Referencia a la tabla de descriptores de interrupcion.

Descripción detallada

Contiene las definiciones globales requeridas para el manejo de interrupciones en la arquitectura IA-32.

Autor:
Erwin Meza emezav@gmail.com

Copyright:
GNU Public License.

Definición en el archivo idt.h.

Documentación de los 'defines'

#define IF_ENABLE 0x202

Valor de el registro EFLAGS, con el bit IF = 1. El bit 1 siempre debe ser 1.

Definición en la línea 21 del archivo idt.h.

#define INTERRUPT_GATE_TYPE 0x0E

Constante para el tipo de descriptor 'interrupt_gate'.

Definición en la línea 17 del archivo idt.h.

#define MAX_IDT_ENTRIES 256

Número de entradas en la IDT: 256 en la arquitectura IA-32.

Definición en la línea 14 del archivo idt.h.

#define NULL_INTERRUPT_HANDLER (interrupt_handler)0

Constante para definir un manejador de interrupcion vacio.

Definición en la línea 125 del archivo idt.h.

Documentación de los 'typedefs'

typedef struct idt_descriptor idt_descriptor

Definición del tipo de datos para el descriptor de segmento.

Definición en la línea 41 del archivo idt.h.

typedef struct idt_pointer_t idt_ptr

Definición del tipo de datos para el apuntador a la GDT.

Definición en la línea 52 del archivo idt.h.

typedef void(* interrupt_handler)(interrupt_state *)

Definición de tipo para las rutinas de manejo de interrupcion.

Definición en la línea 122 del archivo idt.h.

typedef struct interrupt_state interrupt_state

Estructura que define el estado del procesador al recibir una interrupción o una excepción.

Al recibir una interrupción, el procesador automáticamente almacena en la pila el valor de CS, EIP y EFLAGS. Si la interrupción ocurrió en un nivel de privilegios diferente de cero, antes de almacenar CS, EIP y EFLAGS se almacena el valor de SS y ESP. El control lo recibe el código del archivo isr.S, en el cual almacena (en orden inverso) el estado del procesador contenido en esta estructura.

Documentación de las funciones

struct gdt_pointer_t __attribute__ ((packed))

void setup_idt (void)

Esta rutina se encarga de cargar la IDT.

Definición en la línea 86 del archivo idt.c.

Documentación de las variables

unsigned int base

dirección lineal del inicio de la IDT

Definición en la línea 53 del archivo idt.h.

idt_descriptor idt[]

Referencia a la tabla de descriptores de interrupcion.

unsigned int isr_table[]

Referencia a la tabla de rutinas de servicio de interrupcion. Esta tabla se encuentra definida en el archivo isr.S.

unsigned short limit

Tamaño de la IDT.

Definición en la línea 51 del archivo idt.h.

unsigned int offset_high

Bits más significativos del desplazamiento dentro del segmento de código en el cual se encuentra la rutina de manejo de interrupción

Definición en la línea 49 del archivo idt.h.

unsigned short offset_low

Bits menos significativos del desplazamiento dentro del segmento de código en el cual se encuentra la rutina de manejo de interrupción

Definición en la línea 41 del archivo idt.h.

unsigned short selector

Selector del segmento de código en el cual se encuentra la rutina de manejo de interrupción

Definición en la línea 44 del archivo idt.h.

unsigned short type

Tipo del descriptor

Definición en la línea 46 del archivo idt.h.

Referencia del Archivo include/irq.h

Este archivo define las rutinas publicas para la gestion de solicitudes de interrupcion (IRQ) de los dispositivos de Entrada / Salida.

'defines'

- #define EOI 0x20
  OCW2 (End of Interrupt): Codigo para escribir en el puerto de comandos del PIC para indicar que se ha recibido la interrupción.
- #define MASTER_PIC_COMMAND_PORT 0x20
  Dirección del puerto de comandos del PIC maestro.
- #define SLAVE_PIC_COMMAND_PORT 0xA0
  Dirección del puerto de comandos del PIC esclavo.
- #define MASTER_PIC_DATA_PORT 0x21
  Dirección del puerto de datos del PIC maestro.
- #define SLAVE_PIC_DATA_PORT 0xA1
  Dirección del puerto de datos del PIC esclavo.
- #define IDT_IRQ_OFFSET 32
  Desplazamiento en la IDT a partir de la cual se configuran las rutinas de manejo de interrupción. En IA-32, este debe ser mayor o igual a 32 debido a que las primeras 32 interrupciones son usadas por las excepciones.
- #define IRQ0_INTERRUPT IDT_IRQ_OFFSET + 0
  IRQ del Timer del Sistema.
- #define IRQ1_INTERRUPT IDT_IRQ_OFFSET + 1
  IRQ del Controlador de Teclado.
- #define IRQ2_INTERRUPT IDT_IRQ_OFFSET + 2
  Señal en cascada con el PIC esclavo.
- #define IRQ3_INTERRUPT IDT_IRQ_OFFSET + 3
  IRQ del Controlador del Puerto Serial COM2, si está presente. Compartido con el puerto serial COM4.
- #define IRQ4_INTERRUPT IDT_IRQ_OFFSET + 4
  IRQ del Controlador del Puerto Serial COM1, si está presente. Compartido con el puerto serial COM3.
- #define IRQ5_INTERRUPT IDT_IRQ_OFFSET + 5
  IRQ del puerto 2 de LPT o tarjeta de sonido.
- #define IRQ6_INTERRUPT IDT_IRQ_OFFSET + 6
  IRQ Controlador de Disco Floppy.
- #define IRQ7_INTERRUPT IDT_IRQ_OFFSET + 7
  IRQ del Controlador del puerto 1 de LPT o cualquier dispositivo paralelo.
- #define IRQ8_INTERRUPT IDT_IRQ_OFFSET + 8
  IRQ del Timer RTC.
- #define IRQ9_INTERRUPT IDT_IRQ_OFFSET + 9
  IRQ disponible o usada por el controlador SCSI.
- #define IRQ10_INTERRUPT IDT_IRQ_OFFSET + 10
  IRQ disponible o usada por el controlador SCSI o NIC.
- #define IRQ11_INTERRUPT IDT_IRQ_OFFSET + 11
  IRQ disponible o usada por el controlador SCSI o NIC.

- #define IRQ12_INTERRUPT IDT_IRQ_OFFSET + 12
  IRQ del Mouse o Controlador PS/2.
- #define IRQ13_INTERRUPT IDT_IRQ_OFFSET + 13
  IRQ del Co-Procesador Matemático, FPU o interrupción inter-procesador. a disposición del sistema.
- #define IRQ14_INTERRUPT IDT_IRQ_OFFSET + 14
  IRQ del Canal ATA primario.
- #define IRQ15_INTERRUPT IDT_IRQ_OFFSET + 15
  IRQ del Canal ATA SEcundario (para discos duros o CD)
- #define MAX_IRQ_ROUTINES 16
  Define el número máximo de rutinas de manejo de IRQ que se pueden definir en el sistema.

'typedefs'

- typedef interrupt_handler irq_handler
  Alias para el manejador de irq.

Funciones

- void setup_irq (void)
  Esta rutina se encarga de crear las entradas en la IDT para las interrupciones que se desean manejar. Por defecto configura las interrupciones correspondientes a las IRQ de los dispositivos de entradda/ salida, que han sido mapeadas a los numeros 32..47.
- void install_irq_handler (int number, irq_handler handler)
  Esta rutina permite definir un nuevo manejador de IRQ.
- void uninstall_irq_handler (int number)
  Esta rutina permite quitar un manejador de IRQ.

Descripción detallada

Este archivo define las rutinas publicas para la gestion de solicitudes de interrupcion (IRQ) de los dispositivos de Entrada / Salida.

Autor:
Erwin Meza emezav@gmail.com

Copyright:
GNU Public License.

Definición en el archivo irq.h.

Documentación de los 'defines'

#define EOI 0x20

OCW2 (End of Interrupt): Codigo para escribir en el puerto de comandos del PIC para indicar que se ha recibido la interrupción.

Definición en la línea 17 del archivo irq.h.

#define IDT_IRQ_OFFSET 32

Desplazamiento en la IDT a partir de la cual se configuran las rutinas de manejo de interrupción. En IA-32, este debe ser mayor o igual a 32 debido a que las primeras 32 interrupciones son usadas por las excepciones.

Definición en la línea 35 del archivo irq.h.

#define IRQ0_INTERRUPT IDT_IRQ_OFFSET + 0

IRQ del Timer del Sistema.

Definición en la línea 44 del archivo irq.h.

#define IRQ10_INTERRUPT IDT_IRQ_OFFSET + 10

IRQ disponible o usada por el controlador SCSI o NIC.

Definición en la línea 67 del archivo irq.h.

#define IRQ11_INTERRUPT IDT_IRQ_OFFSET + 11

IRQ disponible o usada por el controlador SCSI o NIC.

Definición en la línea 69 del archivo irq.h.

#define IRQ12_INTERRUPT IDT_IRQ_OFFSET + 12

IRQ del Mouse o Controlador PS/2.

Definición en la línea 71 del archivo irq.h.

#define IRQ13_INTERRUPT IDT_IRQ_OFFSET + 13

IRQ del Co-Procesador Matemático, FPU o interrupción inter-procesador. a disposición del sistema.

Definición en la línea 74 del archivo irq.h.

#define IRQ14_INTERRUPT IDT_IRQ_OFFSET + 14

IRQ del Canal ATA primario.

Definición en la línea 76 del archivo irq.h.

#define IRQ15_INTERRUPT IDT_IRQ_OFFSET + 15

IRQ del Canal ATA SEcundario (para discos duros o CD)

Definición en la línea 78 del archivo irq.h.

#define IRQ1_INTERRUPT IDT_IRQ_OFFSET + 1

IRQ del Controlador de Teclado.
Definición en la línea 46 del archivo irq.h.

#define IRQ2_INTERRUPT IDT_IRQ_OFFSET + 2

Señal en cascada con el PIC esclavo.
Definición en la línea 48 del archivo irq.h.

#define IRQ3_INTERRUPT IDT_IRQ_OFFSET + 3

IRQ del Controlador del Puerto Serial COM2, si está presente. Compartido con el puerto serial COM4.
Definición en la línea 51 del archivo irq.h.

#define IRQ4_INTERRUPT IDT_IRQ_OFFSET + 4

IRQ del Controlador del Puerto Serial COM1, si está presente. Compartido con el puerto serial COM3.
Definición en la línea 54 del archivo irq.h.

#define IRQ5_INTERRUPT IDT_IRQ_OFFSET + 5

IRQ del puerto 2 de LPT o tarjeta de sonido.
Definición en la línea 56 del archivo irq.h.

#define IRQ6_INTERRUPT IDT_IRQ_OFFSET + 6

IRQ Controlador de Disco Floppy.
Definición en la línea 58 del archivo irq.h.

#define IRQ7_INTERRUPT IDT_IRQ_OFFSET + 7

IRQ del Controlador del puerto 1 de LPT o cualquier dispositivo paralelo.
Definición en la línea 61 del archivo irq.h.

#define IRQ8_INTERRUPT IDT_IRQ_OFFSET + 8

IRQ del Timer RTC.
Definición en la línea 63 del archivo irq.h.

#define IRQ9_INTERRUPT IDT_IRQ_OFFSET + 9

IRQ disponible o usada por el controlador SCSI.
Definición en la línea 65 del archivo irq.h.

#define MASTER_PIC_COMMAND_PORT 0x20

Dirección del puerto de comandos del PIC maestro.
Definición en la línea 20 del archivo irq.h.

#define MASTER_PIC_DATA_PORT 0x21

Dirección del puerto de datos del PIC maestro.
Definición en la línea 26 del archivo irq.h.

#define MAX_IRQ_ROUTINES 16

Define el número máximo de rutinas de manejo de IRQ que se pueden definir en el sistema.
Definición en la línea 82 del archivo irq.h.

