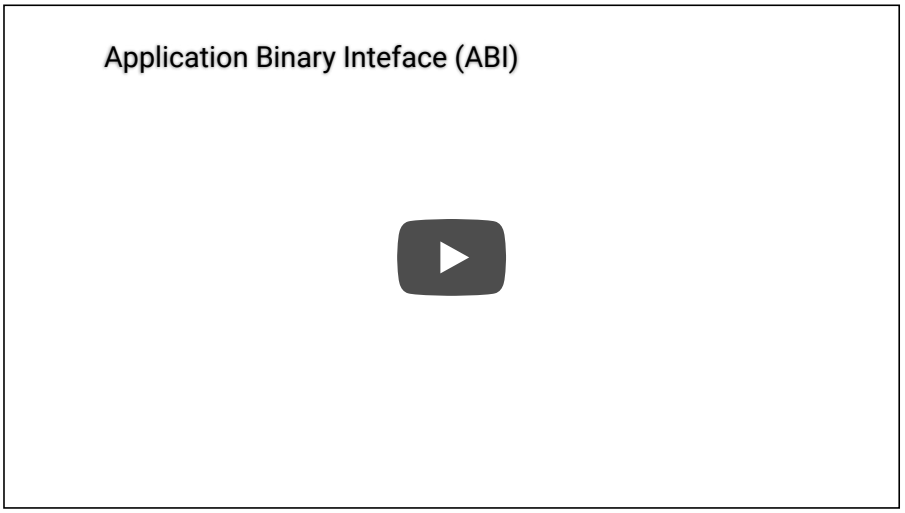
Interfaz entre Assembler y gcc

por Dario Alejandro Alpern



Transcripción

Hola. Mi nombre es Darío Alpern y hoy vamos a ver Application Binary Interface.

Aplication Binary Interface

El Application Binary Interface (ABI) es la interfaz entre dos módulos en binario, o sea ya compilados.

En nuestro caso nos vamos a centrar en cómo escribir programas en Assembler de procesadores Intel x86 para que se puedan utilizar con módulos escritos con C y compilados usando gcc.

Hay que tener en cuenta que la interfaz es diferente para otros compiladores, tales como Visual Studio y para otros procesadores.

Lo que vamos a ver en este video es cómo se pasan los parámetros a una función, cómo se usan variables locales y variables globales. También vamos a ver algunos ejemplos.

Uso de la pila

En este esquema se puede ver que la pila se divide en marcos ("stack frame" en inglés), uno por cada función que se anide. Tanto los argumentos de las funciones como las variables locales residen en la pila.

En ambos diagramas, las direcciones más altas de la pila están arriba, así que a medida que ingresan nuevos datos a la pila, el puntero de pila se mueve hacia abajo. Por eso el marco más nuevo se encuentra abajo.

Aunque en el diagrama de la derecha todos los marcos tienen el mismo tamaño, en realidad no es así porque su tamaño depende de la cantidad de parámetros de la función y la cantidad de bytes asignados para variables locales.

El ABI requiere que se use el registro EBP para apuntar al marco correspondiente a la función que se está ejecutando.

El registro ESP siempre apunta a la dirección más baja del marco. Esto permite que no se sobreescriban los datos que están allí si ocurre alguna interrupción, ya que el procesador usa la pila para almacenar el registro EFLAGS y la dirección lógica de retorno.

El registro EBP apunta al registro EBP correspondiente al marco anterior, es decir, el de la función que llamó al que está corriendo ahora.

Así se enlazan los marcos, como en una lista. Eso permite mostrar la pila de llamadas ("call stack" en inglés) en un debugger. La pila de llamadas muestra los nombres de las funciones anidadas y sus parámetros.

En el diagrama de la izquierda se muestra el contenido de cada marco.

Primero la función llamadora coloca los argumentos en la pila, ordenados de derecha a izquierda. De esta manera el último argumento que ingresa es el primero (el de la izquierda) y por eso es el que figura más abajo.

Luego la función llamadora ejecuta una instrucción CALL para llamar a la función actual. De esta manera, el procesador pone en la pila la dirección efectiva correspondiente a la siguiente instrucción luego de ese CALL.

La función actual comienza salvando el registro EBP, ya que lo tiene que usar para apuntar a este marco y hasta ese momento apuntaba al marco anterior. Eso se logra mediante una instrucción PUSH EBP que pone en la pila el viejo valor de EBP.

A continuación la función actual hace que EBP apunte al marco actual mediante la instrucción MOV EBP,ESP y a partir de ese momento los argumentos se pueden acceder mediante direccionamiento indirecto usando EBP como se muestra en el extremo izquierdo.

Finalmente la función actual reserva el área de variables locales restando al puntero de pila ESP la longitud de esa área.

Partes de una función

En base a lo que vimos recién, podemos dividir el código de una función en tres partes: prólogo, cuerpo y epílogo.

El prólogo es el encargado de terminar de armar el marco de pila con las instrucciones que vimos recién.

El cuerpo es el conjunto de instrucciones que realizan lo que debe hacer la función, utilizando entre otros recursos el marco de pila.

El epílogo es el encargado de destruir el marco de pila haciendo que EBP vuelva a apuntar al marco anterior.

La primera instrucción MOV ESP,EBP libera la zona de variables locales. La segunda instrucción POP EBP hace que el registro EBP apunte al marco de la función llamadora y finalmente la instrucción RET vuelve a la función llamadora.

Cuando termina el epílogo, la pila no queda balanceada, así que la función llamadora se debe encargar de sumar al puntero de pila ESP la cantidad de bytes correspondiente a los parámetros.

Si no hay variables locales, la última instrucción del prólogo y la primera del epílogo no hacen falta.

Llamada a función de C en Assembler

Como comentamos anteriormente, lo primero que debe hacer el código para llamar a una función, es poner los parámetros en la pila.

Si los tipos de datos de los parámetros tienen 1 byte (char), 2 bytes (short) o 4 bytes (int, long, float o puntero a cualquier tipo), se ponen 4 bytes en la pila. Si tienen 8 bytes (long long o double) se ponen los 8 bytes en la pila.

En el ejemplo se puede observar una llamada a función que tiene dos argumentos. El orden de las instrucciones PUSH es de derecha a izquierda, luego llamamos a la función y finalmente liberamos el espacio ocupado por ambos argumentos, que como ocupan 4 bytes cada uno, debemos sumar 8 al puntero de pila ESP.

Valor de retorno

En caso que el código en C llame a nuestra rutina en Assembler, debemos saber en qué registros debe ubicarse el valor de retorno.

Si la función retorna un byte (char), el resultado debe ubicarse en el registro AL.

Si la función retorna dos bytes (short), el resultado debe ubicarse en el registro AX.

Si la función retorna cuatro bytes como entero o puntero, el resultado debe ubicarse en el registro EAX.

Si la función retorna 8 bytes como entero, deberá ubicarse la parte alta del resultado en el registro EDX y la parte baja en EAX.

Si la instrucción retorna el tipo float o double, el resultado debe ubicarse en el primer elemento de la pila de la unidad de punto flotante ST(0).

Ejemplo 1

Aquí se puede ver cómo se compila en Assembler la función en C que se muestra a la izquierda.

Como no hay variables locales, se puede ver que se usa la versión resumida de dos instrucciones del prólogo y del epílogo.

Ahora veremos el cuerpo de la función.

El segundo parámetro, seg, tiene la dirección efectiva EBP+12. Por lo tanto la instrucción seg++; se traduce a INC DWORD [EBP+12]. La cláusula DWORD indica cuatro bytes, que es el tamaño del tipo int.

La segunda instrucción, return pri + seg * 7 debe colocar el resultado del cálculo en el registro EAX, ya que ahí debe ir el valor de retorno porque tiene tipo int.

Primero cargamos en el registro EAX el valor de seg, que como dijimos, se encuentra en la dirección efectiva EBP+12 y luego lo multiplicamos por 7.

El resultado de la multiplicación se encuentra en los registros EDX la parte alta y EAX la parte baja. Como en el estándar C sólo se utiliza la parte baja de la multiplicación, el valor del registro EDX no nos interesa. Al producto que quedó en EAX, le sumamos el valor del argumento pri, que se encuentra en la dirección efectiva EBP+8. De esa manera, queda el valor de retorno en el registro EAX.

Conversión de tipos de datos enteros

En C la conversión de tipos de datos es muy habitual. Por ejemplo, si quiero sumar una variable de tipo char a una variable de tipo int, existe una conversión automática de la primera variable a int para poder realizar la suma, ya que ambos operandos de la suma deben tener el mismo tamaño.

Cuando se hace una conversión de tipos de enteros, existen dos posibilidades dependiendo de si el tipo más pequeño es signado o no. En el primer caso, para expandir la variable, se utiliza el bit más significativo que es el de signo.

En el primer ejemplo, el short 0xA0A0 tiene el bit más significativo a uno (signo negativo), por lo que completamos con unos a la izquierda y el resultado es 0xFFFFA0A0.

En el segundo ejemplo, el short 0x2020 tiene el bit más significativo a cero (signo positivo), por lo que completamos con ceros a la izquierda y el resultado es 0x00002020.

Si el tipo más pequeño no es signado, siempre se completa con ceros.

Por ejemplo, unsigned short 0xA0A0 se convierte a int agregando ceros a la izquierda y el resultado es 0x0000A0A0.

Instrucciones MOVZX y MOVSX

El procesador tiene dos instrucciones para convertir rápidamente los tipos de datos: MOVZX (move with zero extension) y MOVSX (move with sign extension).

MOVZX se usa cuando el tipo pequeño es no signado y lo que hace es completar el destino con ceros a la izquierda.

MOVSX se usa cuando el tipo pequeño es signado y completa el destino con el bit de signo del operando fuente.

Se pueden ver dos ejemplos de conversión de datos signados y no signados.

En el primer ejemplo queremos convertir un signed char que se encuentra en BL al tipo int almacenándolo en el registro EAX. Ejecutamos la instrucción MOVSX EAX,BL.

En el segundo ejemplo convertimos un unsigned short que se encuentra en el primer argumento al tipo int almacenándolo en el registro ECX. La instrucción a ejecutar es MOVZX ECX,WORD [EBP+8]. La cláusula WORD es necesaria porque podríamos convertir un byte también.

Instrucción LEA

La instrucción LEA se usa para cargar punteros, generalmente a la pila, es decir argumentos y variables locales.

Esta instrucción tiene dos operandos. El primero, que es el destino, es un registro, y el segundo que es el fuente, es un direccionamiento indirecto a memoria, y por eso lleva corchetes.

Sin embargo, la instrucción no lee memoria, sino que sólo calcula el puntero y lo carga en el registro destino. El procesador no valida el puntero, así que no se genera excepción 13 o 14 si el puntero es inválido.

Se pueden ver dos ejemplos:

En el primer ejemplo obtenemos un puntero a una variable local mediante LEA ECX,[EBP-4]. El procesador calcula EBP-4 y pone el resultado de la resta en el registro ECX.

En el segundo ejemplo obtenemos el puntero a un elemento de un array global de enteros, que comienza en la dirección efectiva ESI+0x224 y el índice se encuentra en el registro EDI. El procesador calcula ESI + 4*EDI + 0x224 y almacena el resultado de este cálculo en el registro EDX.

Ejemplo 2

En este ejemplo se puede ver una variable local y una conversión de datos de char a int, porque la segunda instrucción de la función tiene una resta de dos tipos diferentes, y el compilador genera una conversión de datos automática.

Se puede ver en rojo el prólogo y en verde el epílogo. La cantidad de bytes de variables locales es de 4, porque tenemos una variable de tipo int. Por eso reservamos el área de variables locales mediante la instrucción SUB ESP,4 en el prólogo.

Ahora vamos a analizar el cuerpo de la función, que tiene las instrucciones marcadas en azul.

La primera instrucción ejecutable en C es varlocal = k - m; donde el argumento k tiene tipo int y el argumento m tiene tipo char. Por lo tanto debemos utilizar una instrucción de conversión de datos. Como el tipo más pequeño es signado, la instrucción es MOVSX.

El argumento m es el segundo, así que su dirección efectiva es EBP+12. Por lo tanto, leemos el valor de m mediante la instrucción MOVSX EBX,BYTE [EBP+12].

Luego leemos el primer parámetro k mediante la instrucción MOV EAX,[EBP+8] y restamos ambos parámetros mediante SUB EAX,EBX. El resultado debe colocarse en la variable local varlocal mediante la instrucción MOV [EBP-4],EAX ya que varlocal se encuentra en la dirección efectiva EBP-4.

Para la instrucción return varlocal * k; multiplicamos el valor recién hallado por el parámetro k dejando la parte baja del producto en el registro EAX.

Comparaciones

En las instrucciones if, while y for, normalmente se comparan dos operandos y se salta si es igual, mayor, menor, etc. Ambos operandos deben ser signados o no signados. En caso contrario el estándar C no define cuál es el resultado, y el compilador señala una advertencia (warning en inglés).

Para compilar la comparación, primero se genera una instrucción CMP para comparar ambos operandos y luego se genera la instrucción de salto condicional dependiendo del tipo de los operandos, signados o no, y de las seis clases de comparaciones, por igual, mayor, menor, distinto, mayor o igual o menor o igual.

Ejemplo 3

En este ejemplo vemos punteros, arrays y comparaciones.

La función tiene un array de 6 enteros, un short y un int como variables locales. En el prólogo se puede ver que se reservan 6*4 bytes para el array, 4 para el short y 4 para el int.

Esto es así porque las variables de tipo int se ubican en las direcciones efectivas múltiplo de 4. Así que hay dos bytes en el área de variables locales que no se utilizan.

La primera instrucción ejecutable es ptrArr = arr; que se resuelve mediante dos instrucciones de Assembler. Primero obtenemos el puntero al array arr mediante la instrucción LEA EAX,[EBP-24] y luego cargamos ese puntero en la variable local correspondiente mediante la instrucción MOV [EBP-32],EAX.

Luego implementamos el ciclo for.

La primera parte del ciclo for, que es m=0, siendo la variable local m de tipo short, es decir 16 bits, se efectúa mediante la instrucción MOV WORD [EBP-28],0.

A continuación ponemos la etiqueta ciclo, que indica donde comienza el cuerpo del for.

Dentro del ciclo for, debemos compilar la instrucción arr[m] = m; Hay que tener en cuenta que el array local arr es de tipo int y la variable local m es de tipo short, así que hay una conversión automática para poder realizar la asignación.

Lo primero que debemos hacer es convertir a int la variable local m, y eso se logra mediante la instrucción MOVSX ECX,WORD [EBP-28].

Para completar la asignación, debemos escribir el registro ECX en el elemento indicado del array.

El array comienza en la dirección efectiva EBP-24 y el elemento a escribir, que es m, ya está en el registro ECX. Así que la instrucción de Assembler es: MOV [EBP-24 + ECX*4],ECX.

A continuación debemos cerrar el ciclo for.

Incrementamos la variable local m mediante la instrucción INC WORD [EBP-28] y luego lo comparamos contra el valor final 6 mediante CMP WORD [EBP-28],6.

Debemos utilizar el salto signado por menor, y por eso usamos la instrucción JL ciclo.

Como la función es void, no necesitamos poner ningún valor en el registro EAX.

Secciones

Lo que vimos hasta ahora es el manejo de parámetros de funciones y variables locales que se encuentran en la pila.

Ahora veremos dónde se ubica el código y variables globales y estáticas.

El programa ejecutable se divide en secciones, que identifican el tipo de información que poseen.

Las secciones más importantes para nosotros son cuatro:

El código ejecutable se encuentra en la sección .text.

Los datos constantes, es decir aquéllos que no varían durante la ejecución del programa, se encuentran en la sección .rodata

Los datos inicializados que se pueden modificar se almacenan en la sección .data

Finalmente, los datos no inicializados se almacenan en la sección .bss. La rutina de C que corre antes de la función main inicializa esta sección a cero.

Ejemplo 4

En este ejemplo se ven variables globales y estáticas.

Las variables y arrays que no tienen la cláusula static se pueden ver desde otros módulos de C, así que cuando se genera código en Assembler, las variables globales deben estar incluidas en la directiva global. Las variables estáticas no se deben usar con la directiva global.

Las variables declaradas con extern en lenguaje C están definidas en otro módulo y cuando se genera código en Assembler, debemos incluir estas variables en la directiva extern.

En el ejemplo vemos cuatro variables, tres definidas en este módulo y otra definido en un módulo externo.

Los arrays arr y str son globales, así que en Assembler debemos escribir las directivas global arr y global str.

La variable ctr es estática así que no corresponde directiva global o extern en Assembler.

La variable value está definida en otro módulo y la declaramos en Assembler mediante la directiva extern value.

El array definido mediante int arr[100]; no está inicializado, así que declaramos la sección .bss mediante la directiva section .bss y reservamos espacio para 100 enteros usando arr resd 100.

El array char str[] = "Hola mundo"; está inicializado, así que lo definimos en la sección .data. Entonces, usamos la directiva section .data y luego declaramos el array de caracteres mediante str db "Hola mundo", 0. El cero es el terminador de la cadena de caracteres.

La variable estática ctr declarada mediante static int ctr = 3; también está inicializada y debe estar en la sección .data. Así que no hace falta declarar una nueva directiva section. Luego declaramos la variable de 32 bits mediante ctr dd 3.

Pasaje de parámetros en 64 bits

Si bien en este video hablamos de ABI de gcc en procesadores Intel de 32 bits, acá incluimos cómo se pasan los parámetros en el caso de aplicaciones de 64 bits.

No siempre se pasan los parámetros por la pila. Los seis primeros parámtros de tipo entero pasan por registro: RDI, RSI, RDX, RCX, R8 y R9 en ese orden.

En el caso de argumentos de punto flotante (float o double), se usan registros SIMD para los primeros 8 parámetros: XMM0, XMM1, XMM2, XMM3, XMM4, XMM5, XMM6 y XMM7 en ese orden.

Si hay más parámetros, se pasan por la pila.

Espero que les haya sido de interés. Hasta luego.

Interrupciones en modo protegido

por Dario Alejandro Alpern

Archivos necesarios

- Ejemplo_Interrupciones.asm
- init_pci.inc

Transcripción

Hola. Mi nombre es Darío Alpern y hoy vamos a ver interrupciones en modo protegido.

Interrupción

Una interrupción es el cambio temporario de flujo del programa que se está ejecutando debido a un evento externo al procesador, por ejemplo apretar un tecla, la llegada de caracteres por el puerto serie, etc.

En el diagrama se puede ver el programa principal que está corriendo hasta que llega un evento de hardware.

En ese momento el procesador termina de ejecutar la instrucción en curso y ejecuta el manejador de la interrupción, que es una subrutina cuyo contenido depende del evento de hardware, por ejemplo puede poner el carácter que llega del puerto serie en una FIFO.

Luego de terminar de ejecutar el manejador de interrupción, el procesador continúa ejecutando el programa principal desde la siguiente instrucción después de la última que había ejecutado antes de que ocurriera la interrupción.

Es importante que el manejador de interrupciones salve el contenido de los registros cuando entra y los recupere antes de salir, ya que en caso contrario, los valores de los registros no serían los esperados y se colgaría el programa o ejecutaría algo no esperado.

8259

Aquí se puede ver cómo se conectan los controladores programables de interrupción (PIC por la sigla en inglés) al procesador.

Originalmente el PIC estaba implementado mediante el chip 8259 de Intel. Actualmente esta funcionalidad se encuentra dentro del chipset en el motherboard de la computadora.

El PIC tiene una interfaz de programación. El primer PIC (configurado como maestro) usa los puertos 20 hexa y 21 hexa y el segundo (configurado como esclavo) usa los puertos A0 hexa y A1 hexa.

Puertos

Los puertos de entrada/salida conforman un espacio separado del de memoria y se utiliza para periféricos que requieran pocas direcciones, como el caso del PIC que necesita dos direcciones para operar.

Aquí se ven las instrucciones que se utilizan para acceder a los puertos tanto para escritura como para lectura. Se pueden leer o escribir 8, 16 ó 32 bits por vez. Como el PIC solo usa 8 bits de datos, usaremos el registro AL para almacenar el dato.

Abajo se puede ver un ejemplo de escritura en un puerto.

8259

El sistema heredado tiene 15 entradas de interrupción denominadas IRQ0 a IRQ15, excepto IRQ2. Actualmente, con APIC, puede haber hasta 224 entradas de interrupción.

Las señales IRQ0 a IRQ7 se conectan al PIC amo y las señales IRQ8 a IRQ15 se conectan al PIC esclavo. Varios de estas señales de interupción son estándar. Por ejemplo IRQ0 viene del timer, IRQ1 viene del teclado, IRQ3 viene del puerto serie COM2, IRQ4 del puerto serie COM1, IRQ8 del reloj de tiempo real, etc.

Máscara de PIC

En general solo queremos habilitar algunas fuentes de interrupción, no todas. Para esto se usa la máscara de interrupciones, que es un registro interno del PIC.

Este registro tiene 8 bits, uno por cada IRQ. Si el bit vale uno, la interrupción está deshabilitada, mientras que si vale cero, la interrupción está habilitada.

Este registro se accede mediante el puerto 21 hexa en el PIC amo y el puerto A1 hexa para el PIC esclavo.

Cuando el timer 0 de 16 bits termina su cuenta, dispara la señal IRQ0 que está marcado con el número 1 en el diagrama. Si la máscara de interrupciones del PIC tiene IRQ0 habilitado y no se está ejecutando esa interrupción u otra de mayor prioridad, entonces el PIC pone a '1' la señal INT (marcado con el número 2) que está conectada a la pata interrupt request del procesador.

Si el procesador tiene el bit 9 del registro EFLAGS a '1', indicando que las interrupciones estándar habilitadas, entonces luego de terminar la instrucción en curso, no continúa con la siguiente instrucción, sino que habilita la señal interrupt acknowledge (marcado con el número 3). Esto enciende el bit correspondiente a la IRQ en curso del registro interno denominado In Service Register (ISR), indicando que el PIC está procesando esa IRQ.

El procesador envía un segundo ciclo de interrupt acknowledge (marcado con el número 4), lo que hace que el PIC envíe el tipo de interrupción (que es un número de 8 bits) por el bus de datos (marcado con el número 5).

Tipo de interrupción

El tipo de interrupción tiene 8 bits. Los cinco bits más significativos se programan durante la inicialización del PIC, mientras que los tres bits menos significativos indican el número de IRQ.

Como hay dos PICs en el sistema, se deben inicializar ambos dispositivos para programarlos con tipos diferentes. Por ejemplo, si queremos que el primer PIC genere tipos 20 hexa a 27 hexa, lo programaremos la parte variable con el número 20 hexa y si queremos que el segundo PIC genere tipos 28 hexa a 2F hexa, lo programaremos con el valor 28 hexa.

IDT

Aquí se puede ver la tabla de vectores de interrupción que usa el procesador cuando está en modo protegido. El formato de la tabla de vectores de interrupción en modo real es diferente y no lo vamos a ver aquí.

La tabla de descriptores de interrupción (en inglés la sigla es IDT (Interrupt Descriptor Table) tiene una compuerta por cada tipo de interrupción. El registro IDT apunta a la compuerta correspondiente al tipo de interrupción cero. A continuación se encuentra la compuerta correspondiente al tipo de interrupción 1, y así sucesivamente hasta el tipo más alto de interrupción que se genere en el sistema. Como existen 256 diferentes tipos de interrupción, la tabla tendrá una longitud máxima de 256 compuertas * 8 bytes/compuerta = 2048 bytes. Si por ejemplo, las interrupciones que se generan están en el rango 0x20 a 0x2F, entonces la tabla tendrá 48 entradas. En este caso el límite de la IDT que deberemos poner en la imagen de la IDTR será 48 * 8 - 1 = 383.

Las compuertas son direcciones lógicas que se componen de selector y offset y tienen el formato que se muestra a la derecha. El tipo de compuerta en el caso de interrupciones es 1110 binario. La dirección lógica apunta al inicio del manejador de la interrupción.

End of Interrupt

Cuando termina el manejador de interrupción, éste debe enviar el comando de Fin de interrupción (End of Interrupt) al PIC. Hay dos tipos de comandos de Fin de Interrupción: específico y no específico. El específico apaga el bit indicado por el comando del registro In Service Register (ISR), mientras que el no específico apaga el bit encendido del In Service Register de la IRQ que tenga mayor prioridad. La prioridad más alta la tiene el IRQ0 y la máscara baja la IRQ7. Nosotros siempre vamos a usar comandos de Fin de interrupción no específicos. Una vez que el bit del In Service Register está apagado, el PIC puede volver a interrumpir el procesador con esa IRQ.

Abajo se pueden ver las instrucciones que se deben ejecutar para enviarle al pic el comando de Fin de interrupción no específico.

Multiplicación

Una vez calculada la semilla pseudoaleatoria, obtenemos un número de 32 bits. Tenemos que convertir ese número en el rango de 0 a EBX exclusive.

Para ello, observando como funciona la multiplicación, se puede ver que la parte alta del producto de 64 bits se almacena en EDX y la parte baja en EAX.

Vamos a ver cuánto vale el producto máximo cuando multiplico por el argumento. Eso ocurre cuando EAX = $2^{32} - 1$.

Entonces, haciendo algunas transformaciones se puede ver que el producto es 2^{32} (argum-1) + (2^{32} − argum) donde ambos paréntesis son números que entran en 32 bits.

Comparando las partes altas, es decir las que están multiplicadas por 2^{32}, resulta que el máximo valor posible de EDX es argum − 1, por lo tanto EDX está en el rango entre 0 y argum excluyendo el último valor.

Programmable Interval Timer

Originalmente implementado mediante el circuito integrado Intel 8253 y luego por Intel 8254, actualmente se encuentra dentro del chipset en el motherboard.

El PIT tiene 3 canales de 16 bits. El canal cero se conecta a la entrada IRQ0 del controlador de interrupciones. El canal uno se utilizaba junto con el controlador de DMA para refrescar memorias dinámicas. El refresco de las memorias dinámicas hoy en día no usa este canal y por lo tanto está deshabilitado. El canal dos se usa para generar una frecuencia para el parlante en la PC. Para que no suene todo el tiempo, la salida de este canal tiene una compuerta AND lógica que está controlada por el bit 0 del puerto 61 hexa.

