Ambiente de operación de la CPU x86 en modo protegido

Esquema de administración de la memoria

Cada descriptor descriptor de segmento especifica (entre otras cosas) la dirección base, el tipo y el tamaño límite de un segmento. Una dirección lógica está compuesta de un selector de segmento y un offset de 32 bits. La dirección lineal es la dirección base del segmento más el offset. Con solamente segmentación, una dirección lineal es también la dirección física. Si la paginación está habilitada, la CPU traduce la dirección lineal a dirección física a través de dos niveles de tablas de página.

Excepciones e interrupciones

En el modo real los vectores de excepción y de interrupción están en el 1KB bajo de la memoria de modo real. En el modo protegido, las excepciones e interrupciones están representadas por puertas de trampa y de interrupción en una tabla de descriptores de interrupción. La CPU usa la tabla de descriptores de interrupción para el procesamiento de excepciones e interrupciones. Las tablas de descriptores deben ser establecidas apropiadamente antes de “switchear” la CPU al modo protegido. Si se usa paginación, las tablas de páginas también deben ser establecidas antes de habilitar el hardware de paginación.

Administración de memoria en modo protegido

La administración de memoria consiste de traducción de dirección y de protección de memoria. En el modo protegido, el hardware de administración de memoria de la CPU x86 soporta segmentación y paginación.

14.2.1 Segmentación

Una CPU x86 en modo protegido tiene 6 registros de segmento, denotados por cs, ds, ss, es, fs y gs. Cada registro de segmento es de 16 bits, pero su contenido ya no es una dirección base como en el modo real. En lugar de ello, especifica el índice de un descriptor de segmento en una tabla de descriptores. Cada descriptor de segmento contiene la dirección base y el tamaño límite del segmento que se describe con él. La dirección lineal, la cual es también la dirección física es la dirección base del segmento más el offset. El formato de un registro de segmento se muestra en la Fig. 14.1

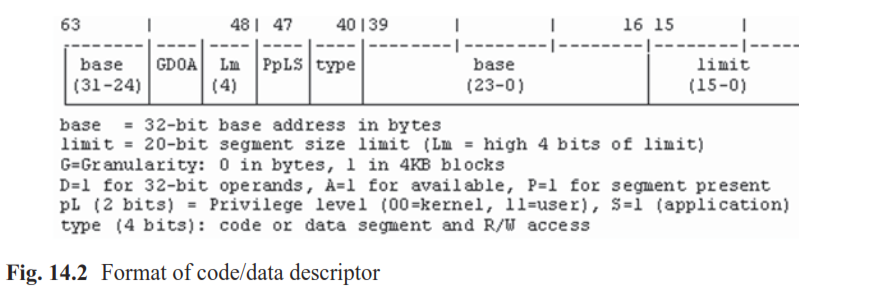


en la cual el index es un offset de 13 bits en una tabla de descriptores, T=0 significa que el selector se refiere a la Global Descriptor Table (GDT), T=1 significa que el selector se refiere a una Local Descriptor Table (LDT), RPL –Run Privilege Level-- es el nivel de privilegio del segmento para protección. Los dos bits del nivel de privilegio varían desde 00, el cual es el nivel más alto, hasta 11, el cual es el nivel más bajo. Los cuatro niveles de privilegio forman un conjunto de anillos de protección, los cuales pueden ser usados para implementar sistemas operativos seguros con múltiples capas de protección. Todos los sistemas tipo Unix usan solamente dos niveles de privilegio; el nivel kernel y el nivel usuario. Para MTX en modo protegido, se usarán dos niveles de privilegio, RPL=0 para el modo kernel y 3 para el modo usuario. Cuando un proceso se ejecuta en el nivel de privilegio 0, éste puede ejecutar cualquier segmento de código y acceder a cualquier segmento de datos. Cuando un proceso se ejecuta en el nivel de privilegio 3, éste no puede acceder a cualquier segmento de RPL 0 directamente. Esto hace que los programas de modo usuario no puedan ejecutar código kernel o acceder a datos del kernel. Como es usual, un proceso de modo usuario puede entrar al modo kernel solamente a través de interrupciones, excepciones, o haciendo explícitamente llamadas al sistema (syscalls).

En el modo protegido, una dirección lógica está compuesta de dos partes: un selector de segmento de 16 bits y un offset de 32 bits, el cual especifica la dirección relativa dentro del segmento. Dada una dirección lógica LA = [segment : offset], la CPU usa el bit T del selector de segmento para acceder ya sea a la GDT o a una LDT. Si T=0, usa la GDT, a la cual apunta el registro GDTR de la CPU. Si T=1, usa una LDT a la cual apunta el registro LDTR de la CPU. Un sistema tiene solamente una GDT, la cual especifica los segmentos de código y de datos del kernel que son comunes para todos los procesos. Cada proceso podría tener su propia LDT, la cual especifica el espacio de direcciones de modo usuario de ese proceso.