#define SLAVE_PIC_COMMAND_PORT 0xA0

Dirección del puerto de comandos del PIC esclavo.
Definición en la línea 23 del archivo irq.h.

#define SLAVE_PIC_DATA_PORT 0xA1

Dirección del puerto de datos del PIC esclavo.
Definición en la línea 29 del archivo irq.h.

Documentación de los 'typedefs'

typedef interrupt_handler irq_handler

Alias para el manejador de irq.
Definición en la línea 39 del archivo irq.h.

Documentación de las funciones

void install_irq_handler (int *number*, irq_handler *handler*)

Esta rutina permite definir un nuevo manejador de IRQ.

Parámetros:

number	numero de irq a configurar
handler	Función a manejar la irq

Definición en la línea 150 del archivo irq.c.

void setup_irq (void)

Esta rutina se encarga de crear las entradas en la IDT para las interrupciones que se desean manejar. Por defecto configura las interrupciones correspondientes a las IRQ de los dispositivos de entradda/ salida, que han sido mapeadas a los numeros 32..47.

Definición en la línea 126 del archivo irq.c.

void uninstall_irq_handler (int *number*)

Esta rutina permite quitar un manejador de IRQ.

Parámetros:

number	numero de irq a quitar

Devuelve:

void

Definición en la línea 164 del archivo irq.c.

Referencia del Archivo include/multiboot.h

Contiene algunas constantes necesarias para el kernel relacionadas con la especificación Multiboot.

Clases

- struct multiboot_header_struct
- *Definición del tipo de datos del Encabezado Multiboot.* struct aout_symbol_table
- *Tabla de símbolos usadas en el formato a.out.* struct elf_section_header_table
- *Tabla de encabezados de sección del kernel en el formato elf.* struct memory_map
- *Estructura de datos que almacena la información de una región de memoria dentro del mapa de memoria proporcionado por GRUB.* struct mod_info
- *Estructura de datos que almacena la información de un módulo cargado por GRUB.* struct multiboot_info_t

Estructura de información Multiboot. Al cargar el kernel, GRUB almacena en el registro EBX un apuntador a la dirección de memoria en la que se encuentra esta estructura. 'defines'

- #define KERNADDR 0x100000
  Dirección física del kernel en memoria.
- #define MULTIBOOT_PAGE_ALIGN 1<<0
  Alinear los módulos cargados a límites de página.
- #define MULTIBOOT_MEMORY_INFO 1<<1
  Proporcionar al kernel información de la memoria disponible.
- #define MULTIBOOT_VIDEO_INFO 1<<2
  Proporcionar al kernel información de los modos de video.
- #define MULTIBOOT_AOUT_KLUDGE 1<<16
  Solicitar a Grub que use las direcciones proporcionados en el encabezado multiboot.
- #define MULTIBOOT_HEADER_MAGIC 0x1BADB002
  Número mágico de la especificacón multiboot.
- #define MULTIBOOT_HEADER_FLAGS
  Constante que incluye las flags que se pasarán a GRUB.
- #define MULTIBOOT_CHECKSUM -(MULTIBOOT_HEADER_MAGIC + MULTIBOOT_HEADER_FLAGS)
  Constante de suma de chequeo.
- #define MULTIBOOT_BOOTLOADER_MAGIC 0x2BADB002
  Número mágico que el cargador de arranque almacena en el registro EAX para indicar que es compatible con la especificación Multiboot.

'typedefs'

- typedef struct
- multiboot_header_struct multiboot_header_t
  Definición del tipo de datos del Encabezado Multiboot.
- typedef struct aout_symbol_table aout_symbol_table_t
  Tabla de símbolos usadas en el formato a.out.
- typedef struct
- elf_section_header_table elf_section_header_table_t
  Tabla de encabezados de sección del kernel en el formato elf.

- typedef struct memory_map memory_map_t
  Estructura de datos que almacena la información de una región de memoria dentro del mapa de memoria proporcionado por GRUB.
- typedef struct mod_info mod_info_t
  Estructura de datos que almacena la información de un módulo cargado por GRUB.

Descripción detallada

Contiene algunas constantes necesarias para el kernel relacionadas con la especificación Multiboot.

Autor:
Erwin Meza emezav@gmail.com

Copyright:
GNU Public License.

Ver también:
http://www.gnu.org/software/grub/manual/multiboot/multiboot.html

Definición en el archivo multiboot.h.

Documentación de los 'defines'

#define KERNADDR 0x100000

Dirección física del kernel en memoria.

Definición en la línea 16 del archivo multiboot.h.

#define MULTIBOOT_AOUT_KLUDGE 1<<16

Solicitar a Grub que use las direcciones proporcionados en el encabezado multiboot.

Definición en la línea 27 del archivo multiboot.h.

#define MULTIBOOT_BOOTLOADER_MAGIC 0x2BADB002

Número mágico que el cargador de arranque almacena en el registro EAX para indicar que es compatible con la especificación Multiboot.

Definición en la línea 38 del archivo multiboot.h.

#define MULTIBOOT_CHECKSUM -(MULTIBOOT_HEADER_MAGIC + MULTIBOOT_HEADER_FLAGS)

Constante de suma de chequeo.

Definición en la línea 34 del archivo multiboot.h.

#define MULTIBOOT_HEADER_FLAGS

```
Valor:MULTIBOOT_PAGE_ALIGN | MULTIBOOT_MEMORY_INFO | \
          MULTIBOOT_AOUT_KLUDGE
```

Constante que incluye las flags que se pasarán a GRUB.

Definición en la línea 31 del archivo multiboot.h.

#define MULTIBOOT_HEADER_MAGIC 0x1BADB002

Número mágico de la especificacón multiboot.

Definición en la línea 29 del archivo multiboot.h.

#define MULTIBOOT_MEMORY_INFO 1<<1

Proporcionar al kernel información de la memoria disponible.

Definición en la línea 22 del archivo multiboot.h.

#define MULTIBOOT_PAGE_ALIGN 1<<0

Alinear los módulos cargados a límites de página.

Definición en la línea 20 del archivo multiboot.h.

#define MULTIBOOT_VIDEO_INFO 1<<2

Proporcionar al kernel información de los modos de video.

Definición en la línea 24 del archivo multiboot.h.

Documentación de los 'typedefs'

typedef struct aout_symbol_table aout_symbol_table_t

Tabla de símbolos usadas en el formato a.out.

typedef struct elf_section_header_table elf_section_header_table_t

Tabla de encabezados de sección del kernel en el formato elf.

typedef struct memory_map memory_map_t

Estructura de datos que almacena la información de una región de memoria dentro del mapa de memoria proporcionado por GRUB.

typedef struct mod_info mod_info_t

Estructura de datos que almacena la información de un módulo cargado por GRUB.

typedef struct multiboot_header_struct multiboot_header_t

Definición del tipo de datos del Encabezado Multiboot.

Referencia del Archivo include/physmem.h

Contiene las definiciones relacionadas con las gestión de memoria del kernel.

'defines'

- #define MMAP_LOCATION 0x500
  Localizacion del mapa de bits de memoria.
- #define MEMORY_UNIT_SIZE 4096
  Tamaño de la unidad de asignación de memoria.
- #define BYTES_PER_ENTRY sizeof(unsigned int)
  Número de bytes que tiene una entrada en el mapa de bits.
- #define BITS_PER_ENTRY (8 * BYTES_PER_ENTRY)
  Número de bits que tiene una entrada en el mapa de bits.
- #define MEMORY_UNITS (memory_length / MEMORY_UNIT_SIZE)
  Número de unidades en la memoria disponible.
- #define bitmap_entry(addr) (addr / MEMORY_UNIT_SIZE) / (BITS_PER_ENTRY)
  Entrada en el mapa de bits correspondiente a una dirección.
- #define bitmap_offset(addr) (addr / MEMORY_UNIT_SIZE) % (BITS_PER_ENTRY)
  Desplazamiento en bits dentro de la entrada en el mapa de bits.

Funciones

- void setup_memory (void)
  Esta rutina inicializa el mapa de bits de memoria, a partir de la informacion obtenida del GRUB.
- char * allocate_unit (void)
  Busca una unidad libre dentro del mapa de bits de memoria.
- char * allocate_unit_region (unsigned int length)
  Busca una región de memoria contigua libre dentro del mapa de bits de memoria.
- void free_unit (char *addr)
  Permite liberar una unidad de memoria.
- void free_region (char *start_addr, unsigned int length)
  Permite liberar una región de memoria.

Descripción detallada

Contiene las definiciones relacionadas con las gestión de memoria del kernel.

Autor:
Erwin Meza emezav@gmail.com

Copyright:
GNU Public License.

Definición en el archivo physmem.h.

Documentación de los 'defines'

#define bitmap_entry(addr) (addr / MEMORY_UNIT_SIZE) / (BITS_PER_ENTRY)

Entrada en el mapa de bits correspondiente a una dirección.

Definiendo la la direccion con el parametro de entrada addr, la direccion en el mapa de bits que ocupa

Definición en la línea 46 del archivo physmem.h.

#define bitmap_offset(addr) (addr / MEMORY_UNIT_SIZE) % (BITS_PER_ENTRY)

Desplazamiento en bits dentro de la entrada en el mapa de bits.

Definición en la línea 50 del archivo physmem.h.

#define BITS_PER_ENTRY (8 * BYTES_PER_ENTRY)

Número de bits que tiene una entrada en el mapa de bits.

Equivale a que el numero de bits por cada unidad de memoria en este caso, son 32bits

Definición en la línea 33 del archivo physmem.h.

#define BYTES_PER_ENTRY sizeof(unsigned int)

Número de bytes que tiene una entrada en el mapa de bits.

Equivale a que el numero de bytes por cada unidad de memoria

Definición en la línea 27 del archivo physmem.h.

#define MEMORY_UNIT_SIZE 4096

Tamaño de la unidad de asignación de memoria.

El valor por default es 4096 es decir 4KB

Definición en la línea 22 del archivo physmem.h.

#define MEMORY_UNITS (memory_length / MEMORY_UNIT_SIZE)

Número de unidades en la memoria disponible.

MEMORY_UNITS calcula el numero de unidades que estan disponible para ello hace la operacion de el tamaño total de memoria disponible dividido por MEMORY_UNIT_SIZE

Definición en la línea 40 del archivo physmem.h.

#define MMAP_LOCATION 0x500

Localizacion del mapa de bits de memoria.

En esta direccion es donde esta localizada la estructura de datos del mapa de bits

Definición en la línea 17 del archivo physmem.h.

Documentación de las funciones

char* allocate_unit (void)

Busca una unidad libre dentro del mapa de bits de memoria.

Devuelve:

Dirección de inicio de la unidad en memoria. El proceso que realiza esta funcion es,

```
1.Verificar si no existen unidades libres, de ser asi, el valor de
  retorno es 0.
2.Luego si existen unidades libres, entonces, se inicia una iteracion
  en busqueda de una unidad de memoria libre por el mapa de bits.
3.De no existir una unidad libre dentro del mapa de bits, entonces
  devuelve 0;
  En caso de que si, devuelve la direccion de inicio lineal en la
  de donde se encuentra libre la unidad encontrada, para ello lo
  hace multiplicando las unidades libres de memoria por el tamaño
  de asignacion de una unidad de memoria;
```

Dirección de inicio de la unidad en memoria.

```
Define las unidades disponibles de memoria
```

que dispone

Definición en la línea 288 del archivo physmem.c.

char* allocate_unit_region (unsigned int *length*)

Busca una región de memoria contigua libre dentro del mapa de bits de memoria.

Parámetros:

length	Tamaño de la región de memoria a asignar.

Devuelve:

Dirección de inicio de la región en memoria.