Para programar el PIT, se debe escribir una palabra de control en el puerto 43 hexa y luego uno o dos bytes en el puerto correspondiente al canal que queremos programar.

Hay dos fuentes de interrupción: una es el timer que es el que muestra los caracteres y otro es el teclado que es el que borra la pantalla.

Introducción a modo protegido

por Dario Alejandro Alpern

Transcripción

Hola. Mi nombre es Dario Alpern y hoy vamos a ver introducción al modo protegido.

Esquema de un sistema operativo

Para entender el uso de modo protegido y qué se quiere proteger, veremos un esquema genérico de sistema operativo.

Un sistema operativo maneja el hardware de las computadoras, como la memoria, los discos, teclado, mouse y otros periféricos, y también recursos de software.

De esta manera, si un programa necesita, por ejemplo, imprimir un documento, no maneja directamente la impresora, sino que a través de órdenes dadas al kernel del sistema operativo, éste encola el trabajo, para que varios programas puedan acceder a la impresora. Estos comandos siguen un protocolo denominado API (Application Programming Interface), que depende del sistema operativo específico que estemos usando.

La comunicación entre el kernel y el hardware se realiza mediante drivers, que es un software que permite abstraer el tipo de hardware. Por ejemplo, el kernel maneja todas las impresoras de la misma manera, y el driver permite acomodar las diferencias que existen entre los comandos de bajo nivel que aceptan las distintas impresoras.

Una aplicación es un programa o conjunto de programas que realiza tareas útiles para el usuario. Ejemplos de aplicaciones pueden ser un procesador de textos, un navegador Web o un juego.

Un programa es un conjunto de instrucciones que realiza una tarea determinada que puede ser simple o compleja. Un proceso es una instancia de programa en ejecución. Por ejemplo, nosotros podemos abrir tres calculadoras en la pantalla, lo que significa que hay tres procesos corriendo que corresponden al mismo programa.

Por lo tanto se puede ver que a una aplicación le corresponde uno o más procesos.

Los threads permiten correr procesos de manera concurrente. Muchos algoritmos se pueden ejecutar en paralelo acelerando su ejecución. Los threads comparten memoria y otros recursos, a diferencia de los procesos que son independientes entre sí.

Estos threads pueden estar despiertos cuando están haciendo algo, o bien pueden estar dormidos, cuando esperan por entrada de teclado o mouse u otro periférico. En el ejemplo de la calculadora, la mayor parte del tiempo los threads están dormidos, hasta que el usuario aprieta un botón y el proceso muestra el dígito ingresado o realiza el cálculo pedido. Luego se vuelve a dormir.

Un sistema operativo multitarea es capaz de ejecutar durante una fracción de segundo cada thread que esté despierto. Así se van turnando la ejecución todos los threads despiertos dando la impresión que los threads están corriendo simultáneamente.

Pero en realidad en un momento determinado, el procesador está ejecutando un thread específico.

Lo que queremos lograr usando las capacidades de modo protegido, es que los procesos no choquen entre sí o con el kernel o los drivers.

Niveles

Parte de esto se puede conseguir mediante el concepto de privilegio. Vamos a decir que tanto el kernel como los drivers tienen privilegio de kernel, mientras que el resto del sistema tiene privilegio de aplicación.

Un programa corriendo con privilegio de aplicación no puede leer ni escribir datos que tengan privilegio de kernel, y ningún segmento de código se puede escribir.

El procesador posee cuatro niveles de privilegio. En nuestro caso, el privilegio de kernel es el nivel cero, mientras que el privilegio de aplicación es el de nivel 3. No vamos a usar los niveles 1 y 2 de privilegio.

En esta explicación falta indicar cómo hacer que un proceso no sobreescriba o lea datos de otro proceso, ya que ambos tienen privilegio de aplicación.

Tabla local

Para solucionar este problema, el procesador maneja tablas locales por proceso que solo permiten acceder a la memoria de código o datos que corresponde a ese proceso. El procesador sólo ve una de estas tablas locales por vez. Entonces el proceso A no puede escribir datos del proceso B porque la tabla local del proceso A no tiene puntero a datos del proceso B.

Unidad de segmentación

La unidad de segmentación posee 6 registros de segmentos, que sirven para apuntar al código mediante el registro CS (code segment), a la pila mediante el registro SS (stack segment) o a datos mediante los otros cuatro registros de segmento.

Cuando el programador escribe en un registro de segmento, ese valor se interpreta como selector. En modo protegido, el procesador modifica los otros tres campos a partir de datos almacenados en tablas de descriptores.

El procesador también dispone de cuatro registros de direcciones del sistema. Excepto el registro TR, los otros tres apuntan a tablas de descriptores.

El registro LDTR (Local Descriptor Table Register) apunta a la tabla local de descriptores que incluye los punteros a los segmentos que se pueden acceder desde el proceso que se está ejecutando.

El registro GDTR (Global Descriptor Table Register) apunta a la tabla global de descriptores que incluye los punteros a los segmentos que se pueden acceder a cualquier proceso.

El registro IDTR (Interrupt Descriptor Table Register) apunta a la tabla de descriptores de interrupción que incluye una compuerta por cada tipo de interrupción. El procesador puede vectorizar hasta 256 tipos diferentes de interrupción, así que la tabla de descriptores de interrupción tiene como máximo 256 entradas.

El registro TR (Task Register) se utiliza para multitarea y se verá su funcionamiento en otro momento.

El microprocesador soporta protección por segmentación (que es lo que estamos viendo ahora) y protección por paginación (que se va a ver en otro momento). Como vamos a hacer énfasis en la protección por paginación, no vamos a utilizar la tabla local de descriptores.

Tablas de descriptores

En este diagrama se pueden ver las tres tablas de descriptores apuntados por los registros correspondientes.

Cada descriptor ocupa 8 bytes.

Dentro de la tabla global de descriptores vamos a encontrar descriptores de código y datos que sirven para apuntar a segmentos de código y datos, descriptores de LDT, que apuntan a tablas de descriptores locales, compuertas de llamada, que sirven para llamar a rutinas en diferente nivel de privilegio, y también existen otros descriptores más.

En la tabla local de descriptores vamos a encontrar descriptores de código y datos que sirven para apuntar a segmentos de código y datos del proceso que esté corriendo.

La tabla de interrupciones contiene compuertas, que son descriptores que incluyen selector y offset. De esa manera el procesador puede ir a la dirección lógica indicada por la compuerta cuando llega una interrupción. En dicha dirección lógica se deberá encontrar la rutina que maneja la interrupción (en inglés se dice interrupt handler).

Selector

A diferencia del modo real, en el que el programador puede cargar cualquier selector en un registro de segmento, en modo protegido el selector tiene el formato indicado en el diagrama.

Los 16 bits del selector se distribuyen en tres campos.

Los bits 15 a 3 especifican el índice dentro de la tabla GDT o LDT. Si este campo vale cero, el selector se refiere a la primera entrada (que son los offsets 0 a 7 de la tabla), si vale 1, el selector se refiere a la segunda entrada (que son los offsets 8 a F hexadecimal) y así

sucesivamente.

El bit 2 indica qué tabla de descriptores va a leer el procesador. Si vale cero, va a leer la GDT, mientras que si vale 1 va a leer la LDT.

Los bits 1 y 0 indican el nivel pedido de privilegio. Por ahora este número coincide con el nivel de privilegio indicado por el descriptor a leer en la GDT o LDT.

Descriptor de código y datos

Aquí podemos ver los 8 bytes del descriptor de código y datos.

Los bytes 2, 3, 4 y 7 contienen la dirección lineal de la base del segmento de código o datos.

Los bytes 0, 1 y la mitad inferior del byte 6 contienen 20 bits del límite. El límite es la longitud del segmento menos 1.

Como el campo límite de los registros de segmento tiene 32 bits, se utiliza el bit 7 del byte 6 que es el bit G, de granularidad para extender el límite a 32 bits como se verá en el siguiente diagrama.

El bit D (Default), cuando vale cero indica que el segmento es de 16 bits y si vale uno indica que el segmento es de 32 bits. Nosotros siempre vamos a usar segmentos de 32 bits.

Segmentos de 16 y 32 bits

En los segmentos de código de 32 bits se invierte el uso de los prefijos 66 y 67 con respecto a los segmentos de 16 bits. Esto quiere decir que si quiero cargar un registro de 32 bits, no se usa el prefijo 66, mientras que si quiero cargar un registro de 16 bits, sí se usa el prefijo 66.

Lo mismo ocurre con el prefijo 67 con respecto a direcciones de 16 o de 32 bits.

Como los códigos binarios son diferentes en segmentos de 16 bits que en segmentos de 32 bits, hay que especificarle al ensamblador qué tipo de registro estamos generando.

Excepto este manejo de los prefijos 66 y 67, el procesador soporta el mismo conjunto de instrucciones y los mismos registros tanto en segmentos de código de 16 bits como en segmentos de código de 32 bits.

Con la directiva USE16 las instrucciones que se ensamblan a continuación son para segmentos de código de 16 bits, mientras que con la directiva USE32 las instrucciones que se ensamblan a continuación son para segmentos de código de 32 bits.

Descriptor de código y datos

A la derecha se ve el byte de derechos de acceso. Esta información se carga en el campo atributos del registro de segmento correspondiente.

El bit 7 debe estar a 1 indicando que el segmento está presente. Los bits 6 y 5 indican el nivel de privilegio del segmento apuntado por este descriptor donde 0 es el máximo privilegio y 3 el mínimo.

Se puede ver que el contenido de los bits 3, 2 y 1 dependen si el descriptor es de código o de datos.

En el caso de descriptor de código, el bit 3 vale 1. El bit 2 indica si el segmento de código es conforme o no. Un segmento conforme es aquel que cuando lo llaman, mantiene el nivel de privilegio del que lo llamó. Estos segmentos siempre tienen DPL = 0. Si lo llama un segmento de privilegio de kernel, este segmento de código seguirá con este privilegio, mientras que si lo llama un segmento de privilegio de aplicación, el segmento conforme se comportará como de aplicación. Cuando el segmento no es conforme, su nivel de privilegio es el indicado por el campo DPL del descriptor. El bit 1 indica si el segmento de código se puede leer o no. Un segmento de código nunca se puede escribir.

En el caso de descriptor de datos, el bit 3 vale cero. El bit 2 indica la dirección de expansión del segmento de datos. La dirección de expansión indica qué offsets son válidos. Si la dirección de expansión vale 0, que es lo normal, entonces el offset debe ser menor o igual que el límite. Si vale 1. el offset debe ser mayor que el límite. El bit 1 es el permiso de escritura. Un segmento de datos siempre se puede leer.

El bit cero indica si el segmento fue accedido. Hay que inicializar este bit a cero.

Cálculo del límite

En el descriptor, el límite ocupa 5 de los 8 dígitos hexadecimales o nibbles. El bit de granularidad indica como se rellenan los otros tres nibbles. Si el bit de granularidad vale cero, entonces la parte alta del límite se rellena con ceros.

Si el bit de granularidad vale uno, el límite se multiplica por 4096, poniendo 3 "efes" a la derecha del límite.

Ejemplo

Definir descriptor de datos en la entrada 3 de la GDT que tenga base 33500000 hexa y límite 00192FFF hexa que se pueda leer y escribir y tenga DPL = 0. Indicar cuál es el selector que le corresponde.

Ejemplo 1

Usando la base y el límite completamos a la derecha los valores que corresponden al descriptor de datos.

Comenzando por la base, el byte más significativo es 33 hexa, así que va en el offset 7 del descriptor. El siguiente byte, 50 hexa, va en el byte 4 y luego 00 en el byte 3 y el byte 2.

Siguiendo con el límite, como termina en tres efes, la granularidad vale 1 y los cinco nibbles a poner en el descriptor son 00192 hexa. En el byte 6 pondremos el bit 7 de granularidad a uno, el bit 6 de default a 1 porque dijimos que vamos a usar segmentos de 32 bits, y el nibble más significativo del límite es cero, por lo que el valor del byte 6 debe ser C0 hexa. El byte 1 son los dos nibbles siguientes, 01 hexa y el byte 0 es el byte menos significativo del límite. Eso nos da 92 hexa.

El byte de derechos de acceso tenemos el bit 7 de presente a uno, los bits 6 y 5 de DPL a 00, el bit 4 a uno, que es fijo para descriptores de código y datos, el bit 3 a cero, porque es un descriptor de datos, el bit 2 de dirección de expansión a cero, que es lo normal, el bit 1 que es la habilitación de escritura a uno y finalmente ponemos el bit 0 de accedido a cero.

En la parte inferior del diagrama se puede ver la directiva define byte que vamos a usar en nuestro código fuente para definir este descriptor, indicando los 8 bytes que pusimos en la columna de la derecha, desde el offset cero hasta el offset 7.

Cálculo del selector

Para calcular el selector, debemos descomponerlo en los tres campos. Como dice el enunciado, el índice vale 3. El indicador de tabla vale cero porque el descriptor debe estar en la GDT. El RPL vale cero para que coincida con el DPL del descriptor.

Esto significa que el selector vale 0018 hexa.

Se puede ver un ejemplo para cargar el data segment con este selector.

Y finalmente podemos ver cómo quedan los cuatro campos del registro de segmento luego de haber cargado el selector 0018 hexa.

Compuertas

Las compuertas son descriptores que contienen selector y offset, es decir una dirección lógica. Estas compuertas pueden apuntar a manejadores de interrupciones o excepciones, o bien a subrutinas en el caso de compuerta de llamada.

En el diagrama se puede observar los seis bytes donde se ubican el selector y el offset.

Excepto la compuerta de llamada que va en la GDT, las otras tres compuertas se ubican en la IDT.

Carga de registros GDTR e IDTR

Como las tablas GDT e IDT tienen dirección lineal de inicio y longitud, tenemos que indicar esta información al procesador para que los almacene en los registros GDTR e IDTR.

La carga de estos registros es muy particular y requiere que ambos parámetros estén en memoria, en una estructura denominada imagen de GDTR o imagen de IDTR.

En ambas estructuras, primero se ubican los dos bytes del límite y luego los cuatro bytes de la base, especificados en el formato little-endian.

Una vez que se completaron las imágenes, se procede a la carga de estos registros especiales mediante las instrucciones LGDT y LIDT, especificando como parámetro el puntero a la imagen.

Registro CR0

El registro CR0 es un campo de bits. Los dos bits más importantes son el bit cero para habilitar el modo protegido y el bit 31 para habilitar la paginación. No se puede activar la paginación sin activar también el modo protegido.

Cuando se modifique CR0 hay de asegurarse que no se toquen otros bits. La secuencia a usar es la que se ve abajo.

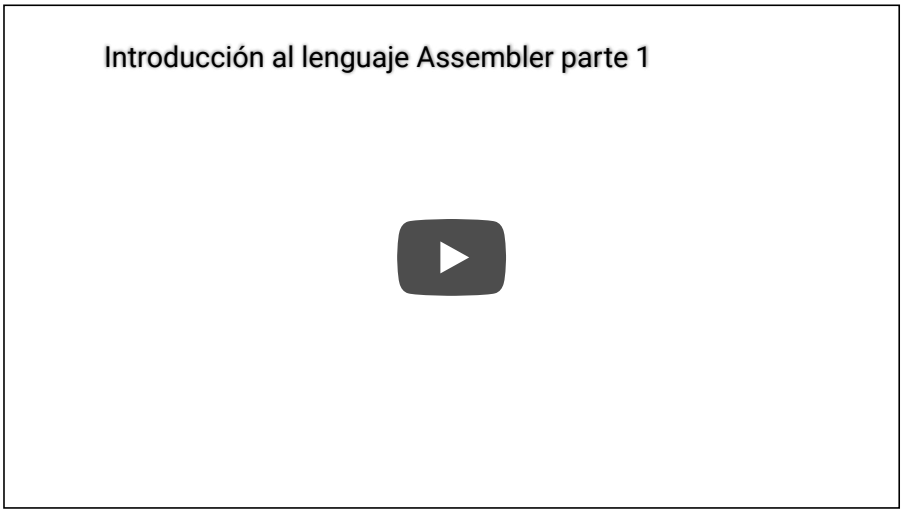
Pasaje a modo protegido

Aquí se ven los pasos necesarios para pasar a modo protegido. Es sumamente importante que las interrupciones estén deshabilitadas durante todo este proceso.

Espero que les haya sido de interés. Hasta luego.

Introducción al lenguaje Assembler parte 1

por Dario Alejandro Alpern



Transcripción

Hola. Mi nombre es Darío Alpern y hoy vamos a ver Introducción a Assembler de procesadores Intel.

Código de máquina

Luego de terminar el reset, todos los microprocesadores ejecutan código de máquina que obtienen de memoria hasta que se van a dormir o se apagan.

Aquí se puede ver un ejemplo de código de máquina que no sabemos qué es lo que hace. A la izquierda se muestran las direcciones lógicas que voy a explicar más adelante. A la derecha, los bytes que forman el código de maquina agrupados de a 8 bytes.

Desensamblado

Para entender qué es lo que va a ejecutar el procesador, todos los fabricantes publican el assembler, que es una traducción legible para humanos de lo que hace el microprocesador. Una instrucción de código de máquina corresponde a una instrucción en assembler.

A la izquierda se pueden ver las direcciones lógicas, en el centro los bytes que componen el código de máquina para cada instrucción, y a la derecha la decodificación en instrucciones de assembler.

A su vez la instrucción en código de máquina se puede subdividir en tres partes: prefijos en azul, códigos de operación en rojo y operandos en negro.

Lo que se ve es una subrutina que halla la raíz cuadrada del número que se encuentra en la posición de memoria 0300 y almacena el resultado en la posición de memoria 0304.

Registros de uso general

Aquí se ven los 8 registros de 32 bits de uso general del procesador. Estos registros sirven para el almacenamiento temporario de información y tienen la ventaja de ser mucho más rápido que usar la memoria principal del sistema.

Los procesadores actuales tienen 16 registros de uso general de 64 bits, pero sólo se pueden acceder en el llamado modo 64 bits.

Con cualquiera de estos registros se pueden realizar operaciones aritméticas y lógicas, pero hay instrucciones que solo pueden usar algunos de estos registros. Por ejemplo el registro ESP siempre se usa como puntero de pila, la multipllicación usa EDX y EAX o bien DX y AX, etc.

Los 8 registros de 32 bits se llaman EAX, EBX,.. ESP. Se puede acceder a los 16 bits menos significativos usando los nombres AX, BX, ... SP.

A su vez AX, BX, CX y DX pueden subdividirse en un par de registros de 8 bits cada uno. Por ejemplo AX se subdivide en AH (parte alta) y AL (parte baja).

Registros de segmento

El procesador permite dividir el espacio direccionable en áreas llamadas segmentos. Estos segmentos se apuntan mediante alguno de los 6 registros CS, DS, ES, FS, GS y SS.

CS es el segmento de código, lo que significa que el código ejecutable debe estar en este segmento.

DS es el segmento de datos. En dicho segmento se encontrarán variables, buffers, etc.

Hay tres registros de segmentos de datos adicionales: ES, FS y GS.

SS es el segmento de pila. La pila debe estar ubicada en este segmento.

Cada registro de segmento posee cuatro campos: selector, base, límite y atributos. El programador sólo puede acceder al campo selector. La lectura del registro de segmento provoca la lectura de su campo selector. La escritura del registro de segmento provoca la escritura del campo selector con ese valor y automáticamente se escribe el resto de los campos.

El campo base indica la dirección lineal donde comienza el segmento en memoria. El campo límite indica el máximo offset permitido para ese segmento. Es decir que es igual a la longitud del segmento menos 1. El campo atributos indica los accesos que posee el procesador a dicho segmento.

Los números que figuran en los diferentes campos indican los valores que el reset carga en los registros de segmento. Los campos en blanco indican que el valor no está definido.

Registro EFLAGS

Este es un registro de indicadores de un bit cada uno (excepto IOPL que tiene dos bits). La parte baja de EFLAGS se denomina FLAGS.

Algunos de estos flags se modifican en las instrucciones aritméticas y lógicas y los saltos condicionales modifican el flujo del programa según el valor de estos flags.

Los flags que se usan en modo protegido se verán en otra clase.

Los otros flags son:

Bit 11: Overflow. Indica que hubo sobrepasamiento en un cálculo de números en complemento a dos. Por ejemplo 60 hexadecimal es positivo en complemento a dos porque el bit más significativo es cero. Si sumo 60 hexa más 60 hexa, el resultado es C0 hexa, que tiene el bit más significativo a uno, es decir que se interpreta como negativo en complemento a dos. Como el cálculo es positivo + positivo = negativo, se prende el flag de overflow.

Bit 10: Dirección: Sirve para instrucciones de cadena, estilo memcpy o memset. Si vale cero, las direcciones fuente y destino se incrementan. Si vale uno, se decrementan. Esto sirve para evitar el efecto dominó cuando el buffer de destino se solapa con el buffer fuente.

Bit 9: habilitación de interrupción. Si vale uno, el procesador reconoce las interrupciones que vienen por la pata de interrupt request. Si vale cero, el procesador ignora dichas interrupciones.

Bit 8: trace. Sirve para la ejecución paso a paso en debuggers. Cuando está encendido, se ejecuta la excepción 1 luego de ejecutar la instrucción en curso.

Bit 7: signo: copia el bit más siginificativo del resultado de la operación aritmética o lógica.

Bit 6: cero: vale uno si el resultado de la operación aritmética o lógica vale cero.

Bit 4: acarreo auxiliar: sirve para realizar operaciones en BCD, cuando hay acarreo o préstamo del bit 3 al 4.

Bit 2: bit de paridad: se enciende cuando la cantidad de bits a 1 del resultado de la operación aritmética o lógica es par.

Bit 0: acarreo (carry en inglés): se enciende cuando hay sobrepasamiento en aritmética de números no signados. Por ejemplo si resto 40 hexa menos 60 hexa, se enciende el bit.

El par de registros CS:EIP apunta a la dirección lógica de la próxima instrucción a ejecutar.

Acceso a memoria

Para acceder a memoria, el programador usa la dirección lógica, que está dado por un selector y un offset. En el caso de acceso a datos, las instrucciones usan prefijos de segmentos para indicar a cuál de los seis registros de segmento se refiere la instrucción. Si el prefijo no está presente, se usa un segmento por defecto, que se va a ver en la próxima página.

El cálculo de la dirección lineal se realiza sumando el campo base del registro de segmento más el offset que se encuentra en la instrucción.

El procesador tiene varios modos de operación. Cuando arranca, luego del reset, se encuentra en modo real. En ese caso la base del segmento se calcula multiplicando el selector por 10 hexadecimal.

Como el tamaño de los segmentos están limitados a 64 KB como se vio anteriormente, no es posible acceder a los 4 GB de direccionamiento en modo real. La dirección máxima apenas supera 1 MB.

Acceso a memoria de 16 bits

Las instrucciones que definen offsets pueden usar registros de 16 o de 32 bits.

Para el caso de 16 bits, el offset o dirección efectiva se calcula como la suma de base, índice y desplazamiento. Donde la base puede ser el registro BX, el registro BP o nada. El índice puede ser el registro SI, el registro DI o nada. El desplazamiento es un entero de 16 bits.

Se pueden ver cuatro ejemplos. En los ejemplos de la derecha se usa el segmento por defecto, que en ambos casos es DS, porque la base es el registro BX.

En el segundo ejemplo el registro CX tiene 16 bits, por lo que el procesador lee dos posiciones de memoria a partir de la dirección indicada.

En el último ejemplo hay que especificar el tamaño del operando, indicando que se leen 4 bytes mediante la palabra DWORD. Otras posibilidades son BYTE para un byte y WORD para dos bytes. Esto ocurre cuando no hay un operando con un registro que especifique el tamaño del operando en memoria. En casi todas las instrucciones de dos operandos coinciden los tamaños de ambos operandos.

Es importante tener en cuenta que en las instrucciones de dos operandos, el de la izquierda es el destino y el de la derecha es el fuente. El resultado de la operación se carga en el destino.

Acceso a memoria de 32 bits

En el caso de 32 bits se agrega un multiplicador. La base puede ser cualquiera de los 8 registros de uso general: EAX, EBX, etc. o nada. El multiplicador puede ser 1, 2, 4 u 8. El índice puede ser cualquiera de los registros excepto ESP o nada. Finalmente, el desplazamiento es un entero de 32 bits.

Se pueden ver cuatro ejemplos. Exceptuando el primer caso, los demás usan el segmento por defecto, que es SS en el ejemplo de abajo a la izquierda y DS en los otros casos.

Cuadrados perfectos

Para entender como funciona la subrutina para hallar la raíz cuadrada, vamos a escribir los primeros cuadrados perfectos.

Diferencias

Y ahora escribimos las diferencias entre dos cuadrados perfectos consecutivos. Podemos ver que son los números impares: 1, 3, 5, 7, etc.

Eso significa que para hallar la raíz cuadrada, basta con restar los números impares sucesivos hasta que el resultado dé negativo. Entonces la raíz cuadrada es la cantidad de restas menos 1.

Por ejemplo, si quiero hallar la raíz cuadrada de 10, le resto 1 (me da 9), luego 3 (me da 6), luego 5 (me da 1), luego 7 (me da -6) y paro ahí porque me dio negativo. Como hice cuatro restas, la raíz cuadrada de 10 es 3.

Ejemplo de programa en Assembler para calcular raíz cuadrada

Ahora vamos a ver el programa en Assembler.

Aquí se puede ver instrucciones como las que vimos en el desensamblado, pero en el programa en assembler que escribimos nosotros, también hay comentarios, que se preceden con punto y coma, directivas (como EQU y USE16) que son comandos para el ensamblador, etiquetas (como Inicio y colgarse) que son el destino de saltos o llamadas a subrutinas.