El proceso que realiza esta funcion es,

```
1.Verificar si no existen unidades libres, de ser asi, el valor de
  retorno es 0
2.Luego si existen unidades libres, entonces, se inicia una iteracion
  en busqueda de una unidad de memoria disponible.
3.De no existir una unidad libre dentro del mapa de bits, entonces
  devuelve 0; en caso de que si, devuelve la direccion de inicio
  de la region de memoria multiplicando las unidades libres
  de memoria por el tamaño de asignacin de una unidad de memoria.
```

Definición en la línea 332 del archivo physmem.c.

void free_region (char * *start_addr*, unsigned int *length*)

Permite liberar una región de memoria.

Parámetros:

start_addr	Dirección de memoria del inicio de la región a liberar
length	Tamaño de la región a liberar

Para permitir la liberacion de una region de memoria, se siguen estos pasos;

```
1.Si la direccion que se recibe es menor a la unidad minima permitida de
 asignacion de memoria, significa que se quiere liberar memoria que no
 esta disponible, ya que esta usada por el KERNEL o los modulos que se
 cargan con el KERNEL, en este caso termina sin realizar la peticion.
2.Se convierte a una direccion lineal para poner en el mapa de bits como
 disponible mediante un ciclo permitiendo liberar toda la region de memoria
 en unidades de memoria disponibles.
3.Se marca esa unidad liberada, como la proxima unidad para asignar, y aumenta
 el numero de unidades libres.
```

Definición en la línea 417 del archivo physmem.c.

void free_unit (char * *addr*)

Permite liberar una unidad de memoria.

Parámetros:

addr	Dirección de memoria dentro del área a liberar.

Para permitir la liberacion de una unidad de memoria, se define la secuencia de pasos como:

```
1.Si la direccion que se recibe es menor a la unidad minima permitida de
 asignacion de memoria, significa que se quiere liberar memoria que no
 esta disponible, ya que esta usada por el KERNEL o los modulos que se
 cargan con el KERNEL, en este caso termina sin realizar la peticion.
2.Se convierte a una direccion lineal para ponerla en el mapa de bits como
 disponible.
3.Se marca esa unidad liberada, como la proxima unidad para asignar, y aumenta
 el numero de unidades libres.
```

Definición en la línea 392 del archivo physmem.c.

void setup_memory (void)

Esta rutina inicializa el mapa de bits de memoria, a partir de la informacion obtenida del GRUB.

Basicamente lo que se hace es configurar el mapa de bits de la memoria, para ello la secuencia de pasos que realiza dicha configuracion es:

```
  1.Cargar una estructura de tipo multiboot_info_t con la
   informacion del multiboot que es la informacion que el GRUB
   almacena luego de cargar el KERNEL.
  2.Limpia el mapa de bits, haciendo memory bitmap[i] = 0,
      donde i=1,2,3.... (memory_bitmap_length-1)
  3.Verifica si el GRUB cargo la informacion solicitada en la
   estructura del multiboot. En el proyecto actual, se omite
   esta parte.
  4.Se establece, la minima direccion de memoria permitida para
   liberar, a esta se le asignara la direccion lineal de donde
   termina el KERNEL.
  5.Verifica si la direccion del mapa de bits son validos, para ello
   verifica que el bit info->flags[6] sea 1, lo que sifnifica que  mmap_length
   y mmap addr son validos .
  6.Verifica que la region de memoria cumple con las condiciones
```

```
   para ser considerada como Memoria Disponible.
   Para ello, se debe considerar que:
   ->La primera unidad de memoria este ubicada en la posición de
   memoria mayor o igual que 1 MB.
   ->nmap->type que es el tipo de area de memoria si es 1.
   6,1.Lo que significa disponible, de lo contrario es que ya esta reservada.
    Si la region esta marcada como disponible y la posicion esta por
    encima de la direccion  de la posicion del KERNEL en memora,
    se procede a verificar si el KERNEL se encuentre en esa region.
    Entonces se toma el inicio de la memoria disponible en la posicion
    en la cual finaliza el KERNEL.
    Seguido a esto, se verifica si se cargaron los modulos con el KERNEL
    En caso de que hayan modulos que se carguen con el KERNEL, entonces
    la direccion de posicion inicial cambian ahora y empezadia desde luego
    de los modulos cargado; de lo contrario no se afecta.
    y por ultimo se resta el espacio que ahora dispone.

   6,2.Si la region ya esta reservada, y/o que la primera unidad de memoria
       es menor que 1 MB, lo que significa que el KERNEL no se encuentra en
       esta region, entonces el tamaño es mayor que la region encontrada.

7.Si existe una region de memoria disponible, es decir si la direccion donde
 inicia la primera unidad de memoria es valida, y el tamaño disponible es
 valido.
 Se procede a establecer en que direccion de memoria se debe ubicar para
 la asignacion de memoria permitida a liberar, para esto, se calcula la
 direccion en la cual finaliza la memoria disponibe, hace unos ajustes de
 redondeo, se calcula el tamaño de region disponible con las unidades ya
 redondeadas, y se procede a establecer las variables globales del KERNEL.
8.Se marca la direccion de memoria como disponibley la direccion de memoria
```

de la cual se puede liberar memoria.

Existe un mapa de memoria válido creado por GRUB?

Si verifica que el bit en la posición 6 es 1, entonces se crea un mapa valido de memoria, es decir que mmap_length y mmap_addr son validos

Definición en la línea 67 del archivo physmem.c.

Referencia del Archivo include/pm.h

Contiene las definiciones relacionadas con el Modo Protegido IA-32.

Clases

- struct gdt_descriptor
- *Estructura de datos para un descriptor de segmento.* struct gdt_pointer_t

Estructura de datos para el registro GDTR (puntero a la GDT) 'defines'

- #define MAX_GDT_ENTRIES 1024
  Número m{aximo de entradas de la GDT.
- #define CODE_SEGMENT 0xA
  Tipo de segmento de código.
- #define DATA_SEGMENT 0x2
  Tipo de segmento de datos.
- #define KERNEL_CODE_SELECTOR 0x08
  Desplazamiento en bytes dentro de la GDT a partir del cual se encuentra el descriptor de segmento de código del kernel. Se debe tener en cuenta que cada descriptor de segmento ocupa 8 bytes.
- #define KERNEL_DATA_SELECTOR 0x10
  Desplazamiento en bytes dentro de la GDT a partir del cual se encuentra el descriptor de segmento de datos del kernel. Se debe tener en cuenta que cada descriptor de segmento ocupa 8 bytes.
- #define RING0_DPL 0
  Nivel de privilegios 0.
- #define RING1_DPL 1
  Nivel de privilegios 1.
- #define RING2_DPL 2
  Nivel de privilegios 2.
- #define RING3_DPL 3
  Nivel de privilegios 3.

'typedefs'

- typedef struct gdt_descriptor gdt_descriptor
  Tipo de datos para el descriptor de segmento.
- typedef struct gdt_pointer_t gdt_ptr
  Tipo de datos para el apuntador a la GDT.

Funciones

- struct gdt_descriptor __attribute__ ((packed))
- unsigned short get_gdt_selector (gdt_descriptor *desc)
  Función que permite obtener el selector en la GDT a partir de un apuntador a un descriptor de segmento.
- gdt_descriptor * get_gdt_descriptor (unsigned short selector)
  Función que permite obtener el descriptor de segmento en la GDT a partir de un selector.
- gdt_descriptor * allocate_gdt_descriptor (void)
  Esta rutina permite obtener un descriptor de segmento disponible en la GDT.
- unsigned short allocate_gdt_selector (void)

Esta rutina permite obtener un descriptor de segmento disponible en la GDT.

- void free_gdt_descriptor (gdt_descriptor *)
  Esta rutina permite liberar un descriptor de segmento en la GDT.
- void setup_gdt_descriptor (unsigned short, unsigned int, unsigned int, char, char, int, char)
  Permite configurar un descriptor de segmento dentro de la GDT.
- void setup_gdt (void)
  Esta función se encarga de cargar la GDT.

Variables

- unsigned int low
  Bits menos significativos del descriptor. Agrupan Limite 0..15 y Base 0..15 del descriptor.
- unsigned int high
- unsigned short limit
- unsigned int base
- gdt_descriptor gdt []
  Tabla Global de Descriptores (GDT). Es un arreglo de descriptores de segmento. Según el manual de Intel, esta tabla debe estar alineada a un límite de 8 bytes para un óptimo desempeño.

Descripción detallada

Contiene las definiciones relacionadas con el Modo Protegido IA-32.

Autor:
Erwin Meza emezav@gmail.com

Copyright:
GNU Public License.

Definición en el archivo pm.h.

Documentación de los 'defines'

#define CODE_SEGMENT 0xA

Tipo de segmento de código.

Definición en la línea 16 del archivo pm.h.

#define DATA_SEGMENT 0x2

Tipo de segmento de datos.

Definición en la línea 18 del archivo pm.h.

#define KERNEL_CODE_SELECTOR 0x08

Desplazamiento en bytes dentro de la GDT a partir del cual se encuentra el descriptor de segmento de código del kernel. Se debe tener en cuenta que cada descriptor de segmento ocupa 8 bytes.

Definición en la línea 23 del archivo pm.h.

#define KERNEL_DATA_SELECTOR 0x10

Desplazamiento en bytes dentro de la GDT a partir del cual se encuentra el descriptor de segmento de datos del kernel. Se debe tener en cuenta que cada descriptor de segmento ocupa 8 bytes.

Definición en la línea 28 del archivo pm.h.

#define MAX_GDT_ENTRIES 1024

Número m{aximo de entradas de la GDT.

Definición en la línea 13 del archivo pm.h.

#define RING0_DPL 0

Nivel de privilegios 0.

Definición en la línea 31 del archivo pm.h.

#define RING1_DPL 1

Nivel de privilegios 1.

Definición en la línea 33 del archivo pm.h.

#define RING2_DPL 2

Nivel de privilegios 2.

Definición en la línea 35 del archivo pm.h.

#define RING3_DPL 3

Nivel de privilegios 3.

Definición en la línea 37 del archivo pm.h.

Documentación de los 'typedefs'

typedef struct gdt_descriptor gdt_descriptor

Tipo de datos para el descriptor de segmento.

Definición en la línea 102 del archivo pm.h.

typedef struct gdt_pointer_t gdt_ptr

Tipo de datos para el apuntador a la GDT.

Definición en la línea 111 del archivo pm.h.

Documentación de las funciones

struct gdt_descriptor __attribute__ ((packed))

gdt_descriptor* allocate_gdt_descriptor (void)

Esta rutina permite obtener un descriptor de segmento disponible en la GDT.

Devuelve:

Referencia al próximo descriptor de segmento dentro de la GDT que se encuentra disponible, nulo en caso que no exista una entrada disponible dentro de la GDT.

Definición en la línea 81 del archivo pm.c.

unsigned short allocate_gdt_selector (void)

Esta rutina permite obtener un descriptor de segmento disponible en la GDT.

Devuelve:

Referencia al próximo descriptor de segmento dentro de la GDT que se encuentra disponible, nulo en caso que no exista una entrada disponible dentro de la GDT.

Esta rutina permite obtener un descriptor de segmento disponible en la GDT.

Devuelve:

Selector que apunta al descriptor de segmento disponible encontrado dentro de la GDT

Definición en la línea 108 del archivo pm.c.

void free_gdt_descriptor (gdt_descriptor * *desc*)

Esta rutina permite liberar un descriptor de segmento en la GDT.

Parámetros:

desc	Apuntador al descriptor que se desa liberar

Definición en la línea 118 del archivo pm.c.

gdt_descriptor* get_gdt_descriptor (unsigned short *selector*)

Función que permite obtener el descriptor de segmento en la GDT a partir de un selector.

Parámetros:

selector	Selector que permite obtener un descriptor de segmento de la GDT

Devuelve:

Referencia al descriptor de segmento dentro de la GDT

Definición en la línea 65 del archivo pm.c.

unsigned short get_gdt_selector (gdt_descriptor * *desc*)

Función que permite obtener el selector en la GDT a partir de un apuntador a un descriptor de segmento.