En esta parte del programa se puede ver la inicialización del puntero de pila en la dirección 0: 8000 hexa y la carga del radicando con el valor 76. Luego se llama a la rutina de cálculo de la raíz cuadrada y finalmente el código hace un ciclo indefinido de saltos (jump) ya que como dijimos al principio, el procesador ejecuta instrucciones a toda velocidad hasta que se va a dormir o se apaga.

A continuación aparece la subrutina raiz_cuadrada. Luego de leer el valor del radicando y poner a uno el sustraendo haciendo XOR del registro consigo mismo poniéndolo a cero y luego sumándole 1 con la instrucción INC. En la línea 26 inicializamos a cero la cantidad de restas.

Luego comienza el ciclo que realiza las sustracciones. Primero restamos al acumulador el número impar que corresponda (sea 1, 3, 5, etc.).

En la línea 29 saltamos si el carry flag vale 1. Eso quiere decir que saltamos si el resultado de la resta es menor que cero, ya que en ese caso no hay que hacer más restas.

Siguendo con el ciclo, en la línea 30 hallamos el siguiente número impar sumando 2 y luego incrementamos la cantidad de restas.

Finalmente con la instrucción JMP saltamos al principio del ciclo (a la línea 22).

El procesador llega a la línea 33 cuando no hay más restas por hacer. Como el valor del registro DX es la cantidad de restas que se hicieron menos 1, guardamos ese valor en memoria en la posición RAIZ. Finalmente terminamos la subrutina con la instrucción RET para volver al programa principal.

El valor inicial de EIP en el reset es FFF0 hexadecimal, por lo que en la línea 37 llenamos con ceros la memoria de código hasta llegar a dicho offset. La directiva times repite la cantidad de veces que está a continuación la instrucción o directiva que sigue a esa cantidad. El valor de $ es el offset que el ensamblador está procesando. De esta manera $ - Inicio es la cantidad de bytes en el segmento de código antes de ensamblar la directiva times. Como queremos ensamblar la instrucción de arranque en el offset FFF0 hexadecimal, restamos FFF0 hexacimal menos $-Inicio para obtener la cantidad de bytes de relleno. Así la cantidad de bytes de programa más el relleno vale FFF0 hexadecimal. La directiva db 0 inserta un byte a cero. Entonces luego de la directiva times, el ensamblador debe procesar el offset FFF0 hexadecimal.

La última instrucción salta al principio del programa principal. $$ apunta al principio del programa.

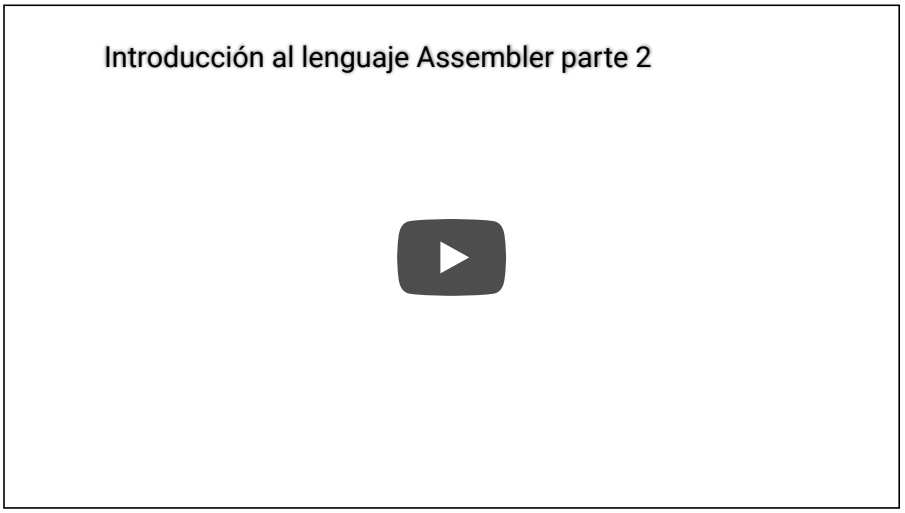
La directiva que se encuentra en la línea 44 hace algo parecido al de la línea 37 agregando relleno para completar los 64 KB de memoria ejecutable.

Ahora vamos a compilar el programa.

No hubo ningún error ni advertencia (warning en inglés), como corresponde. Con el siguiente comando verificamos que el binario tenga exactamente 64 kilobytes, es decir 65536 bytes.

Introducción al lenguaje Assembler parte 2

por Dario Alejandro Alpern



Archivos necesarios

- sqrt.asm

Transcripción

Hola. Mi nombre es Darío Alpern y ahora vamos a ver la segunda parte de la introducción a Assembler de procesadores Intel.

En este video vamos a ver como funciona el Bochs que es un emulador de PC, que emula tanto el microprocesador de Intel como el motherboard de la PC.

Para utilizar este emulador, vamos a usar el programa que vimos en el primer video que es el del cálculo de la raíz cuadrada.

Entonces, vamos a editarlo con el editor Kate.

Apretando CTRL ++ se agrandan las letras y podemos ver los equates que equivalen a defines en lenguaje C para no usar constantes voladoras en el medio del código.

Luego, tenemos el inicio del programa, donde inicializamos puntero de pila y el segmento de datos para que apunten al principio de el espacio direccionable de memoria y luego de eso inicializamos con 76 el valor del radicando, que es el argumento de la raíz cuadrada. Ese es un número que usamos para probar como anda la subrutina.

Luego de ejecutar la subrutina raiz_cuadrada, se ejecuta una instrucción JMP cuyo destino es la misma dirección, por lo tanto termina colgándose.

A continuación tenemos la subrutina raiz_cuadrada que lo que hace es obtener el valor del radicando y le va restando números impares sucesivos hasta que el resultado sea menor que cero.

Entonces, la raíz cuadrada es igual a la cantidad de restas que se hicieron menos uno, como se explicó en el primer video.

Entonces, tenemos la carga del radicando, Después, cargamos al sustraendo el primer número impar que es 1, después ponemos la cantidad de restas a cero, después viene el ciclo principal y finalmente cargamos el resultado con la cantidad de restas que se hicieron y finalmente, se termina la subrutina.

Después de eso, se coloca un relleno de bytes a cero, para que la próxima instrucción que se ejecute esté en la dirección FFF0 hexadecimal. Y luego de eso, se colocan más bytes para rellenar. De esa manera tenemos 64 KB completos de memoria ROM.

Entonces, cerramos, compilamos el programa usando el ensamblador nasm le decimos que la salida con la opción -o sea... Primero ponemos el nombre del programa, después -o el binario y después -l sqrt.lst.

Entonces en el listing se genera información que ustedes pueden ver con las direcciones del programa, el código binario, instrucciones, que son las mismas instrucciones que nosotros escribimos y acá están los diferentes códigos de máquina que se van generando. Eso es el listing. Y una cosa que se puede ver en el listing es que en la dirección inicial del programa que es el EIP que arranca en la dirección

FFF0 hexadecimal tenemos realmente la instrucción que corresponde al principio del programa, y después sabemos que la ROM ocupa 64 KB completos.

Entonces, para poder correr el programa Bochs, hacemos bochs la opción -q es "quiet" que hace que no nos pida ningún menú cuando arranca el Bochs y la opción f es para indicar el archivo de configuración que en nuestro caso es bochs.cfg que es el que habíamos tocado en el primer video.

Entonces, arranca el Bochs y nos muestra el debugger gráfico.

En el panel de la izquierda vemos los registros del procesador. Acá tenemos 8 registros de uso general: EAX, EBX,.... hasta ESP y EBP son todos los registros de uso general.

Después tenemos el Instruction Pointer, que nos dice la dirección efectiva donde está corriendo el código. Fíjense que la dirección efectiva después del reset es FFF0 hexadecimal que tiene que tener la instrucción JMP que habíamos visto.

Después están los flags, que son los indicadores.

Luego tenemos los seis registros de segmento, que como habíamos visto, el registro CS el selector inicial después del reset era F000.

Y después tenemos otros registros más que se van a ver cuando veamos modo protegido.

En el panel central tenemos las instrucciones. Entonces, en verde tenemos la próxima instrucción que se va a ejecutar. En este caso tenemos la dirección lineal FFFFFFF0. ¿Por qué tenemos esa dirección lineal? Habíamos dicho que la dirección base del segmento de código es FFFF0000 y ese valor se determina en el reset. Si a eso le sumamos el valor de IP, nos queda FFFFFFF0.

Si se quiere, podemos ver con el comando sreg en consola, podemos ver información del code segment y acá tenemos que el valor inicial es FFFF0000. Le sumamos me da la dirección lineal que vemos en verde.

Después de esto ejecutamos un "step" y se va a ejecutar el JMP y con esto cambia el valor de IP.

La idea es que la información que está en rojo en el panel de la izquierda son los registros que cambiaron con la instrucción que se acaba de ejecutar.

Entonces, la primera instrucción al principio del programa coloca los registros de segmento a cero.

Entonces, carga AX con cero y después va a cargar el selector de DS y SS a cero. En este caso no hay ningún cambio porque en el reset también estaban a cero, entonces no se modifican.

Después cargo el SP a 8000 hexadecimal y acá podemos ver en rojo que se modificó.

Después cargo el valor de EAX en 76 o 4C hexadecimal y acá se ve como se modifica y ese valor se carga en memoria en la dirección indicada por el DS (data segment) cuya base es cero y el offset es 0300 hexadecimal.

Entonces lo ejecuto, y si quiero ver donde se cargó en la memoria lo que hago es poner View Linear Memdump y coloco la dirección 300 hexadecimal.

Y acá vemos los cuatro bytes que se cargaron 4C 00 00 00 en formato Little-Endian que significa que el byte menos significativo va primero.

Entonces, luego de eso, se va a ejecutar el llamado a subrutina.

Entonces hago "step", y fíjense que aparte de modificarse el Instruction Pointer se modificó el stack pointer: de 8000 pasó a 7FFE o sea bajó dos lugares.

Entonces vamos a ver el contenido de la pila con View Linear Mem dump el valor del puntero de pila es 7FFE.

Borro el resto, y acá usamos el scroll bar para llegar a la derecha y acá tenemos 17 00 que como está en Little-Endian es 00 17 hexadecimal.

Significa que el valor que se puso en memoria es 0017 hexadecimal que es el valor del offset correspondiente al IP del regreso de subrutina. Fíjense que el 0017 es la instrucción que viene después del call. Cuando regrese, tiene que regresar al JMP que está a continuación que es la dirección 0017 hexadecimal.

Luego de eso, una vez que leímos la memoria, ejecutamos y se pone ECX a cero. Acá no se nota porque ya valía cero de antes.

Luego de eso incrementamos ECX y cuando se incrementa ECX va a pasar de cero a uno. Al incrementar, suma uno al registro.

Luego colocamos DX a cero, y entramos en el ciclo principal de restas. Donde se resta 76, que era el número que teníamos para probar la raíz cuadrada, que es el radicando le restamos el valor 1, que es el primer número impar.

Fíjense: EAX = 76 y ECX = 1. Y va valer el resultado 75 que es 4B. La idea es restar números impares consecutivos 1, 3, 5 ,7, ... hasta que el resultado sea menor que cero.

Entonces, le doy "step" y se modificó de 4C a 4B el valor de EAX, se restó uno.

Luego viene el salto "jump on below" que es lo mismo que "jump on carry" que habíamos compilado nosotros que salta si el resultado es menor que cero. ¿Y eso qué significa? Significa que en ese caso se iría del ciclo de resta que no es nuestro caso porque pasó de 76 a 75 entonces no hay "borrow".

Entonces, sigue con la instrucción a continuación y va a sumar dos al valor de ECX que es 1, va a pasar del primer número impar que es 1 al siguiente número impar que es 3.

Fíjense que siempre el destino va a la izquierda, fuente a la derecha y el resultado se carga en el registro destino que sería en este caso ECX.

Entonces, si ejecutamos "step" se cambia de 1 a 3 el valor de ECX.

Luego, incrementamos la cantidad de restas y fíjense que pasó de cero a uno.

Y luego viene un JMP incondicional para volver al principio del ciclo de restas, con la instrucción que está en la posición 25 que es otra vez el sub que estaba antes.

Entonces, "step" volvió al offset 25.

Fíjense que IP vale 25 y va a hacer la siguiente resta. Va a restar 75 - 3. El resultado va a ser 72. Como se ve acá 48 hexadecimal es 72. Otra vez no se da el "borrow" entonces, continúa dentro del loop y así sucesivamente va cargando el siguiente número impar de 3 pasa a 5 e incrementa nuevamente la cantidad de restas.

De 1 va a pasar a 2. Y así vamos a ver varias veces hasta que eventualmente sale del loop. Lo que vamos a hacer es pasar varias veces hasta lo que sería la última resta que sería por aquí.

Ahí está. Esta es la última resta. En la última resta, el valor de EAX que es 12 es menor que el valor de ECX que es 17.

Entonces acá, una vez que haga la resta el resultado va a ser negativo y por lo tanto el "jump on below" va a salir del ciclo.

Entonces, ejecuto. Fíjense que EAX ahora se hace negativo y el "jump on below" se cumple y sale del loop. Ahora va a la dirección 31 que está fuera del loop.

Y el valor de DX es 8, que es la raíz cuadrada de 76, como corresponde y se va a cargar en la posición 0304.

Entonces le doy "step" y miro con View, Linear Mem dump Vamos a poner la misma dirección que antes, 300. Y ahí se ve moviendo de nuevo el scroll bar se ve el número 76 en la posición 300 y el número 8 en la posición 304. Es una tabla de doble entrada. Bien.

Entonces, una vez que se cargó en memoria el resultado, viene la instrucción RET que es para volver de la subrutina que lo que hace es leer la pila la posición apuntada por 7FFE y va a leer el 17 que estaba guardado ahí adentro y ese número lo va a cargar en el registro IP.

Entonces le damos "step", cargó 17 en el registro IP, y salió de la subrutina.

Fíjense que el stack pointer o sea el puntero de pila, volvió al valor inicial. Siempre tiene que terminar igual que como comenzó. O sea, la pila queda balanceada en este caso.

Una vez que llega este JMP, si yo le doy "step" sigue indefinidamente en el JMP porque el destino del JMP es 17 que es la misma dirección que está el JMP. Entonces indefinidamente va a seguir en 17.

Fíjense los valores de los flags en este momento. Acá tenemos en minúsculas los flags que están a cero y en mayúsculas los flags que están a uno.

Entonces, por ejemplo, en el caso de flag de signo, la última instrucción aritmético-lógica fue la resta que se encuentra en esta dirección y se acuerdan que habíamos restado 12 - 17 que el resultado es -5, que es lo que se ve acá en complemento a dos y el flag de signo repite el bit más significativo del resultado que indica que es negativo, el bit está a uno.

Luego el flag de cero está a cero porque el resultado no es cero.

Después, el carry flag vale uno porque la resta dio overflow con respecto a números no signados porque no hay ningún número no signado que valga -5, por lo tanto se tiene que prender necesariamente el flag de carry.

Entonces, con esto ejecutamos ya el programa completo.

Vamos a hacer una pruebita. Supongamos que salimos de nuevo y entramos otra vez.

Entonces, entramos otra vez, y lo que vamos a hacer ahora es poner un punto de parada.

El punto de parada es para no tener que ejecutar paso a paso quizá cientos o miles de instrucciones y lo que voy a hacer es poner punto de parada después del loop o del ciclo de restas.

Entonces, después del ciclo de restas es cuando se escribe en la memoria el resultado, la raíz.

Entonces, para habilitar el punto de parada lo que tengo que hacer es un doble click en la dirección. Cuando hago doble click se me pone la instrucción en rojo y en itálica. Ya con eso sé que tiene un breakpoint habilitado.

Entonces, yo le doy "continue", esto está todo a cero porque recién arranca. Le doy "continue" ¿Y qué pasó? Aparece la misma información que estaba antes EAX vale -5, ECX 17 que era el último número impar que hizo la resta, 8 era la raíz cuadrada, ESP vale 7FFE porque todavía no salió de la subrutina y EFLAGS vale lo que dijimos antes.

Después se pueden ver más cosas, por ejemplo, se pueden ver el contenido de los registros usando "reg". Parte del panel de la izquierda lo puedo ver en formato texto, si yo quiero copiar la información de los registros y pegarlo en algun editor de texto, lo puedo hacer.

Estos breakpoints son útiles si queremos para en una dirección cercana a la que estamos ejecutando pero si nosotros tenemos un código muy largo el problema que se plantea es que no sabemos donde poner el punto de parada.

Entonces, hay un método adicional que permite el Bochs que se llama "magic breakpoints". Lo que vamos a hacer es: paramos el emulador y luego editamos el archivo de configuración del Bochs y buscamos la palabra "magic" y en este caso está habilitado. Podría estar deshabilitado. Hay que asegurarse que no esté el numeral puesto adelante. En este caso el magic breakpoint está habilitado. Bien.

¿Cómo funciona esto? Hay que editar el programa donde queremos poner el breakpoint y vamos a suponer que queremos poner el breakpoint antes y después del ciclo de restas.

Voy a apretar F11 y con F11 veo los números de línea.

El ciclo de restas está entre la línea 28 y la línea 32 es el ciclo de restas.

Entonces, agrego una instrucción xchg bx, bx es un magic breakpoint.

Y luego voy a poner antes del ciclo y luego pongo otro después del ciclo otro magic breakpoint. Entonces, la idea de esto que se pueda parar la ejecución del código cuando se llega a un magic breakpoint.

Salimos de acá, salvamos, salimos, compilamos y luego ejecutamos el Bochs de nuevo. Y está otra vez el Bochs.

Si yo le doy "continue" va a ejecutar hasta el primer magic breakpoint.

Entonces, fíjense que ahora estamos en la resta que es donde comienza el ciclo.

El registro ECX está con el primer número impar, EAX vale 76 que es el valor que queríamos probar, y ahora le damos "continue" de nuevo.

Se paró la ejecución del código justo después del magic breakpoint, que es donde se va a grabar el resultado final que es la raíz se graba la cantidad de restas menos uno que está almacenado en DX.

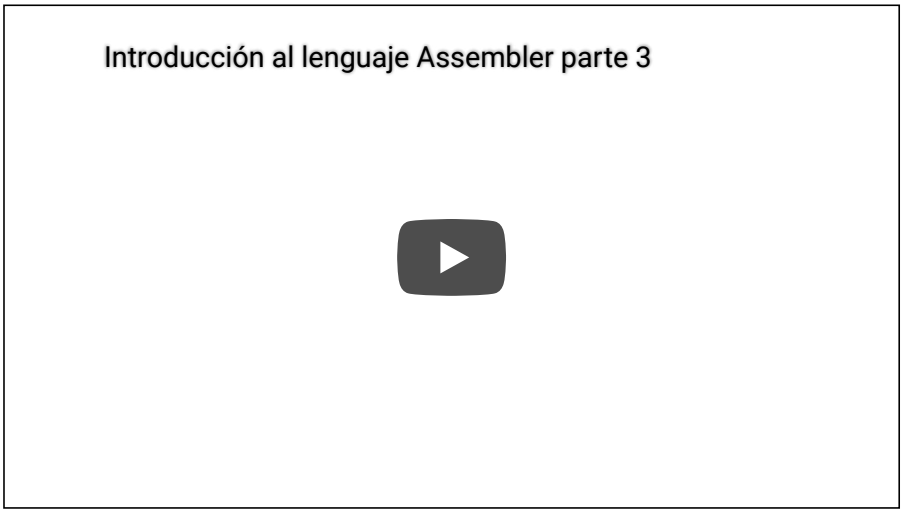
Entonces ahí se puede ver que paró en ambos magic breakpoints que nosotros habíamos puesto.

Si le damos "continue" y paramos con "break" queda en el JMP que habíamos puesto para que se termine colgando el código Se puede seguir con más información como poner breakpoints usando la consola o bien, hacer más cosas pero por hoy es suficiente con todo lo que vimos hasta ahora.

Bueno, hasta luego.

Introducción al lenguaje Assembler parte 3

por Dario Alejandro Alpern



Archivos necesarios

- sqrt2.asm

Transcripción

Hola. Mi nombre es Darío Alpern y hoy vamos a ver la tercera parte de Introducción a Assembler de procesadores Intel

En este segundo programa, vamos a ver más instrucciones de Assembler.

Raíz cuadrada usando búsqueda binaria

Queremos hacer otra subrutina de raíz cuadrada, pero esta vez usando búsqueda binaria.

Como el radicando tiene 32 bits, sabemos que la raíz cuadrada va a tener 16 bits. La idea es dividir este rango de 65536 valores en dos partes iguales y determinar si la raíz cuadrada está en la parte inferior o en la superior.

Para ello calculamos el cuadrado del valor que está justo en la mitad 32768 en nuestro caso y lo comparamos contra el radicando. Como el radicando es mayor que el cuadrado, entonces la raíz cuadrada está en la mitad superior.

En el segundo paso calculamos el cuadrado del valor que está en la mitad de este rango, en nuestro caso 49152 y según la comparación, sabremos si está en la mitad inferior (rango 32768 a 49151) o en el superior (49152 a 65535). En nuestro caso está en el rango inferior.

Lo mismo ocurre en el tercer paso. Así seguimos un total de 16 veces hasta agotar los rangos.

Cuando termina este ciclo, ya tenemos la raíz cuadrada.

Mapa de memoria

Otra modificación que vamos a hacer al programa es que en vez de correr de memoria no volátil, vamos a correrlo desde RAM. Para eso, vamos a copiar el programa principal a la dirección lineal 0x20000 y la subrutina a la dirección lineal 0x50000. Luego, saltamos al programa principal en RAM.

A la izquierda, se pueden ver los 64 KB de ROM que arranca con la rutina de copiar de ROM a RAM. Luego, el programa principal inicializa la pila y el valor del radicando en memoria para luego llamar a la subrutina. Finalmente, va a hacer un salto a la misma dirección para colgarse.

La subrutina va a contener el código de la raíz cuadrada que lee el radicando de memoria, ejecuta la búsqueda binaria, y almacena el resultado en memoria.

Luego viene el relleno, la cantidad de bytes necesarias para que el JMP esté en el offset 0xFFF0 y finalmente el relleno para completar los 64 KB.

A la derecha, se puede ver el contenido de la RAM una vez efectuada la copia.

Como ya sabemos, en modo real la base del segmento se calcula como el selector multiplicado por 0x10 Despejando el selector, resulta que el selector es igual a la base del segmento dividido 0x10.

Por lo tanto, el programa principal deberá estar en un segmento de código cuyo selector valga 0x2000 mientras que la subrutina estará en otro segmento de código cuyo selector valga 0x5000.

Copia de datos usando instrucción de cadena

Ahora vamos a ver una instrucción de assembler que es bastante potente, porque es equivalente al memcpy en lenguaje C.

Antes de ejecutar la instrucción, hay que cargar sus parámetros en determinados registros. Los registros DS:SI apuntan al buffer fuente, los registros ES:DI apuntan al buffer destino, El registro CX indica la cantidad de bytes a copiar y finalmente el flag de dirección vale cero o uno si las direcciones crecen o decrecen con la ejecución de la instrucción.

Entonces, hay que ejecutar una de las dos instrucciones CLD o STD y luego REP MOVSB.

La letra B se puede reemplazar por W o por D para copiar words, que son bloques de 16 bits, o dwords, que son bloques de 32 bits.

También, se puede reemplazar el registro DS correspondiente al buffer fuente por cualquier otro registro de segmento. Para eso, se especifica dicho registro entre el REP y el MOVSB.

El registro ES correspondiente al buffer destino, no se puede modificar.

Explicación del programa

Ahora veremos el programa.

Bien, vamos a editar el programa. Acá está la explicación del programa que estuvimos diciéndolo. Después, las direcciones radicando y raíz son las mismas que antes, Luego vienen los selectores ya explicados del programa principal y la subrutina donde van a estar ubicados en 0x20000 para el programa principal y 0x50000 para la subrutina. Esos van a ser los valores de las direcciones lineales.

Luego viene el inicio, y en inicio comienza la rutina de copia.

Entonces, la primera copia se efectúa la copia del programa principal a la RAM. Para ello, primero cargamos en el extra segment el SEL_PROG_PRINCIPAL o sea, 0x2000. Después, en el puntero fuente que es SI, se carga el puntero al inicio del programa principal que se puede ver que está más abajo O sea, lo que yo quiero copiar (si aprieto F11 veo números de línea), lo que quiero copiar está en la línea 37 a la 45 y ahí se ve, fuera del azul (de la selección), se ve inicio_prog_principal y fin_prog_principal, líneas 36 y 46 respectivamente, Y yo quiero copiar ese rango a RAM Entonces, pongo en el registro fuente el puntero a inicio_prog_principal.

Después, el offset de destino es cero. Luego la cantidad de bytes es fin_prog_principal - inicio_prog_principal, la diferencia entre los offsets, y luego efectúo la copia pero observando que el registro de segmento fuente es el code segment. que es el que contiene el código de programa.

Luego hacemos lo mismo con la subrutina, Como selector en vez de SEL_PROG_PRINCIPAL usamos SEL_SUBRUTINA Después en el registro fuente usamos inicio_subrutina como puntero. Luego offset destino (también DI) lo ponemos a cero, y finalmente tenemos el contador de cantidad de bytes como fin_subrutina - inicio_subrutina. En este caso como podemos ver más abajo, vamos a copiar desde inicio_subrutina hasta fin_subrutina exclusive, o sea toda esta cantidad de bytes en azul. Todo eso va a ir a parar a RAM. Luego de ello, hacemos otra vez REP CS MOVSB, y hacemos el segundo memcpy.

Finalmente saltamos a RAM con un formato de JMP intersegmento donde se cambia el code segment y el instruction pointer. Entonces, el valor de la izquierda es el que va a parar a CS y el de la derecha es el offset que va a parar a IP. Esto va a hacer un salto a la dirección 0x20000.