Parámetros:

desc	Referencia al descriptor de segmento del cual se desea obtener el selector

Devuelve:

Selector que referencia al descriptor dentro de la GDT.

Definición en la línea 47 del archivo pm.c.

void setup_gdt (void)

Esta función se encarga de cargar la GDT.

Esta función se encarga de configurar la Tabla Global de Descriptores (GDT) que usará el kernel. La GDT es un arreglo de descriptores de segmento, cada uno de los cuales contiene la información de los segmentos en memoria. Esta rutina realiza los siguientes pasos:

- Buscar el espacio en la GDT para el descriptor de segmento de código del kernel
- Verificar el offset dentro de la GDT. Debe ser igual a la constante KERNEL_CODE_SELECTOR Definida en pm.h
- Buscar espacio en la GDT para el descriptor de segmento de datos del kernel
- Verificar el offset dentro de la GDT. Debe ser igual a la constante KERNEL_DATA_SELECTOR definida en pm.h
- Definir el segmento de código del kernel como un segmento plano (base = 0, límite = 4 GB, nivel de privilegios 0). Para describir este segmento se usará la segunda entrada de la GDT. El manual de Intel especifica que la primera entrada de la GDT siempre debe ser nula.
- Definir el segmento de datos del kernel como un segmento plano (base = 0, límite = 4 GB, nivel de privilegios 0). Para describir este segmento se usará la tercera entrada de la GDT.
- La instrucción LGDT recibe un apuntador que tiene dos atributos: Tamaño del GDT - 1 y dirección lineal de la GDT en memoria.
- Ejecutar el siguiente código en ensamblador, para cargar la GDT y pasar "de nuevo" a modo protegido.

```
*/
inline assembly(".intel syntax noprefix\n\t\
                          cli                             \n\t\
                          lgdt [%0]                       \n\t\
                          mov edx, cr0                    \n\t\
                          or edx, 1                       \n\t\
                          mov cr0, edx                    \n\t\
                          push %2                         \n\t\
                          push 0  /* eflags */            \n\t\
```

```
                        push %1                         \n\t\
                        push OFFSET 1f                  \n\t\
                        iret                            \n\t\
                1:                                      \n\t\
                        pop ecx                         \n\t\
                        mov ds, cx                      \n\t\
                        mov es, cx                      \n\t\
                        mov fs, cx                      \n\t\
                        mov gs, cx                      \n\t\
                        mov ss, cx                      \n\t\
                        .att syntax prefix\n\t"
                        :
                        : "am"(&gdt_pointer),
                        "b"(kernel_code_selector & 0x0000FFFF),
                        "c"(kernel data selector & 0x0000FFFF)
                        : "dx");
```

/**

Este código en lenguaje ensamblador realiza las siguientes acciones:

1. Deshabilitar las interrupciones
2. Ejecutar la instrucción lgdt, pasando como parámetro un apuntador a la GDT configurada.
3. Activar el bit PE (Protection Enable) en el registro CR0.
4. Simular un retorno de interrupción, para que el registro de segmento CS contenga el valor del selector que referencia al descriptor de segmento de código configurado en la GDT.
5. Actualizar los registros de segmento DS, ES, FS, GS y SS para que contengan el valor del selector que referencia al descriptor de segmento de datos configurado en la GDT.

Observe que básicamente se está pasando de nuevo a modo protegido, esta vez usando la GDT configurada en esta función.

Definición en la línea 178 del archivo pm.c.

void setup_gdt_descriptor (unsigned short *selector*, unsigned int *base*, unsigned int *limit*, char *type*, char *dpl*, int *code_or_data*, char *opsize*)

Permite configurar un descriptor de segmento dentro de la GDT.

Parámetros:

selector	Selector que referencia al descriptor de segmento dentro de la GDT
base	Dirección lineal del inicio del segmento en memoria
limit	Tamaño del segmento
type	Tipo de segmento
dpl	Nivel de privilegios del segmento
code_or_data	1 = Segmento de código o datos, 0 = segmento del sistema
opsize	Tamaño de operandos: 0 = 16 bits, 1 = 32 bits

Definición en la línea 142 del archivo pm.c.

Documentación de las variables

unsigned int base

Definición en la línea 110 del archivo pm.h.

gdt_descriptor gdt[]

Tabla Global de Descriptores (GDT). Es un arreglo de descriptores de segmento. Según el manual de Intel, esta tabla debe estar alineada a un límite de 8 bytes para un óptimo desempeño.

unsigned int high

bits más significativos del descriptor. Agrupan Base 16..23, Tipo, S, DPL, P, Límite, AVL, L, D/B, G y Base 24..31

Definición en la línea 105 del archivo pm.h.

unsigned short limit

Definición en la línea 109 del archivo pm.h.

unsigned int low

Bits menos significativos del descriptor. Agrupan Limite 0..15 y Base 0..15 del descriptor.

Definición en la línea 102 del archivo pm.h.

Referencia del Archivo include/stdio.h

Contiene las primitivas basicas para entrada / salida.
`#include <asm.h>`

'defines'

- #define VIDEO_ADDR 0XB8000
- #define SCREEN_LINES 25
  Número de líneas de la pantalla.
- #define SCREEN_COLUMNS 80
  Número de columnas de la pantalla.
- #define TABSIZE 8
  Espacios en un tabulador.
- #define COLOR(fg, bg) ((bg << 4) | fg)
  Macro para calcular el byte de atributos de texto y fondo.
- #define BLACK 0
  Negro.
- #define BLUE 1
  Azul.
- #define GREEN 2
  Verde.
- #define CYAN 3
  Cyan.
- #define RED 4
  Rojo.
- #define MAGENTA 5
  Magenta.
- #define BROWN 6
  Café
- #define LIGHTGRAY 7
  Gris claro.
- #define DARKGRAY 8
  Gris oscuro.
- #define LIGHTBLUE 9
  Azul claro.
- #define LIGHTGREEN 10
  Verde claro.
- #define LIGHTCYAN 11
  Cyan claro.
- #define LIGHTRED 12
  Rojo claro.
- #define LIGHTMAGENTA 13
  Magenta claro.
- #define LIGHTBROWN 14
  Café claro.

- #define WHITE 15
  Blanco.
- #define SPACE 0x20
  Caracter ASCII de espacio.
- #define BACKSPACE 0x08
  Caracter ASCII de backspace.
- #define LF '\n'
  Caracter ASCII de Fin de línea.
- #define CR '\r'
  Caracter ASCII de Retorno de carro.
- #define TAB 0x09
  Caracter ASCII de Tabulador.

Funciones

- void putchar (char c)
  Función para imprimir un caracter.
- void puts (char *s)
  Función para imprimir una cadena de caracteres.
- void printf (char *,...)
  Esa funcion implementa en forma basica el comportamiento de 'printf' en C.

Variables

- unsigned short * videoptr
  Apuntador al inicio de la memoria de video.
- char text_attributes
  Byte que almacena los atributos de texto.
- int screen_lines
  Variable que controla el número de lineas de la pantalla.
- int screen_columns
  Variable que controla el número de columnas de la pantalla.

Descripción detallada

Contiene las primitivas basicas para entrada / salida.

Autor:
Erwin Meza emezav@gmail.com

Copyright:
GNU Public License.

Definición en el archivo stdio.h.

Documentación de los 'defines'

#define BACKSPACE 0x08

Caracter ASCII de backspace.
Definición en la línea 98 del archivo stdio.h.

#define BLACK 0

Negro.
Definición en la línea 63 del archivo stdio.h.

#define BLUE 1

Azul.
Definición en la línea 65 del archivo stdio.h.

#define BROWN 6

Café
Definición en la línea 75 del archivo stdio.h.

#define COLOR(fg, bg) ((bg << 4) | fg)

Macro para calcular el byte de atributos de texto y fondo.
Definición en la línea 59 del archivo stdio.h.

#define CR '\r'

Caracter ASCII de Retorno de carro.
Definición en la línea 102 del archivo stdio.h.

#define CYAN 3

Cyan.
Definición en la línea 69 del archivo stdio.h.

#define DARKGRAY 8

Gris oscuro.
Definición en la línea 79 del archivo stdio.h.

#define GREEN 2

Verde.
Definición en la línea 67 del archivo stdio.h.

#define LF '\n'

Caracter ASCII de Fin de línea.
Definición en la línea 100 del archivo stdio.h.

#define LIGHTBLUE 9

Azul claro.
Definición en la línea 81 del archivo stdio.h.

#define LIGHTBROWN 14

Café claro.
Definición en la línea 91 del archivo stdio.h.

#define LIGHTCYAN 11

Cyan claro.
Definición en la línea 85 del archivo stdio.h.

#define LIGHTGRAY 7

Gris claro.
Definición en la línea 77 del archivo stdio.h.

#define LIGHTGREEN 10

Verde claro.
Definición en la línea 83 del archivo stdio.h.

#define LIGHTMAGENTA 13

Magenta claro.
Definición en la línea 89 del archivo stdio.h.

#define LIGHTRED 12

Rojo claro.

Definición en la línea 87 del archivo stdio.h.

#define MAGENTA 5

Magenta.
Definición en la línea 73 del archivo stdio.h.

#define RED 4

Rojo.
Definición en la línea 71 del archivo stdio.h.

#define SCREEN_COLUMNS 80

Número de columnas de la pantalla.
Definición en la línea 47 del archivo stdio.h.

#define SCREEN_LINES 25

Número de líneas de la pantalla.
Definición en la línea 44 del archivo stdio.h.

#define SPACE 0x20

Caracter ASCII de espacio.
Definición en la línea 96 del archivo stdio.h.

#define TAB 0x09

Caracter ASCII de Tabulador.
Definición en la línea 104 del archivo stdio.h.

#define TABSIZE 8

Espacios en un tabulador.
Definición en la línea 50 del archivo stdio.h.

#define VIDEO_ADDR 0XB8000

Definición en la línea 17 del archivo stdio.h.

#define WHITE 15

Blanco.

Definición en la línea 93 del archivo stdio.h.

Documentación de las funciones

void printf (char * *format*, ...)

Esa funcion implementa en forma basica el comportamiento de 'printf' en C.

Parámetros:

format	Formato de la cadena de salida
...	Lista de referencias a memoria de las variables a imprimir

Definición en la línea 311 del archivo stdio.c.

void putchar (char *c*)

Función para imprimir un caracter.

Esta rutina imprime directamente en la memoria de video. No valida caracteres especiales.

Parámetros:

c	caracter ascii a imprimir

Esta rutina imprime directamente en la memoria de video. Valida caracteres especiales, como fin de línea, tabulador y backspace.

Parámetros:

c	caracter ascii a imprimir

Definición en la línea 66 del archivo stdio.c.

void puts (char * *s*)

Función para imprimir una cadena de caracteres.

Esta rutina no valida caracteres especiales.

Parámetros:

s	Cadena terminada en nulo que se desea imprimir

Esta rutina valida caracteres especiales, como fin de línea, tabulador y backspace.

Parámetros:

s	Cadena terminada en nulo que se desea imprimir

Definición en la línea 124 del archivo stdio.c.

Documentación de las variables

int screen_columns

Variable que controla el número de columnas de la pantalla.

Definición en la línea 40 del archivo stdio.c.

int screen_lines

Variable que controla el número de lineas de la pantalla.

Definición en la línea 37 del archivo stdio.c.

char text_attributes

Byte que almacena los atributos de texto.

Definición en la línea 34 del archivo stdio.c.

unsigned short* videoptr

Apuntador al inicio de la memoria de video.

La memoria de video se encuentra mapeada en la dirección lineal 0xB8000. Cada caracter en pantalla ocupa dos caracteres (bytes) en la memoria de video:

- El byte menos significativo contiene el caracter ASCII a mostrar
- El byte más significativo contiene los atributos de texto y fondo del caracter a mostrar. A su vez este byte se subdivide en:

```
*    7  6  5  4  3  2  1  0
*   +-----------------------+
*   |I |F |F |F |I |B |B |B |
*   +-----------------------+
```

- * Los bits F correspondel al color del texto (Foreground). Los bits B corresponden al color de fondo (Background). El bit I corresponde a la intensidad del color de fondo (0 = oscuro, 1 = claro) o del color del texto.

Definición en la línea 31 del archivo stdio.c.

Referencia del Archivo include/stdlib.h

Contiene las definiciones de algunas funciones de utilidad.

'defines'

- #define test_bit(x, n) (x & (1 << n))
  Macro para verificar el valor de un bit especifico de una variable.
- #define set_bit(x, n) x |= (1 << n)
  Macro para establecer en 1 el valor de un bit de la variable.
- #define clear_bit(x, n) x &= ~(1<<n)
  Macro para establecer en 0 el valor de un bit de la variable.

Funciones

- char * itoa (unsigned int n, char *buf, int base)
  Convierte un numero en base 2, 10 0 16 a un string terminado en nulo. Si la base es 10, toma el numero con signo.
- int atoi (char *buf, int base)
  Convierte un string a un entero, en la base especificada.

Descripción detallada

Contiene las definiciones de algunas funciones de utilidad.

Autor:
Erwin Meza emezav@gmail.com

Copyright:
GNU Public License.

Definición en el archivo stdlib.h.

Documentación de los 'defines'

#define clear_bit(x, n) x &= ~(1<<n)

Macro para establecer en 0 el valor de un bit de la variable.

Parámetros:

x	es la variable a verificar
n	es la posicion de el bit en la variable x a limpiar

Permite establecer mediante una operacion logica a nivel de bit, que a al parametro de entrada x, en la posicion del bit n, establecer el valor de 0

Definición en la línea 45 del archivo stdlib.h.

#define set_bit(x, n) x |= (1 << n)

Macro para establecer en 1 el valor de un bit de la variable.

Parámetros:

x	es la variable a modificar un bit
n	es la posicion de el bit en la variable x a establecer en 1

Permite establecer mediante una operacion logica a nivel de bit, que a al parametro de entrada x, en la posicion del bit n, establecer el valor de 1

Definición en la línea 33 del archivo stdlib.h.

#define test_bit(x, n) (x & (1 << n))

Macro para verificar el valor de un bit especifico de una variable.

Parámetros:

x	es la variable a verificar
n	es la posicion de el bit en la variable x a verificar

Basicamente lo que realiza es una operacion logica, en donde checkea si el parametro x en la posicion a nivel de bits, es o no es un 1.

Definición en la línea 21 del archivo stdlib.h.

Documentación de las funciones

int atoi (char * *buf*, int *base*)

Convierte un string a un entero, en la base especificada.

Parámetros:

buf	Buffer que contiene el numero
base	Base en la cual se quiere transformar el numero

Devuelve:

Número en la base especificada.

Definición en la línea 149 del archivo stdlib.c.

char * itoa (unsigned int *n*, char * *buf*, int *base*)

Convierte un numero en base 2, 10 0 16 a un string terminado en nulo. Si la base es 10, toma el numero con signo.

Convierte un numero sin signo en base 2, 10 0 16 a un string terminado en nulo. Si la base es 10, toma el numero con signo.

Parámetros:

n	Número a transformar en cadena

buf	Buffer que contiene el número transformado a cadena de caracteres
base	Base a la cual se desea transformar el número (2, 10 o 16).

Devuelve:

Apuntador al buffer en el cual se encuentra el número transformado

Definición en la línea 20 del archivo stdlib.c.

Referencia del Archivo README.dox

Referencia del Archivo src/exception.c

Este archivo implementa las primitivas necesarias para el manejo de excepciones en la arquitectura IA-32.
`#include <exception.h>`

Funciones

- void exception_dispatcher ()
  Esta rutina recibe el control de la interrupt_dispatcher. Su trabajo consiste en determinar el vector de interrupcion a partir del estado que recibe como parametro, y de invocar la rutina de manejo de excepcion adecuada, si existe.
- void setup_exceptions (void)
  Esta rutina crea un manejador de interrupción para las 32 excepciones de IA-32 e inicializa la tabla de manejadores de excepción. Para todas las excepciones se establece la rutina 'exception_dispatcher' como la rutina de manejo de interrupción. Esta rutina se encarga de "despachar" a un manejador de la excepción, configurado para cada una de ellas mediante la función install_exception_handler().
- int install_exception_handler (unsigned char index, exception_handler handler)
  Esta rutina permite definir un nuevo manejador de excepcion para una de las excepciones de los procesadores x86.
- void uninstall_exception_handler (unsigned char index)
  Esta rutina permite quitar un manejador de excepcion.

Variables

- exception_handler exception_handlers [MAX_EXCEPTIONS]
- unsigned char * exceptions []
  Excepciones del procesador IA-32.

Descripción detallada

Este archivo implementa las primitivas necesarias para el manejo de excepciones en la arquitectura IA-32.

Autor:
Erwin Meza emezav@gmail.com

Copyright:
GNU Public License.

Definición en el archivo exception.c.

Documentación de las funciones

void exception_dispatcher ()

Esta rutina recibe el control de la interrupt_dispatcher. Su trabajo consiste en determinar el vector de interrupcion a partir del estado que recibe como parametro, y de invocar la rutina de manejo de excepcion adecuada, si existe.

Esta rutina recibe el control de la interrupt_dispatcher. Su trabajo consiste en determinar el vector de interrupcion a partir del contexto actual de interrupcion, y de invocar la rutina de manejo de excepcion adecuada, si existe.

Definición en la línea 93 del archivo exception.c.

int install_exception_handler (unsigned char *index*, exception_handler *handler*)

Esta rutina permite definir un nuevo manejador de excepcion para una de las excepciones de los procesadores x86.

Parámetros:

index	Número de la excepción a la cual se le desea instalar la rutina de manejo
handler	Función de manejo de la excepción

Definición en la línea 136 del archivo exception.c.

void setup_exceptions (void)

Esta rutina crea un manejador de interrupción para las 32 excepciones de IA-32 e inicializa la tabla de manejadores de excepción. Para todas las excepciones se establece la rutina 'exception_dispatcher' como la rutina de manejo de interrupción. Esta rutina se encarga de "despachar" a un manejador de la excepción, configurado para cada una de ellas mediante la función install_exception_handler().

Esta rutina crea un manejador de interrupcion para las 32 excepciones x86 e inicializa la tabla de manejadores de excepcion.

Definición en la línea 71 del archivo exception.c.

void uninstall_exception_handler (unsigned char *index*)

Esta rutina permite quitar un manejador de excepcion.

Parámetros:

index	Número de la excepción a la cual se desea desinstalar su manejador

Definición en la línea 150 del archivo exception.c.

Documentación de las variables

exception_handler exception_handlers[MAX_EXCEPTIONS]

Estructura de datos para almacenar las rutinas que manejaran las excepciones

Definición en la línea 16 del archivo exception.c.

unsigned char* exceptions[]

Excepciones del procesador IA-32.

Definición en la línea 27 del archivo exception.c.

Referencia del Archivo src/idt.c

Este archivo implementa las primitivas para el manejo de interrupciones en la arquitectura IA-32.

```
#include <idt.h>
#include <stdlib.h>
#include <stdio.h>
#include <asm.h>
#include <pm.h>
```

Funciones

- idt_descriptor idt[MAX_IDT_ENTRIES] __attribute__ ((aligned(8)))
  Tabla de descriptores de interrupción (IDT)
- void setup_idt (void)
  Esta rutina se encarga de cargar la IDT.
- void install_interrupt_handler (unsigned char index, interrupt_handler handler)
  Instala un nuevo manejador de interrupción para un número de interrupción determinado.
- void uninstall_interrupt_handler (unsigned char index)
  Desinstala un manejador de interrupción.
- void interrupt_dispatcher ()
  Esta rutina recibe el control de la Rutina de Servicio de Interrupción (ISR) isr0, isr1.. etc. correspondiente. Su trabajo consiste en determinar el vector de interrupción a partir del estado que recibe como parametro, y de invocar la rutina de manejo de interrupción adecuada, si existe.

Variables

- idt_ptr idt_pointer
  El apuntador al IDT que se utiliza en la instruccion lidt.
- interrupt_handler interrupt_handlers [MAX_IDT_ENTRIES]
  Arreglo que almacena las referencias a las rutinas de manejo de interrupción. La rutina install_interrupt_handler almacena las referencias a las rutinas en este arreglo.
- unsigned short kernel_code_selector
  Referencia al selector de segmento de codigo del kernel.

Descripción detallada

Este archivo implementa las primitivas para el manejo de interrupciones en la arquitectura IA-32.

Autor:

Erwin Meza emezav@gmail.com

Copyright:

GNU Public License.

Definición en el archivo idt.c.

Documentación de las funciones

idt_descriptor idt [MAX_IDT_ENTRIES] __attribute__ ((aligned(8)))

Tabla de descriptores de interrupción (IDT)

void install_interrupt_handler (unsigned char *index*, interrupt_handler *handler*)

Instala un nuevo manejador de interrupción para un número de interrupción determinado.

Parámetros:

index	Número de interrupción para la cual se desea instalar el manejador
handler	Función para el manejo de la interrupción.

Definición en la línea 111 del archivo idt.c.

void interrupt_dispatcher ()

Esta rutina recibe el control de la Rutina de Servicio de Interrupción (ISR) isr0, isr1.. etc. correspondiente. Su trabajo consiste en determinar el vector de interrupción a partir del estado que recibe como parametro, y de invocar la rutina de manejo de interrupción adecuada, si existe.

Definición en la línea 138 del archivo idt.c.

void setup_idt (void)

Esta rutina se encarga de cargar la IDT.

Definición en la línea 86 del archivo idt.c.

void uninstall_interrupt_handler (unsigned char *index*)

Desinstala un manejador de interrupción.

Parámetros:

index	Número de la interrupción para la cual se va a desinstalar el manejador

Definición en la línea 124 del archivo idt.c.

Documentación de las variables

idt_ptr idt_pointer

El apuntador al IDT que se utiliza en la instruccion lidt.

Definición en la línea 20 del archivo idt.c.

interrupt_handler interrupt_handlers[MAX_IDT_ENTRIES]

Arreglo que almacena las referencias a las rutinas de manejo de interrupción. La rutina install_interrupt_handler almacena las referencias a las rutinas en este arreglo.

Definición en la línea 27 del archivo idt.c.

unsigned short kernel_code_selector

Referencia al selector de segmento de codigo del kernel.

Definición en la línea 23 del archivo pm.c.

Referencia del Archivo src/irq.c

Contiene la implementacion de las rutinas necesarias para manejar las solicitudes de interrupcion (IRQ) de los dispositivos de entrada / salida.

```
#include <idt.h>
#include <irq.h>
#include <stdio.h>
#include <stdlib.h>
```

Funciones

- void irq_dispatcher (interrupt_state *state)
  Esta rutina recibe el control de la rutina de manejo de interrupcion y canaliza esta solicitud a la rutina de manejo de IRQ correspondiente, si se encuentra definida. Dentro de la estructura de datos que recibe, se puede obtener el numero de la interrupcion que ocurrio, asi como el estado del procesador. Con el numero de la interrupcion y restando IDT_IRQ_OFFSET se puede obtener el numero de IRQ.
- void irq_remap (void)
  Función que se encarga de re-mapear las IRQ 0x8 a 0xF.
- void setup_irq (void)
  Esta rutina se encarga de crear los manejadores de interrupcion para las 16 IRQ en los procesadores x86. Estas IRQ se re-mapean a las interrupciones con vector 32 .. 47. Todas ellas son manejadas por la rutina 'irq_dispatcher', que se encarga de invocar la rutina de manejo de IRQ correspondiente, si se encuentra definida.
- void install_irq_handler (int number, irq_handler handler)
  Esta rutina permite definir un nuevo manejador de IRQ.
- void uninstall_irq_handler (int number)
  Esta rutina permite quitar un manejador de IRQ.