El programa principal, que es lo que vemos acá a continuación, que es más o menos parecido a lo que habíamos visto en el programa anterior, que inicializaba el puntero de pila. Ahí inicialzamos el segmento de datos. Cargamos con 1000000 el valor del radicando en vez de 76. para hacer un número diferente, y luego llamamos a la subrutina con un CALL intersegmento De nuevo se modifica CS y EIP. Acá es necesario un CALL intersegmento porque estamos yendo a una zona de memoria completamente diferente del programa principal. Luego de eso está la rutina de colgarse.

¿Cómo funciona la raíz cuadrada mediante la búsqueda binaria? Leemos el radicando, luego inicializo la posible raíz cuadrada a cero. Verifico si radicando vale cero. Si es así, salto porque es el valor correcto. Y si no es cero tengo que hacer la búsqueda binaria.

Entonces, cargo el delta en la mitad del rango, o sea, 0x8000. y después en la búsqueda binaria lo que hago es sumar el principio del rango que está en SI más el delta, que es DI, y eso va a parar a EAX.

Luego, multiplico EAX por sí mismo y eso me reemplaza EAX por el cuadrado de EAX. Entonces, en EAX tengo el cuadrado de la mitad del rango.

Luego comparo el radicando contra el cuadrado. La instrucción CMP hace lo mismo que SUB pero sin reemplazar el destino. Lo único que hace es modificar los flags. Y entonces, ya con los flags modificados puedo saltar si es menor o igual, "jump on below or equal".

Si es menor o igual, salteo la suma del delta. como habíamos dicho anteriormente con la búsqueda binaria. En caso contrario sumo delta a la posible raíz porque eso significa que estamos en la mitad superior. Si saltamos sin sumar es porque estamos en la mitad inferior.

Luego de eso, una vez que ya sumamos o no, sabemos que estamos en la mitad superior o inferior, lo que hay que hacer con delta es dividirlo por 2. O sea, de 0x8000 debe pasar a 0x4000. Para eso hacemos una instrucción "shift right", SHR, donde el primer registro es el destino que sería DI en este caso y acá está la cantidad de bits que yo quiero shiftear que es 1. Entonces, shifteo a la derecha un lugar que significa dividir por 2.

Si no es cero, significa que todavía tenemos rango para comparar si es menor o mayor Si no es cero vuelvo otra vez al ciclo de búsqueda binaria, y si es cero, se terminó el loop, pero el SI que era lo que yo estaba buscando me quedó un número de menos entonces lo incremento para ajustarlo.

Y acá tengo un "magic breakpoint", se acuerdan, con xchg bx,bx y luego cargo el resultado con MOV [RAIZ],SI Eso guarda la raíz cuadrada en la posición de memoria que esperamos nosotros, que es la 0x304.

Y luego, en vez de RET, tenemos la instrucción RETF, que es un retorno de subrutina lejano, FAR. Eso se usa cuando hay un CALL intersegmento hay que aparearlo con RETF (ret far). Y con eso terminamos la subrutina.

Después hacemos lo mismo que con el programa anterior relleno, el JMP del reset, y después el otro relleno y eso es todo.

Ejecución del programa en Bochs

Bueno, ahora vamos a correr este programa. Para eso primero compilamos, La salida primero puse sqrt.bin en vez de sqrt2 así aprovechamos el seteo del bochs.cfg, así no lo modifico. Corremos el Bochs. y ahí arrancó, bien.

Saltamos en el reset inicial y estamos al principio de la zona de copia.

Cargo 0x2000 en ES. Si nosotros vemos con sreg, que son los registros de segmento y vamos al ES podemos ver que se cargó el selector con 0x2000 y la base se carga automáticamente, como estamos en modo real ven que acá dice Real Mode, se carga con 0x20000, que es 0x2000 por 0x10.

Luego tenemos el puntero fuente que es 0x27. Ese 0x27 si ustedes miran más abajo es justamente donde arranca el programa principal. Está acá, ¿no? Que es cuando se inicializa la pila y diferentes cosas. El 0x27 ya está.

Después si ejecutamos un "step" más. Este es el puntero destino, que se pone a cero, cantidad de bytes 0x1B y después hacemos el REP MOVSB que lo que hace es hacer el memcpy.

Si yo hago "step" varias veces, se queda ahí incrementando, como pueden ver en rojo las direcciones porque hace una copia por vez. Entonces, para hacer todas las copias juntas, pongo un punto de parada en la instrucción siguiente y le doy "continue" y ahí copió todo.

Después de eso, hacemos lo mismo con la subrutina. Vamos directamente a poner un punto de parada y copió también la subrutina en RAM.

Y ahora lo que vamos a hacer, es hacer el salto lejano al otro segmento de código, a la dirección 0x2000:0x0000 Entonces, cuando le doy "step", ahí se puede ver que se modificó el CS a 0x2000 y el IP se modificó a cero. Y estamos en la dirección lineal 0x20000.

Y entonces, ahora hacemos "step" "step", "step", "step" Ahí se cargó con 1000000 el valor del radicando.

Y acá viene el call intersegmento a la dirección 0x5000:0x0000. Acá se va a cargar 0x5000 en CS y 0x0000 el IP. Pero como es un CALL se va a cargar la dirección de retorno en la pila, entonces ahora vamos a mirar qué hay en la pila.

Ejecuto "step", y vemos que la pila en vez de ir a 0x7FFE como la vez pasada, ahora va a 0x7FFC O sea hay cuatro bytes en vez de dos. Vamos a ver qué hay en los cuatro bytes. Ponemos View > Linear Memdump. Entonces, vamos con el scroll hacia la derecha, y ahí vemos que se grabaron cuatro bytes. Primero siempre se graba el offset. El offset es 0x19 y luego va el selector que es 0x2000. Acuérdense que están al revés los bytes de lo que uno espera porque están en "little-endian". Entonces, el offset es 0x19 y el selector 0x2000, que es la dirección de retorno.

Entonces, ahora EBX se cargó con el valor 0xF4240 que es 1000000. Después, si se carga con el valor supuesto de la raíz cuadrada, que obviamente es cero por ahora.

Vemos si EBX vale cero. Obviamente no es. Entonces el flag de cero es cero. Significa que el resultado no es cero.

"step" y no saltó, obviamente.

Ahora DI es el delta, que se carga con la mitad del rango, que es 32768.

"step" Y ahora lo que hago es cargar EAX luego de la suma. Fíjense que EAX vale 0x8000 que es justamente la mitad del rango.

Entonces hago "step". Acá hizo la multiplicación: 0x8000 por 0x8000 da 0x40000000. Luego de eso, Comparamos EBX que es el radicando contra EAX que es el cuadrado Y acá cambia los flags según corresponde y va a saltar si es menor o igual EBX con respecto a EAX. y EBX es menor que EAX como se ve acá. O sea, un millón es menor que 1600 millones, más o menos, que es lo de arriba.

Entonces, ¿qué va a pasar? Va a pasar que no suma el delta porque estamos en la mitad inferior.

Luego de eso, el valor de DI, que es el delta tengo que dividirlo por 2 con la instrucción SHR DI,1 shiftea un lugar a la derecha. y de 0x8000 pasó a 0x4000. o sea, dividió por 2.

Obviamente 0x4000 no es cero, entonces vuelve y ejecuta de nuevo el loop así 16 veces y para no aburrirnos y hacerlo 16 veces voy a hacer "continue" hasta el xchg bx,bx que es el "magic breakpoint". Ahí ejecutó hasta el xchg bx, bx y muestra la instrucción siguiente a ejecutar.

Y fíjense que SI vale 0x3E8 que, si yo muevo el cursor a la derecha, fíjense que es el valor 1000, que es la raíz cuadrada de un millón.

O sea quiere decir que el código parece andar bien.

Ahora guardo el valor 1000 dentro de la memoria Y ahora de nuevo nos vamos a asegurar que haya grabado todo bien. View > Linear Memdump y acá se puede ver los cuatro bytes del radicando, que siempre se leen de derecha a izquierda, y la raíz es 00 00 03 E8, que es 1000, como vimos recién.

Y luego va a ejecutar RETF. RETF lee el valor que está en la pila o sea 0x7FFC y habíamos visto que en 0x7FFC teníamos el selector a 0x2000, vamos a ver de nuevo. View > Linear Memdump y acá ponemos 0x7FFC Y acá se ve: el selector está a 0x2000 y el offset 0x19. Entonces cuando yo le doy "step" pasó CS a 0x2000 e IP a 0x19, que es lo esperado y fíjense que SP volvió a 0x8000 o sea, que estamos con el stack balanceado que es lo que esperamos y salta, justamente, a la instrucción JMP donde se cuelga.

Bueno, creo que con esto ya tienen otro ejemplo para mirar. Espero que les haya interesado y hasta luego.

Modo largo en procesadores Intel

por Dario Alejandro Alpern

Transcripción

Hola. Mi nombre es Darío Alpern y hoy vamos a ver introducción a modo largo.

Modos de operación

Los procesadores de Intel soportan varios modos de operación. A continuación se verán los más importantes:

El modo heredado es básicamente un conjunto de submodos creados antes que el modo largo.

El modo real es el que corre desde el arranque del procesador, permite direccionar 1 MB y no hay ningún método para proteger el sistema de accesos a memoria fuera de rango o usando punteros nulos.

El modo protegido permite proteger el sistema operativo de las aplicaciones y éstas entre sí. Permite acceder a 4GB y si se activa paginación con PAE, se puede acceder a direcciones físicas por arriba de 4 GB. El modo protegido permite correr sistemas operativos de 16 bits, que son obsoletos hace por lo menos dos décadas, y de 32 bits.

El modo virtual 8086 permite correr aplicaciones compiladas para modo real en sistemas operativos de 32 bits. Dicho modo es una tarea, lo que significa que el sistema operativo puede correr varias sesiones de modo virtual 8086 al mismo tiempo junto con otras aplicaciones de modo protegido de 16 o de 32 bits. El primer uso importante del modo virtual 8086 fue en el sistema operativo Windows 3.0 del año 1990 que en el modo 386 mejorado permitía correr aplicaciones de Windows y de DOS al mismo tiempo.

El modo largo fue creado por la empresa AMD y apareció por primera vez en abril de 2003 en su procesador Opteron, que estaba orientado a servidores. Este modo apareció originalmente en Intel en el modelo de procesador Xeon denominado Nocona, también para servidores, que salió al mercado en junio de 2004. En procesadores para uso personal, esta arquitectura apareció en algunos modelos del Pentium 4 denominado Prescott, que salieron pocos meses después.

Existen diferencias en la implementación del modo largo entre ambas empresas, pero son menores.

El modo largo requiere sistemas operativos de 64 bits, completamente diferentes de los sistemas operativos de 32 bits que corren en modo protegido. Se pueden correr aplicaciones de 64 bits, o bien de 32 o 16 bits en el llamado modo compatibilidad.

Aparte de poder acceder a registros de 64 bits, el código que corre en segmentos de 64 bits tiene a su disposición 16 registros de uso general, 16 registros de SSE, y 16 registros de control en vez de 8, que es lo que se puede acceder en segmentos de código de 16 o de 32 bits.

El modo largo no incluye el modo virtual 8086, por lo que no se pueden correr programas compilados para modo real, como por ejemplo, aplicaciones que corren bajo el sistema operativo DOS.

El modo de gerencia del sistema o SMM se activa mediante la pata SMI del procesador o mediante la instrucción SMI. El procesador almacena su estado en una zona determinada de la memoria y luego ejecuta el manejador de este modo de manera similar a una interrupción. Dentro de este modo, el procesador opera de manera similar al modo real, excepto que no hay verificación de límite, por lo

que se puede acceder a la totalidad del espacio direccionable. La instrucción RSM sirve para salir del modo de gerencia del sistema y se recuperan todos los registros. De esta manera el procesador continúa ejecutando en el modo anterior al ingreso a SMM.

La virtualización permite correr una máquina virtual desde el arranque dentro de un sistema operativo. En este contexto, hay dos sistemas operativos, el anfitrión, que es el que estaba corriendo de antes, y el invitado, que es el que corre dentro de la máquina virtual.

Es posible por ejemplo, correr un sistema operativo de 32 bits dentro de un sistema operativo de 64 bits mediante virtualización, y de esa manera correr aplicaciones antiguas de DOS.

Dos conceptos importantes son los de entrada a la máquina virtual (VM Entry en inglés), en el que el procesador comienza a ejecutar el sistema operativo invitado, y la salida de la máquina virtual (VM Exit en inglés), en el que el procesador deja de ejecutar el invitado y continúa con el anfitrión.

De esta manera se puede usar un driver del sistema operativo anfitrión para interactuar con hardware cuando el sistema operativo invitado necesita accederlo.

Registros de uso general

En estos diagramas se ven los registros de uso general accesibles en segmentos de código de 64 bits. En el modo de compatibilidad, es decir, en 16 o 32 bits, los registros accesibles son los mismos que en el modo heredado.

A la izquierda figuran los 16 registros de uso general de 64 bits. Las extensiones de los 8 registros conocidos de 32 bits reemplazan la letra inicial "E" por la letra "R". Por ejemplo, la extensión a 64 bits del registro EDX es RDX. Los ocho nuevos registros se denominan R8 a R15.

Es posible acceder a los 32 bits menos significativos de estos registros. Por ejemplo, la parte baja de RAX es EAX. En el caso de los registros nuevos, se agrega la letra D, de doubleword, que son 32 bits, al final. Por ejemplo la parte baja de R10 es R10D.

A su vez, los registros de 32 bits se pueden subdividir por la mitad y se puede acceder a la parte baja. En el caso de los registros nuevos, se agrega el sufijo W, de word, que son 16 bits. Por ejemplo, la parte baja de R13D es R13W.

Finalmente, se pueden dividir los registros de 16 bits y acceder a la parte baja, o en algunos casos, a la parte alta de dicho registro. En el caso de los registros nuevos, se agrega el sufijo B, de byte, que son 8 bits. Por ejemplo, la parte baja de R8W es R8B.

Los registros de puntero de instrucciones y los indicadores también se expanden a 64 bits, y sus nombres son RIP y RFLAGS respectivamente.

Prefijo REX

Así como en el procesador 80386 se agregaron los prefijos de tamaño de operandos 0x66 y de tamaño de direcciones 0x67 para que se puedan usar los códigos de operación con registros de 32 bits y con direccionamiento de 32 bits, en el modo de 64 bits hubo que agregar más prefijos para poder acceder a los registros de 64 bits y para poder usar 16 registros en vez de los 8 que se permiten en segmentos de código de 16 y de 32 bits.

Como todos los bytes ya estaban usados para código de operación y los prefijos definidos en el modo heredado, la solución fue eliminar instrucciones de un byte para hacer lugar para estos prefijos.

Instrucciones INC y DEC

De esta manera, en este modo, los bytes 0x40 a 0x47 que correspondían a instrucciones INC de registros y los bytes 0x48 a 0x4F que correspondían a instrucciones DEC de registros, se convierten en los nuevos prefijos.

Esto no quita funcionalidad al procesador porque estas instrucciones de un byte también podían expresarse mediante un código de operación de dos bytes. Por ejemplo, INC EBX, que se podía codificar como 0x43, también podía hacerlo mediante el par de bytes 0xFF 0xC3.

Prefijo REX

Como se puede ver en el diagrama, el prefijo usa cuatro bits independientes: el bit W es el de tamaño de operandos, que define si se usan 32 o 64 bits. En este modo se sigue usando el prefijo 0x66 para indicar que el tamaño de operando es de 16 bits. El acceso a operandos de 8 bits se realiza mediante un código de operación diferente y este tamaño no se puede modificar mediante el prefijo 0x66 o el bit W del prefijo REX.

Luego existen tres bits adicionales que agregan un bit a cada uno de los campos que se encuentran en el código de operación: el número de registro, el registro índice y el registro base. En los tres casos los campos en el modo 64 bits pasan de tres a cuatro bits, siendo el bit definido en el prefijo REX el bit más significativo.

Los registros AH, BH, CH y DH sólo se pueden acceder si la instrucción no tiene prefijo REX. En caso de haberlo, la instrucción accede a SIL, DIL, BPL o SPL, que son la parte baja de SI, DI, BP o SP respectivamente.

Cuando la instrucción accede a memoria en el modo 64 bits, normalmente usa direccionamiento de 64 bits. Si se desea usar direccionamiento de 32 bits, se debe agregar el prefijo 0x67 de tamaño de direcciones.

Ejemplos REX

A continuación veremos varios ejemplos del uso de los prefijos en 64 bits.

En el primer ejemplo vemos el código de operación sin prefijos. En el caso de la instrucción MOV de registros que no sean de 8 bits, se usa el código de operación 89 y luego en el byte siguiente los dos bits más significativos valen 1, a continuación los tres bits del registro fuente (en nuestro caso el registro ECX tiene el valor uno) y finalmente los tres bits del registro destino (en nuestro caso EAX tiene el valor cero).

En el segundo ejemplo, lo único que modificamos es el tamaño de operandos, que pasamos de 32 a 64 bits, por lo que necesitamos un prefijo REX donde el bit W valga uno y el resto de los bits cero.

En el tercer ejemplo, usamos operandos de 32 bits, pero el registro destino es uno de los nuevos. Así que necesitamos prefijo REX. En el caso de la instrucción MOV de dos registros, el registro destino se codifica en el campo que se usa para la base cuando usamos acceso a memoria, mientras que el registro fuente se codifica en el campo que se usa para el registro cuando accedemos a memoria. El registro R8D tiene el código 1000 binario, por lo que deberemos poner el bit B a uno y el campo base ("b b b" en el diagrama) a cero. De esta manera tenemos prefijo REX y el mismo código de operación que en los dos ejemplos anteriores.

En el cuarto ejemplo, tenemos operandos de 16 bits, por lo que agregamos el prefijo 0x66 al primer ejemplo.

El quinto ejemplo es similar al tercero, pero usa operandos de 64 bits, así que en el prefijo REX ponemos a 1 el bit W aparte del bit B que ya estaba puesto a 1 en el tercer ejemplo.

En el último ejemplo, modificamos el ejemplo anterior para que el registro fuente sea R9, que tiene el valor 1001. Como dijimos que el registro fuente se codifica en el campo registro ("r r r" en el diagrama), debemos poner a 1 el bit R.

Ejemplos REX de acceso a memoria

Ahora veremos algunos ejemplos de acceso a memoria.

El formato de la instrucción para cargar un registro que no sea de 8 bits desde la posición de memoria apuntada por otro registro de 32 o 64 bits no es muy diferente a lo que vimos antes: el código de operación es 8B, en el byte siguiente los dos bits más significativos valen cero, luego siguen los tres bits que identifican el registro destino y finalmente los tres bits que identifican el registro base.

En el primer ejemplo, usamos operandos de 32 bits, direccionamiento de 64 bits y ningún registro nuevo, así que no hace falta el prefijo REX.

Como el direccionamiento por defecto es de 64 bits, en el segundo ejemplo necesitamos el prefijo 0x67 para poder usar direccionamiento de 32 bits.

En el tercer ejemplo, el registro es uno de los nuevos y además los operandos tienen 64 bits así que el bit W debe valer 1 y el bit R también.

En el cuarto ejemplo, usamos uno de los registros nuevos como registro base, así que ponemos a uno el bit B.

En el último ejemplo, usamos un registro nuevo y usamos direccionamiento de 32 bits, por lo que necesitamos el prefijo 0x67 para direccionar 32 bits y ponemos el bit R a 1 para indicar que el campo registro tiene un registro nuevo.

En estos ejemplos no se ve ningún caso en el que el bit X esté encendido. Eso requiere direccionamiento indirecto en el que se encuentren registros base e índice, donde el registro índice sea uno de los nuevos registros.

Descriptores de código y datos

En este diagrama se pueden ver los descriptores de código y datos que se usan en el modo largo.

El formato es muy similar al usado en el modo protegido. Al igual que en dicho modo, el byte de derechos de acceso, que es el byte 5, se intepreta diferente según el tipo de descriptor.

La diferencia más significativa se encuentra en el byte 6, que incluye los campos D y L. Según el contenido de estos dos bits, se puede determinar si el segmento es de 16, de 32 o de 64 bits.

El manejo de la base y el límite en segmentos de 16 y de 32 bits es igual que en el modo protegido. El manejo en 64 bits se verá más adelante.

Dirección canónica

La dirección canónica es una dirección lineal en el modo 64 bits, en donde los bits 63 a 48 son iguales al bit 47.

Si el código desea acceder a una dirección no canónica, el procesador genera una excepción 13.

Model Specific Register

Estos registros, que aparecieron originalmente en el procesador Pentium en 1993, tienen varias funcionalidades y en cada generación se agregan nuevos MSR. También existen MSR que dejan de existir en procesadores más modernos. El fabricante de procesadores señaló algunos MSR como permanentes y garantiza su existencia a futuro. Esos MSR se denominan de arquitectura.

Un ejemplo de MSR de arquitectura es el Time Stamp Counter (TSC) que arranca en cero cuando se enciende el procesador y se incrementa por cada ciclo de reloj.

La longitud de los MSR es de 64 bits y se acceden mediante un índice de 32 bits.

Las instrucciones de lectura y escritura de MSR se denominan RDMSR (de Read Model Specific Register) y WRMSR (de Write Model Specific Register). Dichas instrucciones sólo se pueden ejecutar con CPL = 0. El índice debe grabarse en el registro ECX y el resultado de la lectura se almacena en EDX:EAX.

La gran mayoría de la nueva funcionalidad se obtiene accediendo a MSR.

Segmentos de 64 bits

En segmentos de 64 bits, el procesador ignora el límite. La única validación que hace es que la dirección lineal tenga formato canónico.

Con respecto a la base, para los registros CS, DS, ES y SS también la ignora. En cambio, en el caso de los registros FS y GS, el procesador carga en el campo base de estos registros lo que esté indicado en el descriptor.

Como no se pueden representar direcciones lineales más grandes que 4 GB en los descriptores de datos, el procesador cuenta con dos MSR que permiten cargar cualquier valor de 64 bits en la base de estos registros, siempre que estén en formato canónico.

Los MSR involucrados son MSR_FS_BASE, que tiene índice 0xC0000100, y MSR_GS_BASE, que tiene índice 0xC0000101.

Además existe una instrucción que se llama SWAPGS, que intercambia la base del registro GS con el MSR que tiene el nombre MSR_KERNEL_GS_BASE y cuyo índice es 0xC0000102.

Compuertas

Las compuertas contienen direcciones lógicas, es decir selectores y offsets. Como el offset tiene 64 bits, necesita 8 bytes. Junto con los dos bytes del selector, se excede el tamaño de 8 bytes por descriptor.

Por ello se usa la siguiente potencia de 2, es decir 16 bytes como tamaño de las compuertas.

En este diagrama se puede ver el formato de las compuertas de llamada que se encuentra en la GDT y las de interrupción y excepción que se encuentran en la IDT.

Una diferencia de las compuertas entre el modo largo y el modo protegido, aparte del tamaño, es que en el caso de la compuerta de llamada no existe el campo de cantidad de palabras que permite copiar los argumentos de una pila a otra de mayor privilegio.

Otra diferencia es que las compuertas de interrupción y excepción tienen un nuevo campo, denominado IST (Interrupt Stack Table), que permite que un manejador de interrupción o excepción use una pila diferente que la normal.

Paginación

Una condición necesaria para poder ingresar al modo largo es que esté habilitada la paginación con PAE encendido.

La jerarquía de paginación en modo largo tiene cuatro niveles, a diferencia de modo protegido donde hay dos o tres.

Para poder realizar la traducción de direcciones lineales a físicas, la dirección lineal se divide en seis campos.

Los bits 63 a 48 no entran en la traducción al ser canónicas las direcciones lineales válidas. Entonces como tienen el mismo valor que el bit 47, no agregan información.

Los bits 47 a 39 corresponden al índice en la tabla PML4.

Los bits 38 a 30 de la dirección lineal corresponden al índice de la tabla de punteros de directorio de páginas.

Los bits 29 a 21 corresponden al índice del directorio de pagínas.

Los bits 20 a 12 corresponden al índice de la tabla de páginas.

Los bits 11 a 0 no se traducen y forman el offset dentro de la página.

El registro CR3 apunta a la tabla de mayor jerarquía, que es la tabla PML4.

En modo largo existen páginas grandes de 2 MB y de 1 GB.

Pila

Como en el caso de modo protegido, hay una pila por nivel de privilegio.

El puntero de pila siempre debe ser múltiplo de la cantidad de bits del segmento de código. Esto quiere decir que para segmentos de 32 bits, ESP debe ser múltiplo de 4 y para segmentos de 64 bits, RSP debe ser múltiplo de 8.

El nivel de privilegio de la pila siempre es igual al nivel de privilegio del segmento de código. Una ventaja en modo largo con respecto al modo protegido es que no hace falta crear descriptores de datos para las pilas de mayor privilegio.

Existen dos casos principales cuando hay llamadas intersegmento, interrupciones o excepciones: si hay cambio de nivel de privilegio o no.

En el segundo caso, que es más sencillo, el procesador memoriza en registros temporarios los valores de SS y RSP, luego modifica RSP para que sea múltiplo de 16 y a continuación almacena los valores de SS y RSP viejos, RFLAGS en el caso de interrupciones y excepciones, CS y RIP. Para determinadas excepciones, luego se coloca en la pila el código de error. Todas las entradas son de 8 bytes, incluyendo los selectores, donde se ponen los seis bytes más significativos a cero y los dos menos significativos con el valor del selector.

La modificación de RSP es importante en el caso de llamar a un segmento de código 64 bits desde otro de 32 bits. De esa manera nos aseguramos que RSP esté alineado como corresponde.

Cuando hay cambio de nivel de privilegio, se debe cambiar la pila.

El offset de la nueva pila se obtiene de campos dentro de la estructura del segmento de estado de la tarea (TSS o Task State Segment). Si la compuerta es de llamada o el campo IST de la compuerta vale cero, el offset de la nueva pila sa obtiene del campo RSPn de la TSS. En cambio, si el campo IST no es cero, ese campo es un índice a un array de punteros de pila ISTn que se encuentra en la TSS.

El procesador carga el selector de SS con el valor cero y atributo de nivel de privilegio con el del nuevo CPL.

Una vez hecho esto, se cargan los registros vistos anteriormente en la nueva pila.

El campo IST es útil por ejemplo para la excepción de fallo de pila ya que permite usar una pila nueva que no debería causar una nueva excepción.

TSS

En el diagrama se puede ver el segmento de estado de la tarea o Task State Segment (TSS) con los campos RSPn e ISTn ya mencionados.

En el caso de cambio de nivel de privilegio, cuando ocurre una llamada intersegmento, una interrupción o una excepción, el procesador cambia a otro segmento de mayor privilegio, es decir menor numéricamente. Por eso no existe el campo RSP3.

En el caso de ejecutar una instrucción de entrada/salida tal como IN o OUT, el procesador compara CPL contra el indicador IOPL ubicado en el registro RFLAGS. Si CPL es menor o igual que IOPL, entonces se ejecuta la instrucción. Por ejemplo, si CPL vale cero, siempre se puede ejecutar IN o OUT. SI CPL es mayor que IOPL, el procesador consulta el bitmap de entrada/salida, que es un array de bits que permite determinar si se ejecuta la instrucción: si el bit vale cero, se puede ejecutar, y si vale uno o el límite de la TSS es menor que el offset donde debería leerse el bitmap de entrada/salida ocurre una excepción 13.

Task Register

El formato del Task Register es el mismo que el de los registros de segmento. El selector apunta a un descriptor de TSS que se encuentra en la GDT.

El descriptor de TSS tiene un bit B que indica si el procesador está usando el TSS apuntado por el descriptor o no.

El TSS debe necesariamente estar por debajo de los 4 GB, ya que el descriptor tiene 32 bits para la base.

Lectura TR

Existen dos instrucciones para acceder al Task Register.

Para escribirlo usamos LTR, que significa Load Task Register. Sólo se puede ejecutar con CPL = 0 y carga el selector de registro de tarea con el valor indicado en el argumento de la instrucción.

Para leer el selector de TR usamos STR, que significa Store Task Register. Dicho selector se almacena en el registro o posición de memoria indicado en el argumento de la instrucción.

Sólo se puede ejecutar LTR con un selector cuyo descriptor apunte a un TSS libre. Luego de la instrucción queda ocupado. Como el procesador escribe el bit B del descriptor, es necesario que la GDT esté en memoria RAM.

Registro EFER

EFER es la sigla de Extended Feature Enable Register o registro de habilitación de características extendidas. Es un MSR en el que cada bit habilita una característica diferente. En ese sentido es similar a los registros de control CR0 y CR4.

Los bits que nos interesan a nosotros son el bit 8, que habilita el modo largo, el bit 10, que sirve para saber si el procesador se encuentra en dicho modo y el bit 11, que es el bit de habilitación de atributo de página No execute (NX).

El bit 10 es necesario porque para entrar en modo largo, aparte de habilitar el bit 8, hay que habilitar la paginación con PAE encendido.

Ingreso a modo largo:

Aquí se enumeran los pasos necesarios para entrar a modo largo.

Los pasos son:

Deshabilitar interrupciones.

Deshabilitar paginación si estaba activada.

Cargar tablas de paginación con los cuatro niveles de jerarquía verificando que el código que pase a modo largo esté en identity mapping tanto en modo protegido como en modo largo.

Poner a "1" el bit 5 del registro de control CR4 (PAE).

Cargar CR3 con la dirección inicial de la tabla PML4.

Poner a "1" el bit 8 de EFER.

Habilitar paginación y modo protegido (si no estaba antes) poniendo a "1" los bits 0 y 31 del registro de control CR0.

En este momento el procesador está en modo compatibilidad. Hacer salto intersegmento a segmento de código de 64 bits definido en la GDT.

Si se desea se puede poner la dirección inicial de GDT o LDT por arriba de los 4 GB una vez que se ingresa al modo 64 bits ya que el argumento de las instrucciones LGDT y LIDT apunta a la imagen de GDT o IDT compuesto por dos bytes de límite y ocho de dirección lineal.

Espero que les haya sido de interés. Hasta luego.

Multitarea en procesadores Intel de 32 bits

por Dario Alejandro Alpern

Transcripción

Hola. Mi nombre es Dario Alpern y hoy vamos a ver multitarea en procesadores Intel de 32 bits.

Esquema de un sistema operativo

Para entender cómo funciona la multitarea, veremos un esquema genérico de sistema operativo.

Un sistema operativo maneja el hardware de las computadoras, como la memoria, los discos, teclado, mouse y otros periféricos, y también recursos de software.

De esta manera, si un programa necesita, por ejemplo, imprimir un documento, no maneja directamente la impresora, sino que a través de órdenes dadas al kernel del sistema operativo, éste encola el trabajo, para que varios programas puedan acceder a la impresora. Estos comandos siguen un protocolo denominado API (Application Programming Interface), que depende del sistema operativo específico que estemos usando.

La comunicación entre el kernel y el hardware se realiza mediante drivers, que es un software que permite abstraer el tipo de hardware. Por ejemplo, el kernel maneja todas las impresoras de la misma manera, y el driver permite acomodar las diferencias que existen entre los comandos de bajo nivel que aceptan las distintas impresoras.

Una aplicación es un programa o conjunto de programas que realiza tareas útiles para el usuario. Ejemplos de aplicaciones pueden ser un procesador de textos, un navegador Web o un juego.

Un programa es un conjunto de instrucciones que realiza una tarea determinada que puede ser simple o compleja. Un proceso es una instancia de programa en ejecución. Por ejemplo, nosotros podemos abrir tres calculadoras en la pantalla, lo que significa que hay tres procesos corriendo que corresponden al mismo programa.

Por lo tanto se puede ver que a una aplicación le corresponde uno o más procesos.

Los threads permiten correr procesos de manera concurrente. Muchos algoritmos se pueden ejecutar en paralelo acelerando su ejecución. Los threads comparten memoria y otros recursos, a diferencia de los procesos que son independientes entre sí.

Estos threads pueden estar despiertos cuando están haciendo algo, o bien pueden estar dormidos, cuando esperan por entrada de teclado o mouse u otro periférico. En el ejemplo de la calculadora, la mayor parte del tiempo los threads están dormidos, hasta que el usuario aprieta un botón y el proceso muestra el dígito ingresado o realiza el cálculo pedido. Luego se vuelve a dormir.

Un sistema operativo multitarea es capaz de ejecutar durante una fracción de segundo cada thread que esté despierto. Así se van turnando la ejecución todos los threads despiertos dando la impresión que los threads están corriendo simultáneamente.

Pero en realidad en un momento determinado, el procesador está ejecutando un thread específico.

Para que al usuario le dé la impresión de ejecución simultánea, el ciclo de ejecución de todos los threads que estén despiertos debe durar menos de una décima de segundo. En Linux los pasajes se hacen 100 veces por segundo.

Desde el punto de vista del procesador todos estos threads son tareas, que es un código que se ejecuta asincrónicamente con respecto a otras tareas.

Se debe tener en cuenta que la misma tarea puede estar ejecutando código de aplicación con CPL=3 o de kernel con CPL=0, ya que la aplicación puede hacer una llamada al sistema provocando un cambio de nivel de privilegio.

En este video vamos a ver cómo el procesador ayuda al sistema operativo en el cambio de tareas, es decir, pasar de una tarea despierta a la siguiente tarea despierta que debe correr.

Task State Segment (TSS)

En el diagrama se puede ver el segmento de estado de la tarea o Task State Segment (TSS) con los campos SSn y ESPn, registros del procesador incluyendo registros de uso general, de segmentación, LDTR, EFLAGS, el puntero de instrucciones EIP y el puntero a las tablas de paginación que es el registro de control CR3.

En el caso de cambio de nivel de privilegio, cuando ocurre una llamada intersegmento, una interrupción o una excepción, el procesador cambia a otro segmento de mayor privilegio, es decir menor numéricamente. Para ello se usan los campos SSn y ESPn, que son las direcciones lógicas de los punteros de pila de los niveles de privilegio cero, uno o dos.

Los campos con nombres de registros se utilizan para almacenar los registros del procesador cuando ocurre un cambio de tarea automático (que se verá más adelante), ya que debe recuperarlos cuando vuelve a ejecutar la tarea. Este esquema requiere que haya un TSS por cada tarea.

Con respecto a los registros de segmentación, sólo se salvan en la TSS los selectores. De esta manera, cuando el procesador deba recargar los registros de segmento desde la TSS, si el valor del selector no es cero, habrá una lectura de GDT o de LDT por cada registro de segmento y una lectura de GDT para el registro LDTR.

El campo bitmap de entrada/salida se verá a continuación.

Bitmap de entrada/salida

En el caso de ejecutar una instrucción de entrada/salida tal como OUT o IN, el procesador compara el nivel de privilegio del código CPL contra el indicador IOPL ubicado en los bits 13 y 12 del registro EFLAGS. Si CPL es menor o igual que IOPL, entonces se ejecuta la instrucción. Por ejemplo, si CPL vale cero, siempre se puede ejecutar OUT o IN. Si CPL es mayor que IOPL, el procesador consulta el bitmap de entrada/salida, que es un array de bits que permite determinar si se ejecuta la instrucción: si el bit vale cero, se puede ejecutar, y si vale uno o el límite de la TSS es menor que el offset donde debería leerse el bitmap de entrada/salida ocurre una excepción 13.

Ese bitmap de entrada/salida se encuentra en el offset de la TSS indicado en el campo Bitmap E/S que es un campo de dos bytes que está en el offset 0x66 de la TSS.

Ejemplo bitmap de entrada/salida

En este ejemplo vamos a suponer que CPL=3 e IOPL=0, queremos ejecutar la instrucción IN AL,0x60, el campo bitmap de entrada/salida de la TSS vale 0x100 y el límite de la TSS vale 0x200.

Como CPL es mayor que IOPL, hay que consultar el bitmap.

El offset de la TSS correspondiente al byte del bitmap donde el procesador debe leer se calcula sumando al campo bitmap el número de puerto dividido 8. El cálculo es 0x100 + 0x60 / 8, así que deberá leer el offset 0x10C.

El offset calculado es menor o igual que el límite de la TSS, así que el procesador leerá el offset indicado.

Suponiendo que leyó el byte 0x46, el procesador consulta el bit correspondiente al puerto módulo 8. Como el resto de la división de 0x60 dividido 8 es cero, habrá que consultar el bit cero.

El bit 0 del valor leído 0x46 es cero, así que el puerto 0x60 se puede leer.

Task Register y descriptor de TSS

El formato del Task Register es el mismo que el de los registros de segmento. El selector apunta a un descriptor de TSS que se encuentra en la GDT. Si se intenta acceder a un descriptor de TSS en la LDT, el procesador genera una excepción 13.

Como en el caso de los descriptores de código y datos, el descriptor de TSS tiene los campos de base y límite y el bit de granularidad para indicar la dirección lineal donde comienza la TSS y cuál es su longitud.

El descriptor de TSS tiene un bit B que indica si el procesador está usando el TSS apuntado por el descriptor o no.

Lectura y escritura de TR

Existen dos instrucciones para acceder al Task Register.

Para escribirlo usamos LTR, que significa Load Task Register. Sólo se puede ejecutar con CPL = 0 y carga el selector de registro de tarea con el valor indicado en el argumento de la instrucción.

Para leer el selector de TR usamos STR, que significa Store Task Register. Dicho selector se almacena en el registro o posición de memoria indicado en el argumento de la instrucción.

Sólo se puede ejecutar LTR con un selector cuyo descriptor apunte a un TSS libre. Luego de la instrucción queda ocupado. Como el procesador escribe el bit B del descriptor, es necesario que la GDT esté en memoria RAM.

Compuerta de tarea

Aquí se puede ver el formato de la compuerta de tarea. Si la compuerta de tarea se encuentra en la GDT, se puede utilizar con las instrucciones JMP y CALL intersegmento y si se encuentra en la IDT, se puede utilizar con interrupciones y excepciones para generar un cambio de tarea.

Como todas las compuertas, el de tarea tiene una dirección lógica, es decir, selector y offset. En este caso, el selector apunta a un descriptor de TSS y el campo offset no se usa. Por eso los bytes 0, 1, 6 y 7 están reservados.

Al igual que con el resto de las compuertas, el campo DPL determina cuál es el código con menor privilegio que puede usarla. Lo que hace el procesador es comparar DPL contra CPL. Si CPL es mayor que DPL genera una excepción 13. Sin embargo esta comparación no se realiza si la compuerta se utiliza debido a una excepción o una interrupción por hardware.

Scheduler

El scheduler es una rutina del sistema operativo cuyo objetivo es poder lograr que las diferentes tareas corran en el orden esperado por el sistema operativo.

Para ello, el sistema operativo usa listas, donde se encuentra información sobre los diferentes procesos y threads. Lo más relevante para el scheduler es saber si la tarea está dormida o despierta y cuál es su prioridad, es decir si determinada tarea se debe ejecutar antes que otras tareas.

El scheduler se divide en dos partes: la determinación de la tarea a saltar y el salto a la tarea.

En los sistemas operativos cooperativos, las tareas llaman indirectamente al scheduler cuando deben dormir, por ejemplo porque están esperando caracteres de un puerto serie, entrada de teclado, etc.

En los sistemas operativos no cooperativos, donde el scheduler se llama periódicamente y detiene cualquier tarea que esté corriendo y comienza a correr la siguiente tarea no dormida, se pueden distinguir dos casos: scheduler por compuerta de tarea y por compuerta de interrupción.

Cambio de tarea

El cambio de tarea es una rutina que efectúa el scheduler con o sin ayuda del procesador para salvar el estado de la tarea anterior en el contexto de dicha tarea, recuperar el estado del contexto de la nueva tarea y saltar a la nueva tarea a ejecutar.

El contexto de la tarea es el área donde se almacenan los recursos del procesador tales como registros necesarios para que continúe ejecutando la tarea. En el caso de cambio de tarea automático dicha área se encuentra en el TSS. Si el cambio de tarea no es automático, el sistema operativo reservará memoria para el contexto de cada tarea.

En la multitarea cooperativa (cooperative multitasking en inglés), cada tarea decide cuándo pasar el control a la siguiente a través del scheduler.

En la multitarea no cooperativa (preemptive multitasking en inglés), el scheduler se ejecuta en forma periódica gracias a una interrupción del procesador.

Cambio de tarea automática

El diagrama muestra como el procesador ejecuta un cambio de tarea automática. Se usan dos TSS: uno para la tarea vieja y otra para la nueva que se está por ejecutar.

El registro TR (Task Register) apunta al descriptor de la tarea vieja, mediante el campo selector, y también a la TSS vieja, mediante los campos base y límite.

Lo primero que ocurre en el cambio de tarea automático, es cargar un registro interno TR_bak no visible para el programador, con el selector de la tarea nueva. Esto genera una lectura del descriptor de TSS de la tarea nueva.

Luego que el procesador hace las verificaciones de protección y de los valores del bit de ocupado de ambos TSS, el procesador salva los registros del procesador en el TSS de la tarea vieja, excepto el registro de control CR3.

En el siguiente paso, el procesador carga los registros desde la TSS, esta vez sí incluyendo CR3.

Luego el procesador copia el contenido del registro interno TR_bak en el registro TR, haciendo que éste apunte al TSS de la nueva tarea. Esto no genera una nueva lectura del descriptor de TSS de la nueva tarea porque también se copian los campos base y límite de TR_bak

a TR.

Instrucciones que cambian de tarea

Las instrucciones que cambian de tarea en forma automática son las siguientes:

JMP intersegmento cuyo selector apunta a un descriptor de TSS o compuerta de tarea.

CALL intersegmento cuyo selector apunta a un descriptor de TSS o compuerta de tarea.

Instrucción INT, interrupción de hardware o excepción cuya compuerta en la IDT es de tarea.

IRET cuando el indicador NT (Nested Task), que es el bit 14 de EFLAGS vale 1.

En todos los casos luego de ejecutar correctamente el cambio de tarea, el procesador enciende el bit 3 del registro de control CR0 denominado Task Switched.

JMP y CALL indirectos

Cuando la cantidad de tareas del sistema es variable, la instrucción de salto que se usa es indirecto. Esto significa que la instrucción JMP no incluye selector y offset sino un puntero a una estructura que contiene el selector y offset. El scheduler sobreescribe el campo selector de esa estructura antes de saltar.

En la estructura apuntada por la instrucción, el offset ocupa cuatro bytes y el selector dos bytes.

Para especificar que el salto o la llamada intersegmento es indirecto, se utiliza la cláusula FAR entre el nombre de la instrucción y el puntero. Así se escribe JMP FAR [puntero] o CALL FAR [puntero].

Instrucción JMP

Ahora veremos cómo funciona la instrucción JMP usada para el cambio de tareas.

El descriptor de TSS viejo está apuntado por el registro TR. El descriptor de TSS nuevo está apuntado por el selector de JMP o bien por el selector de la compuerta de tarea que está apuntado por el selector del JMP.

El DPL del descriptor de TSS y el de la compuerta de tarea, si se usa, deben ser mayores o iguales que CPL.

El bit de ocupado del descriptor de TSS viejo debe estar a uno, mientras que el del descriptor de TSS nuevo debe estar a cero.

Luego del cambio de tarea automático, se invierten los bits de ocupado: el del descriptor de TSS viejo se pone a cero y el del descriptor de TSS nuevo se pone a uno.

Este esquema no permite cambiar de una tarea a sí misma. Esto previene que se pierdan los registros que se salvan en las TSS.

El registro TR apunta al TSS nuevo al finalizar la instrucción.

Cambio de tarea 1 a 2 con instrucción JMP

En el diagrama se pueden ver tres tareas. Se puede observar que cuando se realiza cambio de tarea mediante la instrucción JMP, todas las tareas tienen el bit de ocupado a cero, excepto la tarea en curso.

Así, en el cambio de tarea 1 a la 2, la tarea 1 se libera mientras que la 2 se ocupa.

La tarea 3 tiene el bit de ocupado a cero porque no participa en el cambio de tarea.

Cambio de tarea 2 a 3 con instrucción JMP

En el cambio de tarea 2 a 3 con la instrucción JMP la tarea 2 se libera mientras que la 3 se ocupa.

La tarea 1 tiene el bit de ocupado a cero porque no participa en el cambio de tarea.

Cambio de tarea 3 a 1 con instrucción JMP

En el cambio de tarea 3 a 1 con la instrucción JMP la tarea 3 se libera mientras que la 1 se ocupa.

La tarea 2 tiene el bit de ocupado a cero porque no participa en el cambio de tarea.

Instrucción CALL o INT

Ahora veremos cómo funcionan las instrucciones CALL o INT, interrupciones de hardware o excepciones usadas para el cambio de tareas.

El descriptor de TSS viejo está apuntado por el registro TR. El descriptor de TSS nuevo está apuntado por el selector de CALL o bien por el selector de la compuerta de tarea que está apuntado por el selector del CALL o que se encuentra en la IDT en el caso de interrupción o excepción.

El DPL del descriptor de TSS y el de la compuerta de tarea, si se usa, deben ser mayores o iguales que CPL.

El bit de ocupado del descriptor de TSS viejo debe estar a uno, mientras que el del descriptor de TSS nuevo debe estar a cero.

Luego del cambio de tarea automático, el bit de ocupado del descriptor de TSS viejo continúa a uno y el del descriptor de TSS nuevo se pone a uno.

Este esquema no permite cambiar de una tarea a sí misma o a otra que haya llamado directa o indirectamente a la tarea vieja.

Esto previene que se pierdan los registros que se salvan en las TSS.

El campo backlink del nuevo TSS se carga con el selector de TSS de la vieja tarea. Así se forma una lista de las tareas que llaman a otras.

El registro TR apunta al TSS nuevo al finalizar la instrucción.

El procesador pone a uno el indicador Nested Task, que es el bit 14 de EFLAGS.

Cambio de tarea 1 a 2 con instrucción CALL

En el diagrama se pueden ver tres tareas. Se puede observar que cuando se realiza cambio de tarea mediante la instrucción CALL, todas las tareas aún no llamadas tienen el bit de ocupado a cero, y la tarea que está corriendo y las que la llamaron directa o indirectamente tienen el bit de ocupado a uno.

Así, en el cambio de tarea 1 a la 2, la tarea 2 se ocupa.

La tarea 3 tiene el bit de ocupado a cero porque no fue llamada aún.

Cambio de tarea 2 a 3 con instrucción CALL

En el cambio de tarea 2 a 3 con la instrucción CALL o INT la tarea 3 se ocupa.

La tarea 1 tiene el bit de ocupado a uno porque llamó a la tarea 2.

No se puede realizar un cambio de tarea de la 3 a la 1 porque está ocupada. Esto es correcto, ya que de no existir el bit de ocupado, se perderían los registros almacenados en la TSS de la tarea 1.

Instrucción IRET si NT es igual a 1

El propósito de la instrucción IRET cuando el indicador NT vale uno es volver a la tarea que se estaba ejecutando antes del último CALL o INT que cambió de tarea.

Es importante tener en cuenta que la instrucción RETF que sirve para volver de una subrutina que se encuentra en otro segmento, no se puede utilizar para cambiar de tarea. Así que la instrucción CALL se aparea con IRET.

El descriptor de TSS viejo está apuntado por el registro TR. El descriptor de TSS nuevo está apuntado por el campo callback del TSS viejo.

Los bits de ocupado de ambos descriptores deben estar a uno.

Luego del cambio de tarea el bit de ocupado del descriptor de TSS viejo se pone a cero y el del TSS nuevo continúa a uno.

El registro TR apunta al TSS nuevo al terminar la instrucción.

Cambio de tarea 3 a 2 con instrucción IRET

En el caso del cambio de tarea 3 a 2 con instrucción IRET el selector del descriptor del nuevo TSS se encuentra en el campo backlink del TSS de la tarea 3.

La tarea 3 que es la vieja se libera, mientras que la tarea 2 continúa ocupada al ejecutar IRET.

La tarea 1 figura como ocupada porque llamó a la tarea 2.

Cambio de tarea 2 a 1 con instrucción IRET

En el caso del cambio de tarea 2 a 1 con instrucción IRET el selector del descriptor del nuevo TSS se encuentra en el campo backlink del TSS de la tarea 2.

La tarea 2 que es la vieja se libera, mientras que la tarea 1 continúa ocupada al ejecutar IRET.

Tarea inicial

En sistemas multitarea siempre hay una tarea inicial, que es la que se ejecuta antes de configurar el registro TR.

Dicha tarea inicial debe tener su TSS asociado que originalmente debe tener su bit de ocupado a cero ya que la instrucción LTR requiere un TSS libre.

Todos los registros de segmentación y el registro LDTR deben inicializarse con selectores válidos o cero. De esta manera, cuando el procesador vuelva a ejecutar esta tarea, no carga selectores inválidos en registros de segmentación, lo que puede generar excepción 13.

La tarea inicial se puede reutilizar como tarea "idle" que ejecuta un ciclo infinito que contiene la instrucción HLT para ahorrar energía cuando otras tareas no están corriendo.

Scheduler por compuerta de tarea

En este diagrama se puede ver cómo funciona el scheduler por compuerta de tarea.

El scheduler está en una tarea aparte que se llama periódicamente mediante una compuerta de tarea ubicada en la IDT.

Cada vez que ocurre una interrupción del timer, la tarea que está corriendo se detiene y cambia a la tarea del scheduler. En ese momento el procesador escribe el campo backlink de la TSS del scheduler con el selector de TSS de la tarea que estaba ejecutando.

El scheduler determina el selector de la próxima tarea a ejecutar, sobreescribe el campo backlink con ese selector y ejecuta IRET, efectuando un cambio de tarea hacia la tarea que definió el scheduler como destino.

La cantidad de TSS a usar es la cantidad de tareas más una para el scheduler y otra para la tarea inicial.

Se debe inicializar el scheduler y la tarea inicial como desocupados y el resto como ocupados.

Después de ejecutar por primera vez la instrucción IRET, el procesador salva los registros en la TSS del scheduler. Entre otros salva el puntero de instrucción CS:EIP. Dicho puntero apunta a la instrucción siguiente al IRET. De esta manera, cuando se ejecute el scheduler por segunda vez, el procesador va a ejecutar la instrucción que viene después del IRET y por eso allí debe haber un salto al principio de la rutina del scheduler.

Como la tarea del scheduler es independiente del resto de las tareas, no es necesario ejecutar PUSHAD al principio del manejador de interrupción y POPAD al final.

Instrucciones para cambio de tarea usando compuerta de tarea

Aquí se puede ver la secuencia de instrucciones para implementar el cambio de tarea usando compuerta de tarea.

Lo primero que debe hacer el scheduler es hallar el selector de TSS correspondiente a la próxima tarea a ejecutar, o tarea idle si no hay ninguna a ejecutar, usando tablas o listas.

Suponiendo que el scheduler obtiene en el registro AX dicho selector, el scheduler carga un puntero al campo backlink del TSS del scheduler. Como ese campo está al principio del TSS, podemos ejecutar la instrucción MOV EBX,TSS_scheduler.

Luego sobreescribimos el campo backlink con el selector del TSS de la nueva tarea mediante la instrucción MOV [EBX],AX.

A continuación cambiamos de tarea mediante la instrucción IRET.

Finalmente volvemos a la primera instrucción del manejador de la interrupción mediante la instrucción JMP scheduler.

Scheduler por compuerta de interrupción

En este diagrama se puede ver cómo funciona el scheduler por compuerta de interrupción.

En este caso se usa compuerta de interrupción para el scheduler, lo que no provoca cambio de tarea. El cambio de tarea ocurre cuando se ejecuta una instrucción JMP dentro del scheduler.

Las flechas numeradas en el diagrama muestran el orden en el que ocurren las operaciones.

Suponiendo que se está ejecutando el código de la tarea 1, ocurre la interrupción del timer lo que provoca la ejecución del scheduler. Dicha rutina determina que la siguiente tarea a ejecutar sea la 2. Entonces ejecuta un salto intersegmento usando el selector de esta tarea provocando un cambio de tarea. El scheduler luego ejecuta IRET y comienza a ejecutar el código de la tarea 2.