Variables

- irq_handler irq_handlers [MAX_IRQ_ROUTINES]
  Arreglo que contiene los apuntadores a las rutinas de manejo de interrupción.

Descripción detallada

Contiene la implementacion de las rutinas necesarias para manejar las solicitudes de interrupcion (IRQ) de los dispositivos de entrada / salida.

Autor:
Erwin Meza emezav@gmail.com

Copyright:
GNU Public License.

Definición en el archivo irq.c.

Documentación de las funciones

void install_irq_handler (int *number*, irq_handler *handler*)

Esta rutina permite definir un nuevo manejador de IRQ.

Parámetros:

number	numero de irq a configurar
handler	Función a manejar la irq

Definición en la línea 150 del archivo irq.c.

void irq_dispatcher (interrupt_state * *state*)

Esta rutina recibe el control de la rutina de manejo de interrupcion y canaliza esta solicitud a la rutina de manejo de IRQ correspondiente, si se encuentra definida. Dentro de la estructura de datos que recibe, se puede obtener el numero de la interrupcion que ocurrio, asi como el estado del procesador. Con el numero de la interrupcion y restando IDT_IRQ_OFFSET se puede obtener el numero de IRQ.

Parámetros:

state	Apuntador al estado del procesador cuando ocurre la IRQ

Definición en la línea 181 del archivo irq.c.

void irq_remap (void)

Función que se encarga de re-mapear las IRQ 0x8 a 0xF.

Al arranque, las IRQ 0 a 7 estan mapeadas a las interrupciones 0x8 - 0xF. Estas entradas estan asignadas por defecto a las excepciones x86 (la entrada 8 es Double Fault, por ejemplo). Por esta razon es necesario mapear las IRQ a otros numeros de interrupcion.

Reprogramación del PIC

```
Initialization Command Word 1 - ICW1
```

Esta es la palabra primaria para inicializar el PIC. Para inicializar el PIC se requiere que los bits 0 y 4 de ICW1 esten en 1 y los demas en 0. Esto significa que el valor de ICW1 es 0x11. ICW1 debe ser escrita en el registro de comandos del PIC maestro (dirección de e/s 0x20). Si existe un PIC esclavo, ICW1 se debe enviar tambien su registro de comandos del PIC esclavo (0xA0)

Initialization Command Word 2 - ICW2 Esta palabra permite definir la dirección base (inicial) en la tabla de descriptores de interrupcion que el PIC va a utilizar.

Debido a que las primeras 32 entradas estan reservadas para las excepciones en la arquitectura IA-32, ICW2 debe contener un valor mayor o igual a 32 (0x20). Los valores de ICW2 representan el numero de IRQ base que manejara el PIC

Al utilizar los PIC en cascada, se debe enviar ICW2 a los dos controladores en su registro de datos (0x21 y 0xA1 para maestro y esclavo respectivamente), indicando la dirección en la IDT que va a ser utilizada por cada uno de ellos.

Las primeras 8 IRQ van a ser manejadas por el PIC maestro y se mapearan a partir del numero 32 (0x20). Las siguientes 8 interrupciones las manejara el PIC esclavo, y se mapearan a partir de la interrupcion 40 (0x28).

Initialization Control Word 3 - ICW3 Esta palabra permite definir cuales lineas de IRQ van a ser compartidas por los PIC maestro y esclavo. Al igual que ICW2, ICW3 tambien se escribe en los registros de datos de los PIC (0x21 y 0xA1 para el PIC maestro y esclavo,respectivamente).

Dado que en la arquitectura Intel el PIC maestro se conecta con el PIC esclavo por medio de la linea IRQ 2, el valor de ICW3 debe ser 00000100 (0x04), que define el bit 3 (correspondiente a la linea IRQ2) en 1.

Para el PIC esclavo, el numero de la linea se debe representar en notacion binaria. Por lo tanto, 000 corresponde a la linea de IRQ 0, 001 corresponde a la linea de IRQ 1, 010 corresponde a la linea de IRQ 2, y asi sucesivamente. Debido a que se va a utilizar la linea de IRQ 2, el valor de ICW3 para el PIC esclavo debe ser 00000010, (0x02).

Initialization Control Word 4 - ICW4 Para ICW4 solo es necesario establecer su bit 0 (x86 mode) y escribirla en los registros de datos del PIC maestro y esclavo (0x21 y 0xA1). El valor de ICW4 debe ser entonces 00000001, es decir, 0x01.

Se han mapeado las IRQ!. Las IRQ 0-7 seran atendidas por el PIC maestro, y las IRQ 8-15 por el PIC esclavo. Las IRQ0-15 estaran mapeadas en la IDT a partir de la entrada 32 hasta la 47.

Definición en la línea 46 del archivo irq.c.

void setup_irq (void)

Esta rutina se encarga de crear los manejadores de interrupcion para las 16 IRQ en los procesadores x86. Estas IRQ se re-mapean a las interrupciones con vector 32 .. 47. Todas ellas son manejadas por la rutina 'irq_dispatcher', que se encarga de invocar la rutina de manejo de IRQ correspondiente, si se encuentra definida.

Esta rutina se encarga de crear las entradas en la IDT para las interrupciones que se desean manejar. Por defecto configura las interrupciones correspondientes a las IRQ de los dispositivos de entradda/ salida, que han sido mapeadas a los numeros 32..47.

Definición en la línea 126 del archivo irq.c.

void uninstall_irq_handler (int *number*)

Esta rutina permite quitar un manejador de IRQ.

Parámetros:

number	numero de irq a quitar

Devuelve:

void

Definición en la línea 164 del archivo irq.c.

Documentación de las variables

irq_handler irq_handlers[MAX_IRQ_ROUTINES]

Arreglo que contiene los apuntadores a las rutinas de manejo de interrupción.
Definición en la línea 20 del archivo irq.c.

Referencia del Archivo src/isr.S

Contiene la definicion y la implementacion de las rutinas de servicio de interrupcion para las 255 interrupciones que se pueden generar en un procesador IA-32. Todas las rutinas establecen un marco de pila uniforme, y luego invocan a la rutina interrupt_dispatcher().

Descripción detallada

Contiene la definicion y la implementacion de las rutinas de servicio de interrupcion para las 255 interrupciones que se pueden generar en un procesador IA-32. Todas las rutinas establecen un marco de pila uniforme, y luego invocan a la rutina interrupt_dispatcher().

Autor:

Erwin Meza emezav@gmail.com

Copyright:

GNU Public License.

Definición en el archivo isr.S.

Referencia del Archivo src/kernel.c

Código de inicialización del kernel en C. Este código recibe el control de start.S y continúa con la ejecución.

```
#include <pm.h>
#include <multiboot.h>
#include <stdio.h>
#include <stdlib.h>
#include <idt.h>
#include <physmem.h>
```

Funciones

- void cmain (unsigned int magic, void *multiboot_info)
  Función principal del kernel. Esta rutina recibe el control del codigo en ensamblador de start.S.

Variables

- unsigned int multiboot_info_location
  Variable global del kernel que almacena la localizacion de la estructura multiboot que recibe del GRUB.

Descripción detallada

Código de inicialización del kernel en C. Este código recibe el control de start.S y continúa con la ejecución.

Autor:

Erwin Meza emezav@gmail.com

Copyright:

GNU Public License.

Luego de configurar la GDT, la IDT, las interrupciones, las excepciones y las IRQ, este código configura el mapa de bits que permitirá gestionar la memoria física en unidades de 4096 bytes. Este mapa de bits se referencia con la variable memory_bitmap (physmem.c), la cual apunta a la dirección MMAP_LOCATION.

El mapa de bits de memoria ocupa exactamente 128 KB, y permite gestionar hasta 4 GB de memoria física.

Definición en el archivo kernel.c.

Documentación de las funciones

void cmain (unsigned int *magic*, void * *multiboot_info*)

Función principal del kernel. Esta rutina recibe el control del codigo en ensamblador de start.S.

Parámetros:

magic	Número mágico pasado por GRUB al código de start,S

multiboot_info	Apuntador a la estructura de información multiboot

Definición en la línea 40 del archivo kernel.c.

Documentación de las variables

unsigned int multiboot_info_location

Variable global del kernel que almacena la localizacion de la estructura multiboot que recibe del GRUB.

Definición en la línea 31 del archivo kernel.c.

Referencia del Archivo src/physmem.c

Contiene la implementación de las rutinas relacionadas con las gestión de memoria física. La memoria se gestiona en unidades de 4096 bytes.

```
#include <physmem.h>
#include <multiboot.h>
#include <stdio.h>
#include <stdlib.h>
```

Funciones

- void setup_memory (void)
  Esta rutina inicializa el mapa de bits de memoria, a partir de la informacion obtenida del GRUB.
- char * allocate_unit (void)
  Busca una unidad libre dentro del mapa de bits de memoria.
- char * allocate_unit_region (unsigned int length)
  Busca una región de memoria contigua libre dentro del mapa de bits de memoria.
- void free_unit (char *addr)
  Permite liberar una unidad de memoria.
- void free_region (char *start_addr, unsigned int length)
  Permite liberar una región de memoria.

Variables

- unsigned int * memory_bitmap = (unsigned int*) MMAP_LOCATION
  Mapa de bits de memoria disponible.
- unsigned int next_free_unit
  Siguiente unidad disponible en el mapa de bits.
- int free_units
  Numero de marcos libres en la memoria.
- int total_units
  Numero total de unidades en la memoria.
- unsigned int base_unit
  Marco inicial de las unidades disponibles en memoria.
- unsigned int memory_bitmap_length
  Tamaño del mapa de bits en memoria.
- unsigned int memory_start
  Variable global del kernel que almacena el inicio de la región de memoria disponible.
- unsigned int memory_length
  Variable global del kernel que almacena el tamaño en bytes de la memoria disponible.
- unsigned int allowed_free_start
  Mínima dirección de memoria permitida para liberar, a esta se le debera asignar la direccion lineal en donde termina el KERNEL.

Descripción detallada

Contiene la implementación de las rutinas relacionadas con las gestión de memoria física. La memoria se gestiona en unidades de 4096 bytes.

Autor:

Erwin Meza emezav@gmail.com

Copyright:

GNU Public License.

Definición en el archivo physmem.c.

Documentación de las funciones

char* allocate_unit (void)

Busca una unidad libre dentro del mapa de bits de memoria.

Devuelve:

Dirección de inicio de la unidad en memoria.

```
                Define las unidades disponibles de memoria
```

que dispone

Definición en la línea 288 del archivo physmem.c.

char* allocate_unit_region (unsigned int *length*)

Busca una región de memoria contigua libre dentro del mapa de bits de memoria.

Parámetros:

length	Tamaño de la región de memoria a asignar.

Devuelve:

Dirección de inicio de la región en memoria.

El proceso que realiza esta funcion es,

```
1.Verificar si no existen unidades libres, de ser asi, el valor de
  retorno es 0
2.Luego si existen unidades libres, entonces, se inicia una iteracion
  en busqueda de una unidad de memoria disponible.
3.De no existir una unidad libre dentro del mapa de bits, entonces
  devuelve 0; en caso de que si, devuelve la direccion de inicio
  de la region de memoria multiplicando las unidades libres
  de memoria por el tamaño de asignacin de una unidad de memoria.
```

Definición en la línea 332 del archivo physmem.c.

void free_region (char * *start_addr*, unsigned int *length*)

Permite liberar una región de memoria.