Cuando ocurre otra vez la interrupción del timer, vuelve a ejecutar el scheduler y esta vez determina que la siguiente tarea a ejecutar es la 1. Ejecuta otro salto intersegmento donde el selector es el de la tarea número 1, provocando otro cambio de tarea. Finalmente la instrucción IRET hace que se ejecute el código de la tarea 1.

En este diagrama figura dos veces el scheduler, pero en realidad está una sola vez en memoria. Lo que ocurre es que en un caso corre dentro de la tarea 1 y en otro caso dentro de la tarea 2.

Como no se puede saltar a la misma tarea, el scheduler debe verificar esta condición antes de intentar ejecutar la instrucción JMP intersegmento. Si continúa ejecutando la tarea, el scheduler debe omitir la instrucción JMP y ejecutar la instrucción IRET para continuar ejecutando el código de la tarea.

La cantidad de TSS a utilizar en el sistema es la cantidad de tareas más una para la tarea inicial.

Todos los descriptores de TSS deben comenzar como desocupados.

Una vez determinado el selector de la TSS de la tarea a saltar, obtenemos el valor del registro TR con la instrucción STR cuyo argumento es un registro de 16 bits. Si el registro coincide con el selector de la nueva tarea, debemos ejecutar IRET inmediatamente. En caso contrario debemos hacer JMP intersegmento usando el nuevo selector y luego usar la instrucción IRET para volver al código de la tarea.

Instrucciones para cambio de tarea usando compuerta de interrupción

Aquí se puede ver la secuencia de instrucciones para implementar el cambio de tarea usando compuerta de interrupción.

Lo primero que debe hacer el scheduler, como manejador de la interrupción del timer, es salvar los registros de uso general mediante la instrucción PUSHAD.

Luego determina el selector de la tarea a saltar. Supongamos que dicho selector queda cargado en el registro AX.

Debemos obtener el selector de la tarea actual leyendo el registro TR mediante la instrucción STR BX.

Luego comparamos ambos valores y saltamos si son iguales, para no realizar el salto intersegmento. Esto se logra mediante las instrucciones CMP BX,AX y luego JE me_voy.

El cambio de tarea mediante el salto intersegmento se materializa mediante dos instrucciones: primero salvamos el selector de tarea en la estructura que usa el salto indirecto mediante la instrucción MOV [sel_TSS],AX y luego efectuamos el salto indirecto mediante la instrucción JMP FAR [puntero_TSS].

Finalmente terminamos el manejador de la interrupción del timer mediante las instrucciones POPAD e IRET.

En la zona de datos definimos la estructura utilizada por el salto indirecto. Primero se ubica el offset de cuatro bytes y luego el selector de dos bytes.

Modo virtual 8086

El modo virtual 8086 es una tarea que corre con CPL=3 donde los registros de segmento operan igual que en modo real. Esto significa que cuando se carga un selector en un registro de segmento, la base se carga con el valor del segmento por 0x10.

Hay dos maneras de entrar al modo virtual 8086.

En el primer caso se puede entrar mediante un cambio de tarea donde el registro EFLAGS tenga el bit 17 encendido. Ese bit se llama VM, de Virtual Mode.

En el segundo caso se puede entrar mediante una instrucción IRET desde CPL = 0 donde la imagen de EFLAGS en la pila tenga el bit 17 encendido.

Si CPL es mayor que cero, entonces el bit VM no muestra su valor real. Siempre indica cero.

En el caso de interrupciones o excepciones, el procesador sale del modo virtual y reingresa mediante IRET como se comentó previamente.

Las instrucciones para habilitar y deshabilitar interrupciones STI y CLI, generan excepción 13. El manejador debe determinar si habilitar o no las interrupciones en el modo virtual.

A partir del procesador Pentium, existen varias optimizaciones del modo virtual 8086 que permiten que corra mucho más rápido, ya que incluye entre otras cosas un nuevo indicador de interrupciones virtual en el registro EFLAGS y no es necesario correr software en el manejador de la excepción 13.

Protección en compuertas

Ahora veremos algunos ítems de protección que realiza el procesador cuando ejecuta diferentes niveles de privilegio. Esto normalmente ocurre cuando se utilizan sistemas multitarea, que requieren diferentes niveles de privilegio para proteger el sistema.

El procesador efectúa dos verificaciones con respecto a las compuertas.

Si no ocurre una excepción o interrupción por hardware, el procesador compara el DPL de la compuerta contra CPL. Si CPL es mayor que el DPL de la compuerta, ocurre una excepción 13.

Por ejemplo, las compuertas de excepción o de interrupción de hardware deben tener DPL=0, para evitar que una aplicación intente llamar a la interrupción mediante la instrucción INT.

Las compuertas de llamada y de interrupción utilizadas para llamadas al sistema deben tener DPL=3 para que la aplicación pueda usar la compuerta.

Si el campo DPL del descriptor al que apunta el selector de la compuerta es mayor que CPL, ocurre una excepción 13.

Esto significa que un código de privilegio cero no puede llamar a otro de privilegio 3, pero sí se puede dar el caso contrario.

Llamadas al sistema

Las llamadas al sistema (system call en inglés) permiten que una tarea corriendo con CPL=3 llame a una rutina del kernel para realizar un servicio tal como pedir acceso a memoria, interactuar con hardware, poner a dormir la tarea, etc.

Dichas llamadas se pueden implementar mediante la instrucción CALL intersegmento a través de una compuerta de llamada en la GDT o LDT o mediante una instrucción INT usando una compuerta de excepción en la IDT para no deshabilitar las interrupciones.

La compuerta debe tener DPL=3 para permitir el acceso a la aplicación. En caso contrario se genera excepción 13.

Se debe tener en cuenta que las instrucciones JMP y CALL cuyo selector apunte a un descriptor de código, no cambian el nivel de privilegio. Es decir que la compuerta de llamada es obligatoria en el caso de implementar llamadas al sistema mediante la instrucción CALL.

Acceso de datos

Para acceder a datos en memoria, se debe tener en cuenta el campo RPL o Requested Privilege Level del selector, que son los bits 1 y 0, el campo DPL (Descriptor Privilege Level) del descriptor, que son los bits 6 y 5 del byte 5 y el CPL o Current Privilege Level, que es el nivel de privilegio del segmento de código, que se obtiene leyendo los dos bits menos significativos del selector de CS.

El Effective Privilege Level es el máximo numérico entre RPL y CPL.

Si EPL es mayor que DPL, ocurre una excepción 13. En caso contrario se efectúa el acceso a memoria.

Para el caso del segmento de pila, los tres valores deben ser iguales: tanto el CPL como el DPL del descriptor de datos y el RPL del selector de SS. En caso contrario ocurre una excepción 13.

Manejo de pila con compuerta de llamada

En estos diagramas se muestra lo que el procesador pone en la pila cuando hay un CALL mediante compuerta de llamada.

Cuando no hay cambio de nivel de privilegio, el procesador ingresa en la pila el CS del llamador y el EIP del llamador.

Si hay cambio de nivel de privilegio, como el privilegio de la pila es la misma que la del código, el procesador obtiene el puntero de la nueva pila del TSS.

En dicha pila pone el SS y ESP de la pila vieja, luego se copian los parámetros. La cantidad de DWORDs a copiar está indicada en el byte 4 de la compuerta de llamada. Finalmente el procesador coloca en la pila el CS y EIP de la rutina llamadora.

Si debido al byte 4 de la compuerta de llamada el procesador copió parámetros de la pila vieja a la nueva, el retorno de subrutina se debe realizar mediante la instrucción RETF n donde n es la cantidad de bytes que ocupan los parámetros.

Por ejemplo, si el byte 4 de la compuerta de llamada vale 3, significa que el procesador copia 12 bytes entre pilas. Por lo tanto, el retorno de la subrutina debe utilizar la instrucción RETF 12.

Manejo de pila con compuerta de interrupción

Cuando no hay cambio de nivel de privilegio, el procesador pone en la pila el registro EFLAGS y luego el CS y EIP de la rutina interrumpida. En el caso de algunas excepciones, el procesador pone en la pila el código de error.

Si hay cambio de nivel de privilegio, al igual que en el caso de la compuerta de llamada, el procesador busca el puntero de la nueva pila en el TSS, teniendo en cuenta que el privilegio de la pila debe ser igual que el del código.

Luego el procesador salva en la nueva pila los registros SS y ESP que apuntan a la vieja pila, el registro EFLAGS y luego el CS y EIP de la rutina interrumpida. En el caso de algunas excepciones, el procesador pone en la pila el código de error.

El manejador de la rutina de excepción debe retirar el código de error de la pila para poder ejecutar la instrucción IRET.

Cambio manual de tareas

El cambio automático de tareas visto hasta ahora funciona solamente en procesadores corriendo en modo protegido. Esto significa que es incompatible con procesadores de otros fabricantes y también con el modo largo, que es un modo de operación de los procesadores Intel que permite correr programas en 64 bits.

Los sistemas operativos escritos en forma portable para que funcionen con diferentes procesadores, usan el cambio manual de tareas. En vez de usar un TSS por tarea, se usa un solo TSS, que es necesario para poder realizar cambios de nivel de privilegio y opcionalmente si se necesitan proteger puertos de entrada/salida.

El procesador no va a salvar automáticamente los registros en la TSS sino que el schedular es el encargado de salvar los registros en una zona de memoria de privilegio cero llamado contexto de la tarea. Por eso no se usan las instrucciones JMP y CALL intersegmento cuyos selectores apunten a descriptor de TSS o compuerta de tarea, ni tampoco excepciones o interrupciones de hardware o software cuya entrada en la IDT sea una compuerta de tarea. Debido a esto, el valor del indicador Nested Task o NT del registro EFLAGS siempre va a valer cero.

El procesador debe salvar en el área de contexto los registros que figuran en la TSS junto con el par SS0:ESP0 que sirve para poder cambiar de pila cuando hay un cambio de nivel de privilegio, debido a que el nivel de privilegio de la pila siempre coincide con el nivel de privilegio del código (o CPL).

Existen muchas maneras de salvar los registros en el área de contexto. Aquí vamos a suponer que dicha área se encuentra en la pila de privilegio cero.

El scheduler se llama mediante compuerta de interrupción, así que cuando comienza a ejecutar el manejador, la pila de nivel de privilegio cero ya tiene el puntero de pila de privilegio 3 SS:ESP, el registro EFLAGS y el puntero de instrucciones formados por el par de registros CS:EIP.

Preservamos los registros de uso general mediante la instrucción PUSHAD y luego salvamos en la pila los registros de segmentación, excepto CS que ya fue salvado durante la vectorización de la interrupción. A continuación leemos los valores de los campos SS0 y ESP0 de la única TSS e ingresamos estos valores en la pila.

Ahora debemos salvar el puntero de pila SS:ESP en un array de punteros de pila, en la entrada correspondiente a la vieja tarea.

A continuación, cargamos el registro de control CR3 para que apunte al nuevo directorio de páginas y cargamos el puntero de pila SS:ESP con el contenido del array de punteros de pila, en la entrada correspondiente a la nueva tarea.

Luego debemos recuperar los registros en el sentido inverso en que ingresaron en la pila.

Primero obtenemos los dos primeros valores y grabamos los campos SS0 y ESP0 de la TSS, luego seguimos con los cinco registros de segmentación y finalmente con los registros de uso general usando la instrucción POPAD. Antes de ejecutar esta instrucción, debemos ejecutar la secuencia de End of Interrupt requerida por el PIC.

Finalmente ejecutamos la instrucción IRET para ejecutar el código de la tarea.

Como estamos usando las pilas de privilegio cero para almacenar los valores de los registros para cada tarea, antes de que puedan arrancar las tareas, debemos cargar los valores iniciales de los registros en las pilas de privilegio cero de cada tarea, incluyendo el puntero de pila de nivel de privilegio 3, SS3:ESP3, el registro EFLAGS, y el puntero de instrucciones CS:EIP en el orden esperado por la instrucción IRET.

Debe haber un único TSS que se ocupa mediante la instrucción LTR (Load Task Register). A partir de ese momento, el TSS está siempre ocupado.

Escenario de Caballo de Troya

Como vimos en la sección de acceso a datos, una rutina de nivel de privilegio 3 no puede leer o escribir un buffer de nivel de privilegio cero.

Para quebrar el sistema operativo, lo que podría intentar la aplicación es instalar un caballo de Troya, haciendo que sea el kernel el que sobreescriba el buffer, ya que el kernel, como tiene nivel de privilegio cero, puede leer y escribir en buffers de privilegio cero.

Sea la función read(int fd, void *buf, int len); que escribe en el buffer buf la cantidad de bytes especificados por len lo que se lee del archivo indicado por el file descriptor fd.

Si una aplicación que tiene CPL=3 llama al kernel para ejecutar esa función y le pasa un puntero a buffer del propio sistema operativo, la función sobreescribirá dicho buffer con el contenido del archivo, posiblemente "inoculando" algún virus informático.

La rutina que atiende la llamada al sistema tiene que ajustar el valor de RPL de los selectores que pasa la rutina llamadora para que tengan el valor del CPL de esa rutina llamadora.

De esta manera, si la función read se llama desde CPL=0, la función podrá escribir en buffers del sistema operativo, pero si se llama desde una aplicación desde CPL=3, ocurrirá una excepción 13 si el código intenta escribir un buffer del kernel.

Instrucción ARPL

La instrucción ARPL usa dos registros de 16 bits como operandos. El de la izquierda es el destino y el de la derecha es el fuente. Lo que hace la instrucción es copiar los dos bits menos significativos del registro fuente en el registro destino.

Esto es así porque RPL son los dos bits menos significativos del selector.

Espero que esto les haya sido de interés. Hasta luego.

Paginación de 32 bits en procesadores Intel

por Dario Alejandro Alpern

Transcripción

Hola. Mi nombre es Dario Alpern y hoy vamos a ver paginación en procesadores de 32 bits.

Memoria virtual 1

La memoria virtual es una característica del sistema operativo que permite que cada proceso tenga su propio espacio de direccionamiento independiente, lo que permite aislar las aplicaciones entre sí, mediante el método de la protección por paginación.

En este esquema, se subdivide el espacio direccionable del procesador en páginas iguales. En procesadores Intel, la longitud de las páginas es de 4 KB. Como se puede ver, hay una zona de memoria no paginada. Esta zona pertenece al sistema operativo y contiene código y datos que deben estar presentes todo el tiempo en la memoria del sistema. Por ejemplo, los manejadores de interrupción deben estar siempre en RAM.

A la derecha se observa el almacenamiento externo, que puede ser un disco rígido o actualmente SSD, que es la sigla en inglés de disco de estado sólido, implementado con memoria flash. También se subdivide en páginas de 4 KB.

Memoria virtual 2

Aquí se puede ver como el sistema operativo carga las páginas de diferentes procesos en memoria. En este caso cargamos tres páginas, pero en la realidad, como el tamaño de los programas se mide en megabytes, se cargan miles de páginas en memoria por cada proceso.

En la mayoría de los sistemas operativos, los procesos arrancan siempre en la misma dirección lineal (o virtual, como se la conoce en otros procesadores).

Vamos a suponer como se muestra en el diagrama que la página 1 comienza en la dirección lineal 0x00400000, la página 2 comienza en la dirección lineal 0x00401000 y la página 3 comienza en la dirección lineal 0x00402000.

Se puede ver que a la misma dirección lineal, le puede corresponder varias direcciones físicas. Recordemos que las direcciones físicas son aquellas que salen por el bus de direcciones hacia la memoria externa al procesador.

Memoria virtual 3

En el diagrama se ve lo que ocurre cuando el sistema operativo debe cargar en memoria el tercer programa, correspondiente a una planilla de cálculo.

Se puede observar que no hay suficiente memoria RAM. Lo que ocurre en ese caso es que se usa el almacenamiento externo para enviar las páginas menos recientemente utilizadas y de esa manera hacer espacio para cargar el nuevo programa.

Cuando el procesador necesite ejecutar o bien leer o escribir datos de una página descargada en el almacenamiento externo, no hay una traducción entre dirección lineal y dirección fisica de RAM. Entonces el procesador va a generar una excepción de fallo de página, que es la excepción 14, y el sistema operativo deberá recargar esa página en memoria física, mandando otra página a almacenamiento externo si está llena la memoria.

Si bien el almacenamiento externo permite correr más procesos, es muy lento enviar las páginas al medio externo y luego de nuevo a la RAM del sistema. Por eso normalmente se limita el área de almacenamiento externo al doble del tamaño de la RAM del sistema.

Memoria virtual 4

Como el sistema operativo carga las páginas en el primer espacio libre que encuentra, se puede ver que es posible que una instrucción se encuentre en zonas disjuntas de RAM, como en el caso de la segunda instrucción, donde el último byte se encuentra en una dirección física completamente diferente que los dos primeros bytes.

Está claro que la unidad de paginación debe ser capaz de traducir rápidamente entre las direcciones lineales, que fueron las generadas por el linker y las direcciones físicas, que es donde el procesador va a encontrar el código y los datos en la memoria.

Traducción 1

El diagrama que se ve acá corresponde a la paginación como se implementó originalmente en el procesador 80386.

La dirección lineal se divide en tres campos: los diez bits más significativos corresponden al índice del directorio de páginas, los diez bits siguientes corresponden al índice de la tabla de páginas y por último los 12 bits menos significativos representan el offset dentro de la página.

Tanto el directorio de páginas como las tablas de páginas son tablas que inicializa el sistema operativo y que lee el procesador para poder realizar la traducción.

El procesador usa el registro CR3 como puntero a la dirección física del directorio de páginas. Esta tabla tiene 1024 entradas de 4 bytes cada uno y debe comenzar en una dirección física múltiplo de 4 KB, es decir que los 12 bits menos significativos de la dirección física que corresponda al primer byte del directorio debe ser 000.

Cada entrada del directorio se subdivide en dos campos: la base son los 20 bits más significativos y los atributos los 12 bits menos significativos. La base indica la dirección física donde comienza la tabla de páginas. Como la tabla de páginas también comienza en una dirección física múltiplo de 4 KB, entonces los 12 bits menos significativos de esa dirección no tienen información porque siempre valen 000 y sólo se usan los 20 bits más significativos de la dirección física, que es lo que se carga en el campo base. El campo atributos indica cómo se comportan las páginas.

Como se ve en el diagrama, a la dirección física de inicio del directorio apuntado por CR3, se suma el índice que se encuentra en los bits 31 a 22 de la dirección lineal, multiplicado por 4, que es la longitud de cada entrada en el directorio.

De allí se obtiene el puntero al inicio de la tabla de páginas y se repite el proceso sumando el índice que se encuentra en los bits 21 a 12 de la dirección lineal, multiplicado por 4.

Ahí se obtiene la dirección física inicial de la página. El offset dentro de la página no se traduce, entonces los bits 11 a 0 de la dirección lineal siempre coinciden con los bits 11 a 0 de la dirección física.

Atributos 1

A continuación veremos cuáles son los atributos que se encuentran en las tablas de paginación.

El bit 8 es el bit global. Este bit solo es válido para la tabla de páginas y si el bit 7 del registro de control CR4 denominado PGE (Page Global Enable) vale 1. Cuando se modifica el valor del registro CR3 apuntamos a un directorio de páginas diferente, lo que significa que las traducciones de direcciones lineales a direcciones físicas cambia completamente. Debido a ello, el procesador borra la TLB, que es una tabla interna que almacena las últimas traducciones hechas. Sin embargo, no se borran las traducciones que corresponden a páginas globales.

En bit 7 es el de tamaño de página (PS o Page Size en inglés). Este bit solo es válido para el directorio de páginas y si el bit 4 del registro de control CR4 denominado PSE (Page Size Enable) vale 1. Cuando PS vale cero, el campo base de la entrada de directorio apunta a una tabla de páginas, y si vale uno, el campo base apunta a una página grande de 4 MB. En este caso, los bits 21 a 0 de la dirección lineal corresponden al offset dentro de la página grande.

El bit 6 indica si algún byte de la página se escribió. Esto sirve para optimizar la copia a almacenamiento externo, que no hace falta si la página no fue modificada.

El bit 5 indica si la página fue accedida.

Atributos 2

Cuando el bit 4 vale cero, indica que el caché está habilitado para la página. Cuando vale uno, no se usa el caché para la página.

Si el caché está habilitado, el bit 3 indica la política de escritura que se utiliza. Si vale cero, se utiliza write-back, mientras que si vale uno, se usa write-through.

En el caso de la RAM, hay que poner ambos bits a cero para que se use el caché, que es mucho más rápido que la RAM. Si tuviéramos un hardware mapeado a memoria, como un termómetro por ejemplo, se debe deshabilitar el caché en esa página para forzar que la lectura se haga siempre desde el termómetro.

Si el bit 2 vale 1 indica que la página corresponde al usuario (CPL = 3) y si es cero, al supervisor (CPL < 3).

El bit 1 es el permiso de escritura. Si el vale 1 se puede escribir en memoria.

Si el bit 0 vale uno, la página existe en memoria.

Cuando el atributo del directorio no coincide con el de la tabla de páginas, se toma el más restrictivo de los dos:

Si uno es presente y otro es no presente, el procesador elige no presente.

Si uno tiene habilitación de escritura y otro sólo lectura, se elige sólo lectura.

En el caso del atributo usuario/supervisor, como el usuario no puede acceder a una página con atributo supervisor, entonces supervisor es más restrictivo que usuario, por lo tanto, si uno tiene atributo usuario y el otro supervisor, se elige supervisor.

Escritura

Aquí se puede observar si el procesador puede leer o escribir dependiendo de los atributos usuario/supervisor y habilitación de escritura y el código que está corriendo: si es usuario o supervisor.

Se puede observar que el código de usuario no puede leer ni escribir páginas de supervisor. Tampoco puede escribir en páginas de usuario de solo lectura.

Cuando corre el supervisor, el comportamiento depende del indicador Write Protect, que es bit 16 del registro de control CR0. Cuando Write Protect vale cero, el supervisor puede leer y escribir en cualquier página presente, sin importar el valor de los atributos usuario/supervisor ni el de habilitación de escritura.

Si Write Protect vale 1, El procesador solo tiene en cuenta el atributo de habilitación de escritura. Si el atributo está encendido, el supervisor puede leer o escribir en la página. En caso contrario sólo puede leer.

Ejemplo parte 1

Veremos un ejemplo para afianzar los conceptos.

Suponiendo que el directorio de páginas se encuentra en la dirección física 0x00100000 y las tablas de páginas a continuación, indicar el valor de CR3 y las direcciones con el contenido a escribir en las tablas para que a la dirección lineal 0xAA234889 le corresponda la dirección física 0x44522889. Todas las páginas son de supervisor y de lectura/escritura.

Como sabemos que los 12 bits menos significativos de las direcciones lineales no se traducen, primero comprobamos que esto ocurra. Ambas direcciones terminan con los tres nibbles menos significativos en 889, así que está bien.

En base a lo indicado en el enunciado, como el directorio comienza en la dirección física 0x00100000, este es el valor a cargar en el registro CR3.

Ejemplo parte 2

A la izquierda se ve el mapa de memoria como indica el enunciado: el directorio a partir de la dirección física 0x00100000 y las tablas de paginación a continuación.

A la derecha se puede ver como queda el directorio para que las bases apunten a las diferentes tablas de páginas. Para los atributos, seleccionamos supervisor, entonces el bit 2 debe valer cero, permiso de escritura, entonces el bit 1 debe valer uno y finalmente página presente, por lo que el bit 0 también vale uno. De esta manera el atributo queda con el valor 3.

Ejemplo parte 3

Lo primero que debemos hacer es convertir la dirección lineal de hexadecimal a binario. Esto es sencillo porque los números hexadecimales siempre se corresponden con cuatro bits. En la conversión separé los bits en grupos de cuatro para facilitar la traducción.

Luego separamos la parte alta en rojo, que corresponde al índice del directorio y la parte baja en azul, que corresponde al índice de la tabla de páginas.

Una vez obtenidos ambos índices en hexadecimal, podemos ver qué dirección del directorio se debe modificar y qué dato hay que escribir.

Como cada elemento del directorio tiene 4 bytes, multiplicamos el índice por 4 para obtener el offset dentro del directorio. A este valor le sumamos la dirección física de inicio del directorio. Así obtenemos la dirección 0x00100AA0.

El contenido a escribir tiene dos partes: la base que es lo que deberemos hallar, y el atributo, que vale 3 como ya habíamos comentado.

La base es igual a la dirección física de la tabla de pagínas número 0x2A8. Las tablas de páginas están almacenadas una a continuación de la otra a partir de la dirección 0x00101000. Como el tamaño de cada tabla es 0x1000, sumaremos a la dirección física de la tabla de páginas cero, que es 0x00101000, el número de la tabla de página 0x2A8 por la longitud de cada tabla de páginas, que es 0x1000. Al resultado le sumaremos 3, que es el atributo. El resultado es 0x003A9003.

Con respecto a la dirección de la tabla a escribir y su contenido, a la dirección física del inicio de la tabla de página que es 0x003A9000, le sumamos el índice de la tabla de páginas que es 0x234 por 4, que es la longitud de cada entrada de la tabla de páginas. El resultado es 0x003A98D0.

El contenido a escribir es la base de la página, que es 0x44522000 más el atributo que vale 3. El resultado es 0x44522003.

Traducción PAE

El Pentium Pro, lanzado en el mercado a fines de 1995, fue el primer procesador de Intel de la serie x86 con más de 32 bits de bus de direcciones. Este procesador podía direccionar 64 GB con sus 36 bits de bus de direcciones.

Como las direcciones lineales seguían siendo de 32 bits, tuvieron que cambiar el esquema para acceder a direcciones físicas de más de 32 bits. La siguiente potencia de 2 es 64 bits. De esta manera, todas las entradas de las tablas de paginación tienen una longitud de 64 bits, es decir de 8 bytes.

Como las tablas siempre tienen 4 KB de longitud, la cantidad de entradas se reduce a la mitad, es decir, 512 entradas.

De esta manera, ahora los índices tienen 9 bits en vez de 10. En el diagrama se puede observar que tanto el índice del directorio de páginas como el índice de la tabla de páginas tienen 9 bits cada uno. Como sobran 2 bits, se agregó una tabla adicional de 4 elementos llamado puntero a directorio de páginas.

El registro CR3 apunta ahora al puntero a directorio de páginas. Como el registro CR3 tiene 32 bits, la dirección física del puntero de directorio de páginas debe estar por debajo de los 4 GB. Además debe estar alineado a múltiplo de 32 bytes.

Dentro de las tablas, los atributos ocupan los bits 63 a 56 y 11 a 0. El resto corresponde a la base.

Este sistema es incompatible con el anterior y se habilita poniendo a uno el bit 5 del registro de control CR4 denominado PAE (Physical Address Extension).

Diferencias PAE

Aquí se pueden ver las diferencias más significativas en la paginación según si PAE está activo o no.

Cuando PAE, que es el bit 5 de CR4 vale uno, hay tres niveles de paginación. Si vale cero, hay dos.

Cuando PAE está habilitado, las entradas de las tablas de paginación tienen 64 bits, en caso contrario, 32 bits.

Cuando PAE está habilitado, las páginas grandes, aquellas con el atributo Page Size a 1 en el directorio de páginas, ocupan 2 MB, en caso contrario ocupan 4 MB.

Cuando PAE está habilitado, también están habilitadas las páginas grandes, pero PSE (Page Size Enable) que es el bit 4 de CR4, debe estar a cero.

Con PAE activado, se agrega el atributo de no ejecución, que se ubica en el bit 63. Si la página está marcada como de no ejecución, ocurrirá una excepción de fallo de página si se intenta ejecutar una instrucción obtenida de dicha página.

Fallo de página

Aquí se muestra el código de error que ingresa en la pila cuando el procesador llama al manejador de la excepción de fallo de página. Además, el procesador pone en el registro CR2 la dirección lineal que causó el fallo de página.

El bit 0 indica si el fallo se dio por página presente o no.

El bit 1 indica si el fallo ocurrió durante una escritura, si está a uno, o durante una lectura, si el bit está a cero.

El bit 2 está encendido si la falla ocurrió con CPL = 3, correspondiente al usuario. Vale cero si CPL < 3, que corresponde al supervisor.

El bit 3 se enciende si las entradas de directorio o de tabla de páginas tienen un "1" en un campo reservado. Esto solo puede ocurrir si PSE o PAE valen uno.

Cuando está encendido el bit 4, indica que la falla ocurrió por intentar ejecutar una instrucción. Esto solo puede ocurrir si el atributo de no ejecución está activado.

TLB

Cuando PAE está desactivado, para traducir de direcciones lineales a direcciones físicas, el procesador debe leer dos direcciones de memoria, una correspondiente al directorio y otra de la tabla de páginas. Esto ralentizaría enormemente la ejecución de los programas ya que haría falta tres accesos a memoria por cada acceso que requiera el programa.

Para solucionar este inconveniente, dentro de la unidad de paginación, el procesador tiene una memoria caché completamente asociativa que almacena las últimas traducciones.

La memoria asociativa compara el dato de entrada, en este caso la dirección lineal proveniente de la unidad de segmentación con todas las direcciones lineales que se encuentren activas en el TLB en el mismo ciclo de reloj. Si hay coincidencia, el TLB devuelve la dirección física y los atributos de la página que corresponden con esa dirección lineal.

Si no se encuentra la dirección lineal en el TLB, el procesador deberá leer las tablas de paginación que se encuentran en memoria externa para poder realizar la traducción a memoria física.

En el diagrama se puede ver el TLB cuyo tamaño depende del procesador. Por ejemplo, en el 80386, la TLB original tenía 32 entradas.

En procesadores más modernos, hay una TLB por tamaño de página. Así, hay un TLB para páginas de 4 KB y otra para páginas de 2 MB y 4 MB.

Tanto las direcciones lineales como las físicas no incluyen los bits de offsets de página. Es decir que tienen 20 bits si PAE = 0.

Tanto la dirección física como los atributos de pagína se cargan en la TLB cuando se realiza la primera traducción de esa dirección lineal.

Los atributos de caché contienen información que indican si la entrada es válida, o cuándo liberar la traducción si la TLB se llena, usando la política LRU (Least Recently Used en inglés).

Borrado TLB

Existen dos maneras de borrar la TLB: escribiendo el registro CR3 o utilizando la instrucción INVLPG.

Cuando se modifica el valor del registro CR3 apuntamos a un directorio de páginas diferente, lo que significa que las traducciones de direcciones lineales a direcciones físicas cambia completamente. Debido a ello, el procesador borra la TLB. Sin embargo, no se borran las traducciones que corresponden a páginas globales, que se podrían usar para apuntar a la zona no paginada.

La instrucción INVLPG sirve para borrar la traducción que se refiera a la posición de memoria indicada por dicha instrucción. En este caso no importa el valor del atributo global.

Habilitación de paginación

Aquí se muestran los pasos necesarios para activar la unidad de paginación.

Primero se deben preparar las tablas de paginación asegurando que el código que contenga la habilitación de la unidad de paginación esté en identity mapping, lo que significa que las direcciones lineales y las físicas coinciden.

Después debemos asegurarnos que el flag de interrupción esté apagado mediante la instrucción CLI.

A continuación habilitaremos o no PAE en base al tipo de paginación que necesitemos. Por ejemplo, si necesitamos acceder más allá de los 4 GB, entonces PAE debe estar activado.

Luego cargamos en el registro CR3 el puntero a la dirección física del directorio de páginas si PAE no está habilitado o al puntero de directorio de páginas si lo está.

Si PAE no está habilitado pero necesitamos pagínas grandes de 4 MB, deberemos activar PSE.

Finalmente habilitaremos la paginación activando el bit 31 de CR0.

Espero que les haya sido de interés. Hasta luego.

Bloques del 80386

por Dario Alejandro Alpern

Transcripción

Hola. Mi nombre es Dario Alpern y hoy vamos a ver los bloques que componen el procesador 80386.

Diagrama 80386

Aquí podemos ver el diagrama en bloques del microprocesador 80386. Vamos a analizar las distintas unidades desde el bloque inferior derecho en sentido horario, hacia la izquierda y luego en la parte superior del diagrama, hacia la derecha. Las tres unidades que se encuentran en la parte superior, las de segmentación, paginación y de control del bus se activan cuando hay acceso a memoria.

El procesador interactúa con el bus para leer y escribir datos y para leer instrucciones que debe ejecutar el procesador. En inglés esto último se llama fetch. El fetch se realiza con menor prioridad que la lectura o escritura de datos.

Predecodificación

Ahora vamos a ver como funciona la predecodificación de instrucciones.

La unidad de prebúsqueda tiene una cola de 16 bytes donde almacena los bytes que obtiene durante la lectura de instrucciones en los momentos que no hay lectura o escritura de datos. Si bien la cola es de bytes, el microprocesador puede leer hasta cuatro bytes por vez ya que su bus de datos es de 32 bits.

A continuación se encuentra la unidad de predecodificación que incluye una cola de 3 instrucciones predecodificadas. Esto significa que el procesador puede separar las instrucciones del flujo de bytes antes de ejecutarlas.

Para ello, en el código de máquina, que es lo que se encuentra en la cola de bytes de código, algunos bytes se interpretan como prefijos, que se ven en celeste en el diagrama, como 66 hexa, que es el prefijo de tamaño de operandos, que indica si el o los operandos de la instrucción tienen 16 o 32 bits. Otro prefijo es 67 hexa, que indica si el direccionamiento a memoria que posee la instrucción es de 16 o de 32 bits. También hay prefijos de registro de segmento, que indican cuál de los 6 registros de segmento usa la instrucción que accede a memoria. Por ejemplo, el prefijo 2E hexa se refiere a CS, el prefijo 3E hexa se refiere a DS, etc. Por último hay un prefijo LOCK, que es el F0 hexa, que cuando se usa con una instrucción que modifica memoria, activa la pata LOCK negado del bus de control del procesador para forzar una modificación atómica, sin que lo interrumpa otro hardware.

Cuando los bytes no se interpretan como prefijos, lo hacen como códigos de operación, que están en rojo en este diagrama. Los códigos de operación tienen uno o dos bytes (en procesadores actuales pueden tener tres bytes). Estos códigos de operación tienen campos que dicen qué registros se usan y en el caso de acceso a memoria, qué registros se usan para el direccionamiento indirecto. También se puede saber cuántos bytes se usan para el caso de direccionamiento inmediato, que es cuando el dato se encuentra en la propia instrucción, como ADD EAX,5. De esta manera es relativamente fácil en base al código de operación y los prefijos 66 hexa y 67 hexa saber cuántos bytes ocupa la instrucción completa.

Esto permite ir llenando la cola de instrucciones, lo que vacía parcialmente la cola de bytes de código y esto hace que el procesador vaya a buscar más bytes de instrucciones de memoria.

Unidad de control

La unidad de control toma la instrucción más antigua que esté en la cola de instrucciones y en base a su código de operación determina si es una instrucción sencilla o compleja. Si la instrucción es sencilla, como las artiméticas y lógicas, la unidad de control genera los comandos para que la unidad aritmética y lógica ejecute la instrucción. En cambio si la instrucción es compleja, la unidad de control genera microinstrucciones que obtiene de la ROM de control para su posterior ejecución en la unidad aritmética y lógica y debido a ello, la instrucción se ejecuta en varios ciclos de reloj.

Como entrada de la unidad de control, también se encuentran los flags, ya que los saltos condicionales dependen de los valores de algunos de sus bits, como el indicador de cero, de acarreo, de signo, de sobrepasamiento o de paridad, para saber si tiene que saltar o no.

Unidad aritmética y lógica

La unidad aritmética y lógica contiene los 8 registros de uso general de 32 bits, que se pueden subdividir en registros de 16 y de 8 bits. También incluye el registro EFLAGS que tiene diversos indicadores, como los cinco recién mencionados, la habilitación de interrupciones, el indicador de dirección para instrucciones de cadena. La explicación completa de este registro se encuentra en el video de la parte 1 de introducción a Assembler.

Aparte de los registros recién mencionados, la unidad aritmética y lógica tiene hardware para realizar sumas, restas, multiplicaciones y divisiones, y un barrel shifter para hacer rotaciones y desplazamientos de registros o celdas de memoria.

Barrel shifter

El barrel shifter es un módulo que permite desplazar hacia la izquierda o la derecha de 0 a 31 bits un número de 64 bits en un único ciclo de clock, sin utilizar flip flops. De esta manera, cualquier desplazamiento o rotación de un registro requiere 3 ciclos de reloj independientemente de la cantidad de bits a mover. El barrel shifter de 64 bits se puede implementar mediante 384 multiplexores.

Se puede observar como inicializar el barrel shifter y de dónde tomar el resultado para diferentes instrucciones: shift left (SHL), shift right (SHR), rotate left (ROL),rotate right (ROR), shift left double (SHLD) y shift right double (SHRD). También se puede utilizar el barrel shifter para rotate through carry left (RCL) y rotate through carry right (RCR), donde se usan 33 bits para la rotación, donde uno de los bits es el indicador de acarreo.

Unidad de segmentación

Cuando la instrucción requiere acceso a memoria, entonces se usa esta unidad y las que le siguen.

La unidad de segmentación tiene 6 registros de segmento y 4 registros de direcciones del sistema.

Los registros de segmento tienen un campo visible para el programador y tres campos usados por el procesador. El campo base indica donde comienza el segmento, el campo límite indica el offset máximo que se puede utilizar para el segmento, es decir que la longitud del segmento es el límite más 1. Los atributos indican permisos y otras configuraciones del segmento, por ejemplo indican si el segmento es de 16 o 32 bits.

La carga de los registros de segmento depende del modo de operación del microprocesador: en el modo real, el campo base se carga con el valores del selector multiplicado por 10 hexa, mientras que los otros dos campos no se modifican. En el modo protegido, los valores de los tres campos se obtienen de tablas de descriptores que se encuentran en memoria, apuntados por los registros GDTR y LDTR. El elemento de la tabla y si se usa GDTR o LDTR, depende del contenido del selector que estamos cargando en el registro de segmento.

Una vez cargado el registro de segmento correspondiente, no hay diferencia entre los modos real o protegido. El procesador se limita a leer los tres campos que fueron cargados previamente.

El registro IDTR indica la dirección lineal dónde comienza la tabla de vectores de interrupción.

El registro TR se utiliza para implementar multitarea.

Unidad de paginación

La unidad de paginación solo funciona en modo protegido y se puede habilitarse o no, según convenga. Esta unidad traduce de direcciones lineales a direcciones físicas, que son las que usa el bus externo del procesador. Su uso principal es en sistemas operativos multitarea, donde los programas generalmente arrancan en la misma dirección lineal, pero deben estar ubicados en diferente espacio de RAM.

Otra posibildad de uso ocurre cuando el procesador opera en el modo virtual 8086. En este modo la carga de registros de segmento opera igual que en modo real. Pero como existe la posibilidad de tener varias tareas en el modo virtual 8086, y todas generan direcciones lineales de 0 a 10FFEF hexa, entonces la unidad de paginación se usa para que cada sesión se encuentre en una ubicación diferente de RAM. De esta manera corren las programas de DOS en sistemas operativos Windows de 32 bits.

Por último, en aplicaciones embebidas sin sistema operativo, se puede usar la unidad de paginación para detectar punteros nulos, deshabilitando la traducción de direcciones lineales a direcciones físicas para los primeros 4 MB, por ejemplo.

La traducción se basa en tablas que se encuentran en memoria apuntadas por el registro CR3. El registro CR2 lo carga el procesador con la dirección lineal que quiso convertir y no pudo debido a algún problema.

La tabla que se encuentra abajo que se llama TLB almacena las últimas traducciones hechas para no tener que acceder todo el tiempo a las tablas de paginación.

Direcciones

Aquí se pueden ver las diferentes direcciones que maneja el microprocesador: las direcciones lógicas, que están compuestas de selector y offset, son las que maneja el programador. La unidad de segmentación genera las direcciones lineales, que son la suma de la base del segmento y el offset. Por último, la unidad de paginación, si está habilitada, genera la dirección física, que va a la unidad de control del bus. Si la unidad de paginación no está habilitada, entonces la dirección física coincide con la lineal.

Bus

La unidad de control del bus genera las señales de control y opera el bus de direcciones y el de datos. El 80386 tiene 32 bits de direcciones y 32 de datos. El bus de direcciones tiene las líneas A31 hasta A2 y bank enable 0 a 3 negados. Como el proceador puede acceder indistintamente a 1, 2 o 4 bytes, entonces las memorias deben poder ser direccionadas por bytes. En el caso que vemos, hay cuatro memorias (pueden ser RAM o de sólo lectura) de 2^n bytes.

El decodificador, que es el que sirve para habilitar estas memorias, debe asegurarse que las líneas de control memory/input output negado y data/control negado estén a 1 y los bits más altos del bus de direcciones tengan el valor apropiado según el mapa de memoria del hardware.

La salida del decodificador es cero cuando queremos habilitar las memorias. Sin embargo, no queremos que se habiliten todas juntas. Entonces, el procesador posee cuatro pines bank enable negados para determinar cuáles de las cuatro memorias se van a habilitar para lectura o escritura. El bank enable 0 negado se activa con valor 0 cuando la dirección es múltiplo de 4. El bank enable 1 negado se activa con valor 0 cuando la dirección es múltiplo de 4 más 1. Y así sucesivamente.

En el caso de usar memoria RAM, tendremos que conectar el pin de control write/ read negado a las cuatro patas de output enable de las memorias.

Ejemplo 1

En este caso vamos a escribir un word en la dirección física 5. Esto significa que el byte bajo del word va a la dirección 5 y el byte alto del word va a la dirección 6.

Como 5 es igual a 0101 binario, entonces el valor que entrega el procesador a A_{n+1} hasta A_2 del bus de address es 1 hexadecimal. El procesador escribe en el bus de datos la información a grabar en D23 a D8. La entrada output enable negado de las memorias sigue al pin de control write/read negado, por lo que va a escribir en las memorias que se habiliten. Para habilitar la dirección física 5, el pin bank enable 1 negado se pone a cero. Para habilitar la dirección física 6, el pin bank enable 2 negado se pone a cero. Los otros bank enable negados quedan a uno para no habilitar las memorias que están en los extremos izquierdo y derecho.

Ejemplo 2a

En este caso vamos a escribir un double word en la dirección física 5. Esto quiere decir que hay que escribir en las direcciones 5, 6, 7 y 8. Como la dirección 8 tiene un valor diferente para el pin A2 del bus de direcciones, la unidad de control del bus automáticamente genera dos ciclos de memoria.

En la primera habilita las tres memorias de la izquierda, en forma similar al ejemplo anterior.

Ejemplo 2b

Ahora el bus de direcciones cambia de 1 hexa a 2 hexa y se habilita únicamente la memoria del extremo derecho poniendo bank enable 0 negado a cero.

Espero que esta explicación les haya servido. Hasta luego.

Ejemplo de modo protegido

por Dario Alejandro Alpern

Archivos necesarios

- video_prot.asm
- init_pci.inc

Transcripción

Hola. Mi nombre es Darío Alpern y hoy vamos a ver el ejemplo en modo protegido.

Video en modo texto

Como el ejemplo usa la pantalla, es necesario una breve introducción al video en modo texto.

En las PC, el video funciona en modo gráfico y en modo texto. Lo que vamos a ver ahora es modo texto.

En este modo, la pantalla posee 25 líneas de 80 columnas cada uno.

El sistema de video tiene una memoria RAM que es diferente de la RAM principal del sistema. Esta memoria es el buffer de video en el que el procesador escribe lo que el sistema de video debe mostrar en pantalla.

En el caso del modo texto, el buffer de video comienza en la dirección física 0xB8000. El buffer almacena primero la fila cero (fila superior), luego la fila uno y así sucesivamente hasta la fila 24, que es la fila inferior completando así 25 líneas.

Dentro de cada fila, se almacenan los 80 caracteres de izquierda a derecha.

Cada carácter ocupa dos bytes. El primero es el código ASCII del carácter y el segundo es el atributo donde se indica si el carácter parpadea y cuáles son los colores del fondo y del frente.

En este contexto, para el carácter que representa la letra "A", el frente son los puntos que forman la letra mientras que el fondo es el resto del rectángulo.

Enunciado del ejemplo

El ejemplo debe poder llenar las 25 filas de 80 caracteres. La fila superior contendrá la letra "A". La siguente fila, la letra "B" y así sucesivamente hasta la línea inferior, que contenga la letra "Y".

Se deberá usar atributo blanco sobre negro que tiene el código 0x07.

Usaremos registros como variables tales como números de fila, de columna, letra a mostrar y puntero.

Código fuente

Lo que queremos hacer ahora es editar el programa.

Hacemos CTRL++ para ver más grande, Apretamos F11 para ver los números de línea y acá podemos ver el código fuente.

En la línea 7 tenemos la directiva bits 16 que indica que lo que viene a continuación es código correspondiente a segmento de 16 bits. Eso es porque estamos en modo real.

Luego, viene la etiqueta inicio que es el destino del salto que se encuentra al final del segmento de la memoria ROM y luego un include, %include "init_pci.inc"

El archivo init_pci.inc contiene la inicialización del bus PCI y del sistema de video VGA.

Podemos abrirlo para ver qué contiene. Salimos del editor y ejecutamos kate init_pci.inc para verlo. Acá se acaba de abrir. La idea no es mostrar cómo funciona esto sino simplemente ver que el archivo es muy largo y no hace al ejercicio en sí. Entonces, estas 600 líneas más o menos, 681 líneas, son las que permiten inicializar el bus PCI y el sistema de video VGA.

Entonces, la idea es incluirlo directamente y lo que hace el nasm es, cuando llega la línea 10 automáticamente reemplaza el include por las 681 líneas. Es como si yo hubiese escrito 681 líneas en ese lugar.

Luego de eso, pongo un "magic breakpoint" con xchg bx,bx. La idea de esto es poder parar la ejecución luego de la inicialización del bus PCI y del sistema de video VGA.

Y luego hago un salto para saltearme toda la zona de datos que viene más abajo. O sea, hago un salto a pasaje_a_modo_protegido. que está definido más abajo.

Lo que vamos a ver a continuación es la definición de la GDT (Global Descriptor Table). Acá puse para optimización un align 8 que significa que lo que está a continuación va a estar en una dirección efectiva u offset múltiplo de 8.

Entonces arranca la GDT, y la GDT contiene descriptores. Cada descriptor ocupa 8 bytes. El primer descriptor no se utiliza entonces lo defino a cero. DQ sirve para definir 8 bytes, que es un define quadword. Se acuerdan que word son 16 bits, o sea 2 bytes. Entonces quadword son 8 bytes. Entonces, dejo 8 bytes libres que no me interesan.

Luego defino CS_SEL equ $-GDT Bueno, $-GDT vale 8, porque entre GDT y la definición de CS_SEL hay 8 bytes. Entonces CS_SEL vale 8, es el selector de code segment.

Luego de eso, está definido el descriptor correspondiente al segmento de código y la idea de este descriptor es que corresponda con un segmento de código que sea igual que en modo real, o sea que la base va a ser 0xFFFF0000 y el límite va a ser 0x0000FFFF, igual que en modo real, entonces vamos a ver cómo funciona esto. Como el límite se puede expresar dentro de los 20 bits que permite el descriptor, entonces la granularidad va a ser cero y el límite va a ser igual.

Entonces defino el límite con el DW (define word).

Luego vienen los dos bytes menos significativos de la base, a continuación, el siguiente byte más significativo, que tiene que ser 0xFF, o sea el tercer byte de la base, y luego los derechos de acceso que lo defino no en hexadecimal sino en binario un campo de bits, ven que acá está la "b". El sufijo "b" significa que el número está en binario.

Entonces, acá está el desgloce de la información: el bit 7 está encendido indicando que el segmento está presente, bits 6 y 5 están a cero indicando nivel de privilegio cero, o sea, máximo privilegio, bit 4 vale uno, para descriptores de segmento de código y datos, El bit 3 vale uno, para el caso de descriptores de código, el bit 2 vale cero porque es un segmento "no conforming", el bit 1 vale uno, porque está habilitado el permiso de lectura, el bit 0 de accedido vale cero, indicando que no se accedió al segmento.

Luego definimos el byte 6 de la siguiente manera: granularidad dijimos que vale cero, luego el bit "D" de default vale uno porque queremos segmento de 32 bits y luego límite 19 a 16 vale cero porque el bit 19 a 16 es el cero que está acá, entonces vale cero. Conclusión, el valor es 0x40.

Y por último tenemos la base 31 a 24 o sea el byte más significativo de la base tiene que estar a 0xFF.

Con esto definimos el descriptor de código. Ahora vamos a ver el descriptor de datos.

Para el descriptor de datos, yo quiero que sea un segmento flat. Los segmentos flat tienen base cero y límite 4 GB: 0xFFFFFFFF Entonces, decimos que como este límite supera el MB y termina en FFF ponemos granularidad a uno y el límite son FFFFF solamente. Acuérdense que el procesador agrega automáticamente FFF a la derecha.

A partir de lo que dijimos recién, definimos la parte baja del límite como 0xFFFF Luego viene la parte baja de la base, los dos bytes menos significativos valen cero, porque la base vale cero, El tercer byte de la base también vale cero, los derechos de acceso, que es un campo de bits, lo definimos de una manera similar que antes con el bit de presente a uno, DPL vale cero, el bit 4 indicando que es un descriptor de código o datos vale uno, el bit 3 es diferente, es cero porque son datos, el bit 2 vale cero porque es la dirección de expansión normal, el bit 1 vale uno porque está habilitada la lectura y la escritura, y el bit cero lo ponemos también a cero como dijimos que tenemos que hacer siempre con este bit.

Luego tenemos el byte 6 que indicamos granularidad uno, como dijimos más arriba el bit de default va a uno porque es 32 bits, y el límite 19-16 lo ponemos a F.

Y finalmente tenemos base 31 a 24 que es el byte más significativo de la base, lo ponemos a cero.

Luego definimos un símbolo que es el tamaño de la GDT, tam_GDT equ $-GDT. De esta manera, si yo por ejemplo agrego más descriptores tam_GDT va a ir incrementándose y no tengo que hacer ningún cálculo. Automáticamente, el ensamblador calcula el tamaño correcto del símbolo tam_GDT.

A continuación definimos la imagen de GDTR donde primero va el límite y después va la base. El límite es el tamaño de la GDT menos 1. La longitud del campo límite es de 16 bits y por eso se usa DW (define word). y luego pongo la base de la GDT que es la dirección lineal donde comienza la GDT, es 0xFFFF0000 más GDT, donde GDT, este símbolo indica el offset entonces le sumo la dirección lineal del arranque del segmento de código más ese offset, y me da justamente la dirección lineal de la GDT.

A continuación vemos pasaje a modo protegido.

Lo primero que hay que hacer es deshabilitar interrupciones como dijimos en la teoría.

Luego cargamos el registro GDTR apuntando a la imagen de GDTR que está más arriba, digamos por acá. Y fíjense que a la instrucción LGDT le agrego o32. Este o32 permite utilizar los cuatro bytes de la base. Si yo no usara o32, por un tema de compatibilidad con el 80286, utilizaría solo tres bytes de la base. Entonces no me sirve porque no puedo acceder a la parte superior porque en el byte más significativo tengo 0xFF. y si yo no usara o32 me pondría 00 ahí. Entonces necesito o32 sí o sí.

Una vez que está cargado el registro GDTR, pasamos a modo protegido poniendo a uno el bit cero del registro de control CR0.

Y ahora, lo que vamos a hacer es cambiar el segmento de código, para que apunte al segmento de código de modo protegido. el segmento de 32 bits que definimos en la GDT. Para eso hacemos un salto intersegmento donde el selector es CS_SEL que fue el definido más arriba en la GDT y el offset es inicio_32 que es el que está acá a continuación. Esto funciona porque la base del selector de código nuevo es igual a la base del segmento de código que estaba definido en modo real. Si no, habría que utilizar otro esquema diferente.