Parámetros:

start_addr	Dirección de memoria del inicio de la región a liberar
length	Tamaño de la región a liberar

Para permitir la liberacion de una region de memoria, se siguen estos pasos;

```
1.Si la direccion que se recibe es menor a la unidad minima permitida de
 asignacion de memoria, significa que se quiere liberar memoria que no
 esta disponible, ya que esta usada por el KERNEL o los modulos que se
 cargan con el KERNEL, en este caso termina sin realizar la peticion.
2.Se convierte a una direccion lineal para poner en el mapa de bits como
 disponible mediante un ciclo permitiendo liberar toda la region de memoria
 en unidades de memoria disponibles.
3.Se marca esa unidad liberada, como la proxima unidad para asignar, y aumenta
 el numero de unidades libres.
```

Definición en la línea 417 del archivo physmem.c.

void free_unit (char * *addr*)

Permite liberar una unidad de memoria.

Parámetros:

addr	Dirección de memoria dentro del área a liberar.

Para permitir la liberacion de una unidad de memoria, se define la secuencia de pasos como:

```
1.Si la direccion que se recibe es menor a la unidad minima permitida de
 asignacion de memoria, significa que se quiere liberar memoria que no
 esta disponible, ya que esta usada por el KERNEL o los modulos que se
 cargan con el KERNEL, en este caso termina sin realizar la peticion.
2.Se convierte a una direccion lineal para ponerla en el mapa de bits como
 disponible.
3.Se marca esa unidad liberada, como la proxima unidad para asignar, y aumenta
 el numero de unidades libres.
```

Definición en la línea 392 del archivo physmem.c.

void setup_memory (void)

Esta rutina inicializa el mapa de bits de memoria, a partir de la informacion obtenida del GRUB.

Basicamente lo que se hace es configurar el mapa de bits de la memoria, para ello la secuencia de pasos que realiza dicha configuracion es:

```
  1.Cargar una estructura de tipo multiboot_info_t con la
   informacion del multiboot que es la informacion que el GRUB
   almacena luego de cargar el KERNEL.
  2.Limpia el mapa de bits, haciendo memory_bitmap[i] = 0,
      donde i=1,2,3.... (memory bitmap length-1)
  3.Verifica si el GRUB cargo la informacion solicitada en la
   estructura del multiboot. En el proyecto actual, se omite
   esta parte.
  4.Se establece, la minima direccion de memoria permitida para
   liberar, a esta se le asignara la direccion lineal de donde
   termina el KERNEL.
  5.Verifica si la direccion del mapa de bits son validos, para ello
   verifica que el bit info->flags[6] sea 1, lo que sifnifica que  mmap_length
   y mmap_addr son validos .
```

```
  6.Verifica que la region de memoria cumple con las condiciones
   para ser considerada como Memoria Disponible.
   Para ello, se debe considerar que:
   ->La primera unidad de memoria este ubicada en la posición de
   memoria mayor o igual que 1 MB.
   ->nmap->type que es el tipo de area de memoria si es 1.
   6,1.Lo que significa disponible, de lo contrario es que ya esta reservada.
    Si la region esta marcada como disponible y la posicion esta por
    encima de la direccion  de la posicion del KERNEL en memora,
    se procede a verificar si el KERNEL se encuentre en esa region.
    Entonces se toma el inicio de la memoria disponible en la posicion
    en la cual finaliza el KERNEL.
    Seguido a esto, se verifica si se cargaron los modulos con el KERNEL
    En caso de que hayan modulos que se carguen con el KERNEL, entonces
    la direccion de posicion inicial cambian ahora y empezadia desde luego
    de los modulos cargado; de lo contrario no se afecta.
    y por ultimo se resta el espacio que ahora dispone.

   6,2.Si la region ya esta reservada, y/o que la primera unidad de memoria
       es menor que 1 MB, lo que significa que el KERNEL no se encuentra en
       esta region, entonces el tamaño es mayor que la region encontrada.

7.Si existe una region de memoria disponible, es decir si la direccion donde
 inicia la primera unidad de memoria es valida, y el tamaño disponible es
 valido.
 Se procede a establecer en que direccion de memoria se debe ubicar para
 la asignacion de memoria permitida a liberar, para esto, se calcula la
 direccion en la cual finaliza la memoria disponibe, hace unos ajustes de
 redondeo, se calcula el tamaño de region disponible con las unidades ya
 redondeadas, y se procede a establecer las variables globales del KERNEL.
8.Se marca la direccion de memoria como disponibley la direccion de memoria
```

de la cual se puede liberar memoria.

Existe un mapa de memoria válido creado por GRUB?

Si verifica que el bit en la posición 6 es 1, entonces se crea un mapa valido de memoria, es decir que mmap_length y mmap_addr son validos

Definición en la línea 67 del archivo physmem.c.

Documentación de las variables

unsigned int allowed_free_start

Mínima dirección de memoria permitida para liberar, a esta se le debera asignar la direccion lineal en donde termina el KERNEL.

Definición en la línea 61 del archivo physmem.c.

unsigned int base_unit

Marco inicial de las unidades disponibles en memoria.

Definición en la línea 34 del archivo physmem.c.

int free_units

Numero de marcos libres en la memoria.

Definición en la línea 28 del archivo physmem.c.

unsigned int* memory_bitmap = (unsigned int*) MMAP_LOCATION

Mapa de bits de memoria disponible.

Esta variable almacena el apuntador del inicio del mapa de bits que permite gestionar las unidades de memoria. En este caso, la direccion lineal en donde esta ubicado el bitmap es 0x500 e

Definición en la línea 22 del archivo physmem.c.

unsigned int memory_bitmap_length

```
Valor inicial:= ~(0x0)
    / (MEMORY_UNIT_SIZE * BITS_PER_ENTRY)
```

Tamaño del mapa de bits en memoria.

Para un espacio fisico de maximo 4 GB, se requiere un mapa de bits de 128 KB. Si cada entrada ocupa 4 bytes, se requiere 32678 entradas. En la operacion ~(0x0)/(MEMORY_UNIT_SIZE * BITS_PER_ENTRY) lo que hace es averiguar que tamaño tiene el mapa de bits, en donde se divide el máximo numero que se puede guardar en un registro (2^32 -1) entre el tamaño de la unidad de memoria, lo que nos da el numero de unidades de memoria que tenemos en un espacio lineal, y a su vez se divide entre los bits que tiene cada entrada en la tabla, lo que da el total de elementos que tiene la tabla.

Definición en la línea 47 del archivo physmem.c.

unsigned int memory_length

Variable global del kernel que almacena el tamaño en bytes de la memoria disponible.

Definición en la línea 55 del archivo physmem.c.

unsigned int memory_start

Variable global del kernel que almacena el inicio de la región de memoria disponible.

Definición en la línea 52 del archivo physmem.c.

unsigned int next_free_unit

Siguiente unidad disponible en el mapa de bits.

Definición en la línea 25 del archivo physmem.c.

int total_units

Numero total de unidades en la memoria.

Definición en la línea 31 del archivo physmem.c.

Referencia del Archivo src/pm.c

Contiene la implementación de las rutinas para la gestión del modo protegido IA-32 y de la Tabla Global de Descriptores (GDT)

```
#include <pm.h>
#include <stdio.h>
```

Funciones

- gdt_descriptor gdt[MAX_GDT_ENTRIES] __attribute ((aligned(8)))
  Tabla Global de Descriptores (GDT). Es un arreglo de descriptores de segmento. Según el manual de Intel, esta tabla debe estar alineada a un límite de 8 bytes para un óptimo desempeño.
- unsigned short get_gdt_selector (gdt_descriptor *desc)
  Función que permite obtener el selector en la GDT a partir de un apuntador a un descriptor de segmento.
- gdt_descriptor * get_gdt_descriptor (unsigned short selector)
  Función que permite obtener el descriptor de segmento en la GDT a partir de un selector.
- gdt_descriptor * allocate_gdt_descriptor (void)
  Esta rutina permite obtener un descriptor de segmento disponible en la GDT.
- unsigned short allocate_gdt_selector (void)
  Buscar un selector disponible dentro de la GDT.
- void free_gdt_descriptor (gdt_descriptor *desc)
  Esta rutina permite liberar un descriptor de segmento en la GDT.
- void setup_gdt_descriptor (unsigned short selector, unsigned int base, unsigned int limit, char type, char dpl, int code_or_data, char opsize)
  Permite configurar un descriptor de segmento dentro de la GDT.
- void setup_gdt (void)
  Esta función se encarga de cargar la GDT.

Variables

- int current_gdt_entry = 0
  Variable que almacena la siguiente entrada disponible en la GDT.
- unsigned short kernel_code_selector
  Variable que almacena el selector del descriptor de segmento de código del kernel.
- gdt_descriptor * kernel_code_descriptor
  Referencia al Descriptor de segmento de codigo del kernel dentro de la GDT.
- unsigned short kernel_data_selector
  Variable que almacena el selector del descriptor de segmento de datos para el kernel.
- gdt_descriptor * kernel_data_descriptor
  Referencia al Descriptor de segmento de datos del kernel.
- gdt_ptr gdt_pointer
  Apuntador a la GDT usado por la instrucción lgdt para cargar la GDT.

Descripción detallada

Contiene la implementación de las rutinas para la gestión del modo protegido IA-32 y de la Tabla Global de Descriptores (GDT)

Autor:

Erwin Meza emezav@gmail.com

Copyright:

GNU Public License.

Definición en el archivo pm.c.

Documentación de las funciones

gdt_descriptor gdt [MAX_GDT_ENTRIES] __attribute ((aligned(8)))

Tabla Global de Descriptores (GDT). Es un arreglo de descriptores de segmento. Según el manual de Intel, esta tabla debe estar alineada a un límite de 8 bytes para un óptimo desempeño.

gdt_descriptor* allocate_gdt_descriptor (void)

Esta rutina permite obtener un descriptor de segmento disponible en la GDT.

Devuelve:

Referencia al próximo descriptor de segmento dentro de la GDT que se encuentra disponible, nulo en caso que no exista una entrada disponible dentro de la GDT.

Definición en la línea 81 del archivo pm.c.

unsigned short allocate_gdt_selector (void)

Buscar un selector disponible dentro de la GDT.

Esta rutina permite obtener un descriptor de segmento disponible en la GDT.

Devuelve:

Selector que apunta al descriptor de segmento disponible encontrado dentro de la GDT

Definición en la línea 108 del archivo pm.c.

void free_gdt_descriptor (gdt_descriptor * *desc*)

Esta rutina permite liberar un descriptor de segmento en la GDT.

Parámetros:

desc	Apuntador al descriptor que se desa liberar

Definición en la línea 118 del archivo pm.c.

gdt_descriptor* get_gdt_descriptor (unsigned short *selector*)

Función que permite obtener el descriptor de segmento en la GDT a partir de un selector.

Parámetros:

selector	Selector que permite obtener un descriptor de segmento de la GDT

Devuelve:

Referencia al descriptor de segmento dentro de la GDT

Definición en la línea 65 del archivo pm.c.

unsigned short get_gdt_selector (gdt_descriptor * *desc*)

Función que permite obtener el selector en la GDT a partir de un apuntador a un descriptor de segmento.

Parámetros:

desc	Referencia al descriptor de segmento del cual se desea obtener el selector

Devuelve:

Selector que referencia al descriptor dentro de la GDT.

Definición en la línea 47 del archivo pm.c.

void setup_gdt (void)

Esta función se encarga de cargar la GDT.

Esta función se encarga de configurar la Tabla Global de Descriptores (GDT) que usará el kernel. La GDT es un arreglo de descriptores de segmento, cada uno de los cuales contiene la información de los segmentos en memoria. Esta rutina realiza los siguientes pasos:

- Buscar el espacio en la GDT para el descriptor de segmento de código del kernel
- Verificar el offset dentro de la GDT. Debe ser igual a la constante KERNEL_CODE_SELECTOR Definida en pm.h
- Buscar espacio en la GDT para el descriptor de segmento de datos del kernel
- Verificar el offset dentro de la GDT. Debe ser igual a la constante KERNEL_DATA_SELECTOR definida en pm.h
- Definir el segmento de código del kernel como un segmento plano (base = 0, límite = 4 GB, nivel de privilegios 0). Para describir este segmento se usará la segunda entrada de la GDT. El manual de Intel especifica que la primera entrada de la GDT siempre debe ser nula.
- Definir el segmento de datos del kernel como un segmento plano (base = 0, límite = 4 GB, nivel de privilegios 0). Para describir este segmento se usará la tercera entrada de la GDT.
- La instrucción LGDT recibe un apuntador que tiene dos atributos: Tamaño del GDT - 1 y dirección lineal de la GDT en memoria.

- Ejecutar el siguiente código en ensamblador, para cargar la GDT y pasar "de nuevo" a modo protegido.

```
*/
inline_assembly(".intel_syntax noprefix\n\t\
                            cli                             \n\t\
                            lgdt [%0]                       \n\t\
                            mov edx, cr0                    \n\t\
                            or edx, 1                       \n\t\
                            mov cr0, edx                    \n\t\
                            push %2                         \n\t\
                            push 0  /* eflags */            \n\t\
                            push %1                         \n\t\
                            push OFFSET 1f                  \n\t\
                            iret                            \n\t\
                    1:                                      \n\t\
                            pop ecx                         \n\t\
                            mov ds, cx                      \n\t\
                            mov es, cx                      \n\t\
                            mov fs, cx                      \n\t\
                            mov gs, cx                      \n\t\
                            mov ss, cx                      \n\t\
                            .att_syntax prefix\n\t"
                            :
                            : "am"(&gdt pointer),
                            "b"(kernel_code_selector & 0x0000FFFF),
                            "c"(kernel_data_selector & 0x0000FFFF)
                            : "dx");
```

/**

Este código en lenguaje ensamblador realiza las siguientes acciones:

1. Deshabilitar las interrupciones
2. Ejecutar la instrucción lgdt, pasando como parámetro un apuntador a la GDT configurada.
3. Activar el bit PE (Protection Enable) en el registro CR0.
4. Simular un retorno de interrupción, para que el registro de segmento CS contenga el valor del selector que referencia al descriptor de segmento de código configurado en la GDT.
5. Actualizar los registros de segmento DS, ES, FS, GS y SS para que contengan el valor del selector que referencia al descriptor de segmento de datos configurado en la GDT.

Observe que básicamente se está pasando de nuevo a modo protegido, esta vez usando la GDT configurada en esta función.

Definición en la línea 178 del archivo pm.c.

void setup_gdt_descriptor (unsigned short *selector*, unsigned int *base*, unsigned int *limit*, char *type*, char *dpl*, int *code_or_data*, char *opsize*)

Permite configurar un descriptor de segmento dentro de la GDT.

Parámetros:

selector	Selector que referencia al descriptor de segmento dentro de la GDT
base	Dirección lineal del inicio del segmento en memoria
limit	Tamaño del segmento
type	Tipo de segmento
dpl	Nivel de privilegios del segmento
code_or_data	1 = Segmento de código o datos, 0 = segmento del sistema
opsize	Tamaño de operandos: 0 = 16 bits, 1 = 32 bits

Definición en la línea 142 del archivo pm.c.

Documentación de las variables

int current_gdt_entry = 0

Variable que almacena la siguiente entrada disponible en la GDT.
Definición en la línea 19 del archivo pm.c.

gdt_ptr gdt_pointer

Apuntador a la GDT usado por la instrucción lgdt para cargar la GDT.
Definición en la línea 38 del archivo pm.c.

gdt_descriptor* kernel_code_descriptor

Referencia al Descriptor de segmento de codigo del kernel dentro de la GDT.
Definición en la línea 27 del archivo pm.c.

unsigned short kernel_code_selector

Variable que almacena el selector del descriptor de segmento de código del kernel.
Referencia al selector de segmento de codigo del kernel.
Definición en la línea 23 del archivo pm.c.

gdt_descriptor* kernel_data_descriptor

Referencia al Descriptor de segmento de datos del kernel.
Definición en la línea 34 del archivo pm.c.

unsigned short kernel_data_selector

Variable que almacena el selector del descriptor de segmento de datos para el kernel.
Definición en la línea 31 del archivo pm.c.

Referencia del Archivo src/start.S

Punto de entrada del kernel.

Descripción detallada

Punto de entrada del kernel.

Autor:

Erwin Meza emezav@gmail.com

Copyright:

GNU Public License.

El kernel se carga en la dirección de memoria 0x100000 (1 MB) y empieza su ejecución en la etiqueta start. Este kernel cumple con la especificación multiboot, por lo cual debe contener un encabezado (estructura de datos) que le indica al cargador de arranque los parametros necesarios para que cargue el kernel. Este kernel es diferente a los anteriores: Se almacena en una imagen de disco duro, que se encuentra formateada con el sistema de archivos ext2 (linux). Para cargarlo, se recurre a un cargador de arranque (bootloader) llamado GRUB. GRUB es un bootloader que cumple con la Especificacion Multiboot. Puede cargar cualquier kernel que cumpla con esta especificación.

Ver también:

http://www.gnu.org/software/grub/manual/multiboot/multiboot.html Especificación Multiboot

Definición en el archivo start.S.

Referencia del Archivo src/stdio.c

Contiene las primitivas basicas para entrada / salida.
`#include <stdio.h>`

Funciones

- void scroll (void)
  Función privada para subir una línea si se ha llegado al final de la pantalla.
- void update_cursor (void)
  Función privada que permite actualizar el cursor en la pantalla.
- void putchar (char c)
  Función para imprimir un caracter.
- void puts (char *s)
  Función para imprimir una cadena de caracteres.
- void cls (void)
  Función para limpiar la pantalla.
- void printf (char *format,...)
  Esa funcion implementa en forma basica el comportamiento de 'printf' en C.

Variables

- unsigned short * videoptr = (unsigned short *) VIDEO_ADDR
  Apuntador al inicio de la memoria de video.
- char text_attributes = COLOR(LIGHTGRAY, BLACK)
  Byte que almacena los atributos de texto.
- int screen_lines = SCREEN_LINES
  Variable que controla el número de líneas de la pantalla.
- int screen_columns = SCREEN_COLUMNS
  Variable que controla el número de columnas de la pantalla.
- int current_line = 0
  Variable que controla la línea actual en la pantalla.
- int current_column = 0
  Variable que controla la columna actual en la pantalla.

Descripción detallada

Contiene las primitivas basicas para entrada / salida.

Autor:
Erwin Meza emezav@gmail.com

Copyright:
GNU Public License.

Definición en el archivo stdio.c.

Documentación de las funciones

void cls (void)

Función para limpiar la pantalla.

Definición en la línea 186 del archivo stdio.c.

void printf (char * *format*, ...)

Esa funcion implementa en forma basica el comportamiento de 'printf' en C.

Parámetros:

format	Formato de la cadena de salida
...	Lista de referencias a memoria de las variables a imprimir

Definición en la línea 311 del archivo stdio.c.

void putchar (char *c*)

Función para imprimir un caracter.

Esta rutina imprime directamente en la memoria de video. Valida caracteres especiales, como fin de línea, tabulador y backspace.

Parámetros:

c	caracter ascii a imprimir

Definición en la línea 66 del archivo stdio.c.

void puts (char * *s*)

Función para imprimir una cadena de caracteres.

Esta rutina valida caracteres especiales, como fin de línea, tabulador y backspace.

Parámetros:

s	Cadena terminada en nulo que se desea imprimir

Definición en la línea 124 del archivo stdio.c.

void scroll (void)

Función privada para subir una línea si se ha llegado al final de la pantalla.

Definición en la línea 279 del archivo stdio.c.

void update_cursor (void)

Función privada que permite actualizar el cursor en la pantalla.

Esta rutina posiciona el cursor de hardware en la pantalla, con la posicion actual de escritura en la memoria de video. Para ello, utiliza el microcontrolador CRT, el cual posee dos registros:

- 0x3D4 = registro de indice
- 0x3D5 = registro de datos.

Para escribir en el microcontrolador CRT, se debe escribir dos veces: Primero se debe escribir en el registro de índice, para indicar el tipo de datos que se desea escribir, y luego se escribe en el registro de datos el dato que se desea escribir.

Definición en la línea 228 del archivo stdio.c.

Documentación de las variables

int current_column = 0

Variable que controla la columna actual en la pantalla.

Definición en la línea 46 del archivo stdio.c.

int current_line = 0

Variable que controla la línea actual en la pantalla.

Definición en la línea 43 del archivo stdio.c.

int screen_columns = SCREEN_COLUMNS

Variable que controla el número de columnas de la pantalla.

Definición en la línea 40 del archivo stdio.c.

int screen_lines = SCREEN_LINES

Variable que controla el número de líneas de la pantalla.

Variable que controla el número de lineas de la pantalla.

Definición en la línea 37 del archivo stdio.c.

char text_attributes = COLOR(LIGHTGRAY, BLACK)

Byte que almacena los atributos de texto.

Definición en la línea 34 del archivo stdio.c.

unsigned short* videoptr = (unsigned short *) VIDEO_ADDR

Apuntador al inicio de la memoria de video.

La memoria de video se encuentra mapeada en la dirección lineal 0xB8000. Cada caracter en pantalla ocupa dos caracteres (bytes) en la memoria de video:

- El byte menos significativo contiene el caracter ASCII a mostrar

- El byte más significativo contiene los atributos de texto y fondo del caracter a mostrar. A su vez este byte se subdivide en:

```
*    7  6  5  4  3  2  1  0
*   +-----------------------+
*   |I |F |F |F |I |B |B |B |
*   +-----------------------+
```

- * Los bits F correspondel al color del texto (Foreground). Los bits B corresponden al color de fondo (Background). El bit I corresponde a la intensidad del color de fondo (0 = oscuro, 1 = claro) o del color del texto.

Definición en la línea 31 del archivo stdio.c.

Referencia del Archivo src/stdlib.c

Contiene las implementaciones de algunas funciones de utilidad.
`#include <stdlib.h>`

Funciones

- char * itoa (unsigned int n, char *buf, int base)
  Convierte un numero en base 2, 10 0 16 a un string terminado en nulo. Si la base es 10, toma el numero con signo.
- char * utoa (unsigned int n, char *buf, int base)
  Convierte un numero sin signo en base 2, 10 0 16 a un string terminado en nulo. Si la base es 10, toma el numero con signo.
- int atoi (char *buf, int base)
  Convierte un string a un entero, en la base especificada.

Descripción detallada

Contiene las implementaciones de algunas funciones de utilidad.

Autor:
Erwin Meza emezav@gmail.com

Copyright:
GNU Public License.

Definición en el archivo stdlib.c.

Documentación de las funciones

int atoi (char * *buf*, int *base*)

Convierte un string a un entero, en la base especificada.

Parámetros:

buf	Buffer que contiene el numero
base	Base en la cual se quiere transformar el numero

Devuelve:
Número en la base especificada.

Definición en la línea 149 del archivo stdlib.c.

char* itoa (unsigned int *n*, char * *buf*, int *base*)

Convierte un numero en base 2, 10 0 16 a un string terminado en nulo. Si la base es 10, toma el numero con signo.

Parámetros:

n	Número a transformar en cadena
buf	Buffer que contiene el número transformado a cadena de caracteres
base	Base a la cual se desea transformar el número (2, 10 o 16).

Devuelve:

Apuntador al buffer en el cual se encuentra el número transformado

Definición en la línea 20 del archivo stdlib.c.

char* utoa (unsigned int *n*, char * *buf*, int *base*)

Convierte un numero sin signo en base 2, 10 0 16 a un string terminado en nulo. Si la base es 10, toma el numero con signo.

Parámetros:

n	Número a transformar en cadena
buf	Buffer que contiene el número transformado a cadena de caracteres
base	Base a la cual se desea transformar el número (2, 10 o 16).

Devuelve:

Apuntador al buffer en el cual se encuentra el número transformado

Definición en la línea 91 del archivo stdlib.c.