Una vez que hicimos el salto intersegmento, lo que viene a continuación es de 32 bits Como a continuación del salto intersegmento viene código de 32 bits, yo tengo que poner la directiva BITS 32 que es lo mismo que haber puesto use 32.

Entonces, defino la etiqueta inicio_32 y a partir de este momento ya puedo acceder al segmento de 32 bits de código.

A continuación cargo el registro DS con el selector que apunta al descriptor correspondiente al segmento flat de 4 GB de tamaño que arranca en la dirección lineal cero.

Y a continuación tengo la lógica para acceder al buffer de video.

Entonces, ¿cómo voy a hacer para operar con el buffer de video? Bueno, inicializo un registro que me indique la letra que voy a escribir. Por ejemplo el registro BL, lo cargo con la letra "A". Fíjense que el ensamblador, le puedo poner la letra "A" directamente y me lo traduce al ASCII correspondiente que es 0x41.

Luego de eso, cargo el número de fila en el registro CL y luego el puntero en 0xB8000. Fíjense que ahora que estamos en modo protegido podemos usar punteros que estén por arriba de los 64 KB. cosa que en modo real no se podía hacer. Entonces ESI apunta directamente a la dirección lineal donde se encuentra el buffer de video que coincide con la dirección física porque no está activada la paginación.

Luego, dentro de ciclo_externo vamos a definir la columna, la columna la arrancamos en cero, vamos a comenzar con la columna que está a la izquierda.

Y luego entramos en el ciclo que procesa cada carácter dentro de cada fila. Lo que hago es cargar en el puntero que apunta al buffer de video el carácter que quiero mostrar, la letra "A". Y a continuación cargo el atributo que es 0x07 que es blanco sobre negro. Fíjense que la cláusula byte es importante porque el operando a la derecha no tiene tamaño, es un número, no es un registro Entonces tengo que definir que la operación es de tipo byte.

Una vez que ya escribí el carácter y el atributo, incremento el puntero en dos lugares, luego incremento el número de columna y luego tengo que verificar si el número de columna coincide con el final de la fila. Cada fila tiene 80 columnas. Lo que quiero ver es si se terminaron las 80 columnas. Entonces, comparo contra 80 y si no es igual, JNE, "jump not equal", vuelvo al ciclo interno para terminar de completar las 80 columnas.

Una vez que terminó el ciclo de la fila, incremento el número de fila incremento el código ASCII del carácter, o sea la primera vez salta de "A" a "B", digamos, y luego hago la misma operación pero verificando si llegué a la última fila, o sea si llegué a la 25 y si no es igual, me voy al ciclo externo para seguir con la siguiente fila.

Una vez que están los caracteres puestos en el buffer de video. se ejecuta un JMP $ para que se cuelgue el código y no siga ejecutando más allá, porque ya terminó lo que debía hacer el enunciado.

Luego de eso, seguimos con lo que hacíamos siempre, que es hacer el primer relleno.

Después, esto es importante el bits 16 porque el JMP que va a continuación es un JMP de modo real. Por lo tanto, tiene que estar precedido por bits 16. Porque veníamos de código de 32 bits, antes. Entonces ejecutamos el JMP que es el que se ejecuta cuando arranca el procesador.

Y luego ponemos el segundo relleno para completar la imagen de ROM de 64 KB. Salimos del editor de texto.

Ejecución del programa de ejemplo

Vamos a compilar.

Fíjense que el archivo de salida es mi_rom.bin y la idea de eso es, justamente, modificar el archivo bochs.cfg para que la imagen de rom siempre sea mi_rom.bin así no tengo que modificar todas las veces el bochs.cfg.

Entonces, compilo, no tiró ningún error. Y ahora vamos a correr el Bochs para ver qué ocurre. Bien, ahí arrancó el programa.

Estamos en el JMP correspondiente a la instrucción inicial. Hacemos "step". Y ahora tenemos otro JMP que sirve para saltearme toda la zona de datos que viene a continuación.

Lo que vamos a hacer es seguir hasta el magic breakpoint que habíamos puesto después de la inicialización de PCI y VGA apretando el botón "continue" Ahí está. El magic breakpoint salteó toda la inicialización del bus PCI y del sistema de video VGA.

Fíjense que a continuación del JMP se encuentra la GDT, que el Bochs lo interpreta como si fueran instrucciones ejecutables y no lo son. Fíjense que tienen cualquier cosa incluyendo instrucciones inválidas.

Le doy "step" y salteo lo que es la GDT.

Ahora va a cargar el registro GDTR con imagen_GDTR que se encuentra en la dirección efectiva 0x1340,

Ahora aprieto el botón "step" y ahora podemos ver en el panel de la izquierda yendo hacia abajo información del GDTR. Entonces dice: GDTR apunta a la GDT entonces la GDT arranca en 0xFFFF1328 y el límite es 0x17, que es el valor esperado.

Yo puedo ver los valores que están en la GDT viendo con Linear Memdump por ejemplo.

Agrandamos el panel de la derecha para ver más información. La GDT arranca en la segunda mitad de la primera línea que dice address 0xFFFF1320, ya que cada línea muestra 16 bytes. Acá vemos el descriptor nulo 8 bytes a cero Después a continuación en 1330 tenemos el primer descriptor, que es el descriptor de código, con los bytes que ingresamos y después el segundo descriptor que es el descriptor de datos. Obviamente, no está muy inteligible.

Entonces, lo que podemos hacer para entender lo que sucede es poner info gdt.

info gdt me devuelve tres descriptores. GDT[0], GDT[1] y GDT[2].

El primer descriptor es el nulo, así que no lo puede decodificar.

El descriptor 1 es el de código, con base 0xFFFF0000 y límite 0x0000FFFF y en los atributos se puede ver que es de 32 bits.

El descriptor 2 es de datos que apunta a un segmento flat que tiene base cero y límite 4 GB y se puede escribir.

Ajusto el panel central para que se vean las instrucciones y ahora vamos a ingresar a modo protegido. Ahora teóricamente ya estamos en modo protegido porque CR0 le habilitamos el bit menos significativo ven que acá hay un uno.

Y ahora vamos a saltar al selector 8 y offset 0x135D Entonces ahora va a saltar a otro segmento pero en realidad el segmento de código ese coincide con el segmento de código que venimos corriendo. Cuando aprieto el botón "step" continúa en la instrucción siguiente que está en la dirección efectiva 0x135D Y las instrucciones ahora se ven bien porque el Bochs desensambla el código correspondiente a segmento de 32 bits.

Ahora vamos a cargar el valor 10 en el registro DS.

Ahora vamos a ver qué contienen los registros de segmento escribiendo sreg en el panel inferior, que muestra para cada uno de los seis registros de segmento la base, el límite y los atributos. Vamos a ver el contenido de los registros CS y DS.

En el caso de CS, tenemos que el selector es 8 que es lo que se cargó con el JMP intersegmento, después, la base es el que pusimos nosotros, el límite también, y los atributos que habíamos visto en la GDT.

En el caso de DS el selector es 0x10 y tenemos base y límite esperados para un segmento flat.

Luego, seguimos ya tenemos CS y DS Entonces podemos seguir.

BL se carga con 0x41 como vemos acá que el ASCII de la letra "A".

Después, el número de fila, que se almacena en el registro CL, se carga con cero.

ESI apunta a 0xB8000. Ya se cargó.

Después CH se carga con cero, que es el número de columna.

Ahora el procesador va a escribir el primer carácter en el buffer de video que corresponde al extremo superior izquierdo de la pantalla.

Ponemos el carácter "A" y el atributo blanco sobre negro correspondiente al primer carácter.

Si yo quiero ver que fue lo que escribió en pantalla, hay que ir a la ventana de salida de video del Bochs con los botones ALT-TAB hasta llegar a esa ventana. Ahí se ve la letra "A".

Una vez que vimos que la letra "A" se colocó correctamente Hacemos "step", el puntero se incrementó a 0xB8002 Ahora apunta a la fila 0, columna 1, Ahora incrementamos el número de columna, verifico si CH vale 80 o 0x50 Obviamente no vale Fíjense que el flag de cero está a cero Significa que no es igual porque CH vale uno y yo comparé contra 0x50 entonces no son iguales.

Si yo quiero ejecutar hasta completar la primera línea, me posiciono en la primera instrucción después de ciclo_interno y hago doble click y se ve la instrucción que marca el punto de parada en rojo. Luego, aprieto el botón de "continue" para que se llene la línea.

Para verificar que se llenó la línea superior, vamos a ver la salida del video simulado del Bochs. Y acá se pueden ver todas letras "A" que son 80 en la primera línea.

Si yo le doy "continue", me llena la segunda línea que tendría que ser con letras "B". Vamos a ver si eso anda correctamente Ahí se ven las letras "A"y "B".

Mientras tanto el puntero sigue incrementándose. Cada fila son 160 bytes porque son 80 columnas por 2.

Lo que vamos a hacer ahora es sacar el punto de parada y continuar hasta que se cuelgue que es el JMP que está en el 1387. Continuamos y le damos "break" Se para en el JMP y acá podemos ver en la pantalla simulada del Bochs todas las letras de la "A" a la "Y" en toda la pantalla completa, que es lo queríamos hacer.

Creo que con esto es suficiente por ahora. Hasta luego.

Excepciones en modo protegido

por Dario Alejandro Alpern

Transcripción

Hola. Mi nombre es Darío Alpern y hoy vamos a ver excepciones en modo protegido.

Excepción

Una excepción es el cambio temporario de flujo del programa que se está ejecutando debido a un evento interno al procesador, por ejemplo intentar ejecutar una instrucción que no existe, acceder a memoria más allá del límite del segmento, etc.

En el diagrama se puede ver el programa principal que está corriendo hasta que llega ese evento.

En ese momento el procesador no continúa ejecutando el programa en curso. En cambio, ejecuta el manejador de la excepción, que es una subrutina cuyo contenido depende del evento que debemos procesar.

Luego de terminar de ejecutar el manejador de excepción, el procesador continúa ejecutando el programa principal desde la instrucción que generó la excepción, o la siguiente, dependiendo del tipo de excepción.

Es importante que el manejador de excepción salve el contenido de los registros cuando entra y los recupere antes de salir, ya que en caso contrario, los valores de los registros no serían los esperados y se colgaría el programa o ejecutaría algo no esperado.

Clasificación

En los procesadores Intel existen tres tipos de excepciones: faults, traps y aborts.

Las excepciones tipo fault ocurren cuando hay un fallo en el programa y el objetivo del manejador es corregirlo. Cuando termina el manejador de la excepción mediante la instrucción IRET, el procesador va a volver a intentar ejecutar la instrucción que generó la excepción. Por ejemplo, en un sistema que use memoria virtual, con la unidad de paginación activada, parte de la datos puede encontrarse fuera de la memoria física del sistema, en un disco rígido.

Si una instrucción intenta acceder a una página que no está presente en memoria física, ocurrirá una excepción de fallo de página donde el procesador indica que la página no está presente. En este caso el manejador de la excepción de página deberá recuperar del disco rígido los datos requeridos por la instrucción que falló para ponerlos en memoria. Una vez hecho esto el manejador ejecuta la instrucción IRET para finalizar, y luego el procesador vuelve a ejecutar la instrucción, y esta vez no debería fallar.

Las excepciones tipo trap ocurren cuando termina la instrucción en curso y por eso cuando el procesador termina la ejecución del manejador de excepción, va a ejecutar la instrucción siguiente a la que generó la excepción.

Las excepciones tipo abort ocurren cuando hay fallos graves en el sistema y no se puede recuperar. En ese caso, no tiene sentido ejecutar IRET en el manejador de dicha excepción. Generalmente lo que se hace es mostrar los registros e información propia del sistema en pantalla o se guarda en disco, y luego se bloquea el sistema, esperando que el usuario reinicialice el equipo.

Faults

Aquí se ve la lista completa de las excepciones tipo fault. A la izquierda se puede ver el tipo de excepción, tanto en hexadecimal como en decimal. Como en el caso de las interrupciones, el tipo de excepción es el índice dentro de la tabla IDT donde se encuentra la compuerta que tiene la dirección lineal del manejador de la excepción.

En la tercera columna se encuentra un mnemónico de dos letras que identifica la excepción y en la columna de la derecha se puede ver una somera descripción que indica cuando ocurre la excepción.

La excepción tipo cero ocurre cuando hay una división por cero, o bien el resultado no entra en el cociente.

La excepción tipo 1 ocurre cuando se usan los registros de debug DR0 a DR3, DR6 y DR7 para poner breakpoints por ejecución de código. El manejador de la excepción tipo 1 pertenece generalmente a un debugger y debe almacenar los registros para ponerlos en la pantalla.

La excepción tipo 5 ocurre cuando se ejecuta una instrucción BOUND. El primer operando de esta instrucción es un registro y el segundo es un puntero a memoria donde se almacena el límite inferior y luego el superior. Si el valor del registro no se encuentra en el rango especificado, se produce esta excepción.

La excepción tipo 6 ocurre cuando el código de operación no existe o si el tamaño de la instrucción es mayor que 15 bytes. Esta última condición solo puede ocurrir si la instrucción tiene prefijos repetidos.

La excepción tipo 7 ocurre cuando se ejecuta una instrucción matemática o SIMD luego de un cambio de tarea y determinados bits del registro de control CR0 y CR4 están encendidos.

La excepción tipo 10 se da cuando hay errores en la TSS, que es una estructura que se usa para implementar multitarea.

La excepción tipo 11 ocurre cuando se desea acceder un segmento marcado como no presente, en el momento de cargar el registro de segmento.

La excepción tipo 12 ocurre cuando hay problemas con la pila, como PUSH, POP, llamadas o retornos de subrutinas, interrupciones o excepciones que escriban o lean más allá del límite del segmento de pila. Por ejemplo: MOV ESP,0 y luego PUSH EAX.

La excepción tipo 13 indica un error general de protección, como por ejemplo cargar un registro de segmento con un selector que apunte a un descriptor inválido.

La excepción tipo 14 indica un fallo de página, por ejemplo acceder una página no presente, intentar escribir en una página de solo lectura, etc. Para que ocurra esta excepción, la unidad de paginación debe estar activada.

La excepción tipo 16 ocurre cuando existen errores en instrucciones matemáticas de punto flotante. Algunos errores son: números muy pequeños o muy grandes, división por cero, desbordamiento de la pila de números o pérdida de precisión.

La excepción tipo 17 se da cuando un código que corre con nivel de privilegio 3 accede a datos que no están alineados.

La excepción tipo 19 ocurre cuando hay un error en una instrucción de punto flotante de SIMD. Los errores son similares a los que generan la excepción tipo 16.

La excepción tipo 20 ocurre cuando hay un error de virtualización. La virtualización es una característica del procesador que permite correr un sistema operativo (llamado invitado) dentro de otro (llamado anfitrión).

La excepción tipo 21 ocurre cuando hay un ataque al sistema si está activada la característica Control Flow Enforcement (control de cumplimiento de flujo), que posee varias herramientas para detener ataques al sistema operativo. En uno de ellos, el procesador posee una pila oculta donde solo se almacenan direcciones de retorno de subrutinas, interrupciones o excepciones. Los programas no pueden acceder a esta pila porque se encuentran en páginas especiales no accesibles por la aplicación. Cuando se ejecuta una instrucción de retorno, el procesador verifica que en ambas pilas se encuentre la misma dirección. Si eso no ocurre, se dispara la excepción tipo 21.

Existen dos excepciones más tipo fault en procesadores AMD, la 29 y la 30.

Traps

Aquí se pueden observar las tres excepciones tipo trap.

La excepción tipo 1 ocurre si se usan los registros de debug para poner un breakpoint por lectura o escritura de datos, o cuando el bit 8 de EFLAGS, que es el trap flag, está encendido. Esto último permite ejecutar paso a paso usando un debugger, aún cuando el procesador está corriendo en memoria de solo lectura.

La excepción tipo 3 ocurre cuando se utiliza el byte CC hexa para implementar breakpoints. El debugger salva el primer byte de la instrucción donde debe ir el breakpoint y luego escribe el byte 0xCC hexa en dicha posición de memoria. Cuando el procesador ejecuta ese byte 0xCC lo interpreta como INT 3 y se ejecuta el manejador de esta excepción, que debe estar capturado por el debugger.

La excepción tipo 4 ocurre cuando el procesador ejecuta la instrucción INTO (interrupt on overflow) y el indicador de sobrepasamiento (bit 11 de EFLAGS) está encendido.

Aborts

Aquí se pueden observar las dos excepciones tipo abort.

La excepción tipo 8 ocurre cuando el procesador está vectorizando una excepción y en ese momento ocurre otra. Esto no pasa siempre. Más adelante en este video vamos a ver cuándo ocurre.

La excepción tipo 18 ocurre cuando hay un error interno del procesador, del bus o si un hardware externo detecta que hay error en el bus.

IDT

Aquí se puede ver la tabla de vectores de excepción que usa el procesador cuando está en modo protegido.

La tabla de descriptores de interrupción (en inglés la sigla es IDT (Interrupt Descriptor Table) tiene una compuerta por cada tipo de interrupción o excepción. De los 256 tipos diferentes que soporta la IDT, los 32 primeros están reservados para excepciones, excepto la entrada número 2, que se usa para las interrupciones no enmascarables. El procesador tiene una pata NMI que si se activa, se ejecuta el manejador de esta entrada número 2 independientemente del estado del bit 9 de EFLAGS, que es el indicador de habilitación de interrupciones. El registro IDTR apunta a la compuerta correspondiente al tipo de excepción cero. A continuación se encuentra la compuerta correspondiente al tipo de excepción 1, y así sucesivamente hasta la excepción 31. Más arriba en la IDT estarán las compuertas de interrupción.

Las compuertas son direcciones lógicas que se componen de selector y offset y tienen el formato que se muestra a la derecha. El tipo de compuerta en el caso de excepciones es 1111 binario. La dirección lógica apunta al inicio del manejador de la excepción.

Uso de la pila

En este diagrama se puede observar lo que el procesador envía a la pila cuando ocurre la excepción. Cada elemento de la pila tiene 32 bits. Como las direcciones crecen hacia arriba en el diagrama, se puede comprobar que cuando el procesador pone estos registros e información adicional en la pila, el puntero ESP se decrementa. Al ejecutar la instrucción IRET, el puntero ESP se incrementa.

Como vamos a ver en multitarea, cada nivel de privilegio tiene su pila asociada. Cuando hay un cambio de nivel de privilegio, generalmente de privilegio 3 a privilegio cero, el procesador deja de usar la pila de privilegio 3 y pone en la nuevo pila (la de privilegio cero) la información que figura en el diagrama. Eso incluye la dirección lógica de la pila de nivel de privilegio 3.

Dependiendo de la excepción, el procesador puede poner o no un valor en la pila denominado "código de error".

Código de error

Ahora vamos a ver en más detalle el código de error.

Varias excepciones no ponen código de error en la pila: 0 a 7, 16, 18, 19 y 20. Otras ponen simplemente el valor cero, como las excepciones 8 y 17.

Aquí se puede ver el código de error correspondiente a las excepciones que se refieren a errores de segmentación. Cuando no se puede determinar el descriptor correspondiente al problema que generó la excepción, el código de error vale cero.

Fallo de página

En este diagrama se puede ver el formato particular del código de error para la excepción 14 de fallo de página. Aparte de poner el código de error en la pila, el procesador carga en el registro de control CR2 la dirección lineal que causó el fallo de página.

Doble falta

Como dijimos antes, la doble falta se genera cuando durante la vectorización de una excepción ocurre una segunda excepción, bajo determinadas condiciones.

Para saber cuáles son esas condiciones, dividiremos las excepciones en tres grupos: fallo de página, excepciones contributivas y excepciones benignas. Esta clasificación no tiene ninguna relación con la clasificación de excepciones en fault, trap y abort que vimos antes.

Tabla generación doble falta

En la tabla se puede ver en qué casos ocurre la doble falta. En la columna izquierda figura la excepción que se está vectorizando, es decir la primera interrupción, mientras que en la fila superior se puede ver la excepción que ocurre durante esa vectorización, es decir la segunda excepción.

Si ocurre una excepción contributiva o fallo de página durante la vectorización de la excepción de doble falta, ocurre un shutdown. En la PC el shutdown genera un reset del procesador. Por eso es bastante habitual cuando hay errores de código ver que el sistema se reinicia.

Ejemplo de shutdown

Por ejemplo, si ejecuto la instrucción MOV ESP,1 y luego PUSH EAX, el registro EAX iría en el offset 1 - 4 = 0xFFFFFFFD. No se pueden poner los cuatro bytes en la pila porque se supera el límite. Como estamos trabajando con la pila, se genera la excepción 12.

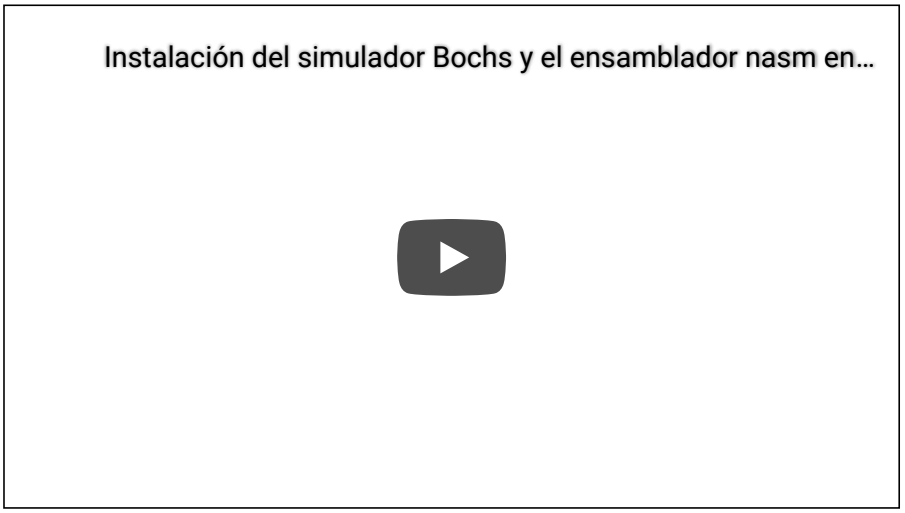
Al intentar vectorizar la excepción 12, el procesador va a intentar guardar en la pila la dirección de retorno y no va a poder, porque se supera el límite del segmento de pila igual que antes.

Como son dos excepciones contributivas, se genera la excepción 8 de doble falta. Pero otra vez tampoco se puede vectorizar, lo que genera el shutdown y posterior reset.

Espero que esta explicación les haya sido útil. Hasta luego.

Instalación del simulador Bochs

por Dario Alejandro Alpern



Transcripción

Hola. Mi nombre es Darío Alpern y hoy vamos a ver cómo se instalan el simulador Bochs y el ensamblador nasm en Linux.

Ahora vamos a instalar las herramientas que vamos a usar para compilar y ejecutar nuestros programas en Assembler en un ambiente simulado, como es el Bochs.

Escribimos sourceforge.net/projects/bochs en la casilla de URL. Luego elegimos Files para ver los archivos que se pueden bajar. Elegimos Bochs y luego 2.6.9. El archivo a bajar debe ser bochs-2.6.9.tar.gz.

Salvamos el archivo y abrimos la consola.

Creamos el directorio Bochs y vamos ahí. Luego copiamos el archivo que acabamos de bajar y lo descomprimimos.

A continuación vamos a usar apt-get para obtener los paquetes necesarios. Siempre que operamos con apt-get debemos hacer un update para obtener los archivos más recientes.

Luego indicamos cuáles son los paquetes a instalar:

- **nasm** es el ensamblador, que nos va a servir para compilar nuestros fuentes en Assembler.
- **build-essential** baja el compilador gcc y otros comandos como make que son necesarios para compilar Bochs en nuestra máquina.
- **gtk-2.0** y **libgtk2.0-dev** instalan el GTK que es una biblioteca de componentes gráficos multiplataforma para desarrollar interfaces gráficas de usuario. Dependiendo de la distribución de Linux instalada puede estar presente o no antes de instalar Bochs. El debugger gráfico de Bochs requiere este paquete.

Una vez que tenemos los paquetes instalados, vamos al directorio donde se encuentra el código fuente, que es bochs-2.6.9

Con el script `configure` indicamos cuáles son las características del Bochs que necesitamos.

- **--enable-x86-64** habilita los registros e instrucciones de 64 bits.
- **--enable-smp** permite ejecución multithread en el simulador.
- **--enable-usb** y **--enable-usb-ohci** permiten usar dispositivos USB en el simulador.
- **--enable-disasm** hace que Bochs pueda mostrar el desensamblado de código.
- **--enable-debugger y** --enable-x86-debugger permiten usar debugger en el Bochs. Es una característica muy importante porque nos va a ayudar a descubrir los errores de nuestro código.
- **--disable-docbook** hace que no se compile código para documentación en el Bochs. Es algo que nosotros no necesitamos.

Con el comando `make` compilamos el simulador Bochs usando la configuración que acabamos de escribir.

Finalmente instalamos Bochs en el sistema usando el comando `sudo make install`.

Para asegurar que el simulador se instaló, corremos bochs y vemos que aparecen errores. Esto es correcto porque nos falta el archivo de configuración del Bochs.

Ahora creamos un directorio de proyecto y copiamos el archivo de configuración por defecto en nuestro directorio. Tenemos que hacer modificaciones en dicho archivo así que lo vamos a editar con la aplicación kate.

Aquí se puede ver el cambio necesario para poder usar el debugger gráfico.

Aquí se puede ver que comenté la imagen del disco rígido con el signo numeral ya que nosotros no tenemos que simular disco rígido.

También comenté la directiva sound, ya que no usamos sonidos.

Por último modifiqué la imagen de ROM para que contenga el nombre del archivo binario que acabamos de generar.

Cuando tengamos que correr el Bochs usaremos la línea de comando `bochs -qf bochs.cfg` que utiliza el archivo de configuración que acabamos de editar.