14.2.1.1 Descriptores de segmento

La Figura 14.2 muestra el formato de un descriptor de segmento de código o de datos, cada descriptor de segmento es de 8 bytes.



Los descriptores de segmento están ya sea en la GDT o en una LDT. El registro GDTR de la CPU apunta a un descriptor de GDT (no confundir con descriptor de segmento) que contiene la dirección y tamaño de la GDT (en realidad el descriptor de GDT contiene la dirección de la GDT y la cantidad de descriptores en la GDT menos 1). Similarmente, el registro LDTR apunta a un selector de LDT en la GDT, el cual apunta a la LDT.

Cuando se establecen la GDT y la LDT es conveniente definir unos cuantos prototipos de descriptores de segmento. Por ejemplo, los siguientes descriptores definen segmentos de código y de datos de 4GB de GDT (límite = 0xFFFFF, y granularidad página G=1, i.e. granularidad de 4KB=4096=0x1000=2^{12} bytes), entonces

Tamaño de segmento = (límite+1) \* (granularidad página)

= 0x100000 \* 0x1000 = 2^{20} \* 2^{12}

= 2^{32} = 4GB



Similarmente, los siguientes descriptores definen dos segmentos de código y de datos de 2MB de una LDT (límite = 0x001FF, y granularidad página, i.e. granularidad de 4KB=4096=0x1000=2^{12} bytes), entonces

Tamaño de segmento = (límite+1) \* (granularidad página)

= (0x001FF+1) \* 0x1000 = 0x00200 \* 0x1000 = 2^{9} \* 2^{12}

= 2^{21} = 2^{20}\*2 = (1MB) \* 2 = 2MB



Para diferentes direcciones base, simplemente hay que cambiar el campo dirección base (el byte 7 y los bytes del 4 al 2, los bytes se enumeran del 0 al 7 de derecha a izquierda y recordemos que dos cifras hexadecimales corresponden a un byte) en el prototipo de descriptor. Similarmente, se puede cambiar los campos límite (nibble bajo del byte 6 y los bytes 1 y 0) para obtener diferentes tamaños de segmento. Además de descriptores de segmento de código y de datos, la GDT también contiene Descriptores de Segmento eStado de Tarea (Task State Segment Descriptor, TSSD). Un TSSD se refiere a un Segmento de eStado de Tarea (Task State Segment, TSS), el cual es una estructura de datos usada por la CPU para guardar los registros de la CPU durante un “switcheo” de tarea por hardware. Este segmento también especifica la dirección lógica de las pilas para los niveles de privilegio 0, 1 y 2.

14.2.1.2 Modelos de Segmentación

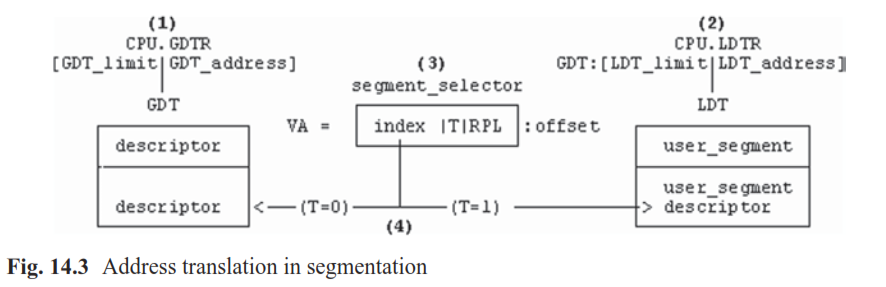
La administración de memoria por segmentación puede utilizar varios modelos de memoria diferentes:

El modelo plano (Flat model): En el modelo plano, todos los segmentos tienen base = 0, G = 1 (4KB) y limit = 0xFFFFF (1MB). En este caso, el mapeo de dirección virtual a dirección física es uno a uno. Una dirección virtual de 32 bits es un offset desde 0, así que también es una dirección física. Un uso especial del modelo plano es para paginación. Antes de habilitar el hardware de paginación, todos los segmentos son establecidos al tamaño máximo de tal manera que el rango de direcciones lineales es de 4 GB.

El modelo plano protegido (Protected flat model): En el modleo plano protegido, los segmentos son mapeados a memoria existente estableciendo el tamaño límite al tamaño de la memoria física disponible. Cualquier intento de acceder a memoria fuera del límite de segmento generará un error de protección.

El modelo multi-segmento (Multi-segment model): En el modelo multi-segmento, un programa podría usar todos los registros de segmento para acceder a hasta 6 segmentos protegidos.

La Figura 14.3 muestra la traducción de memoria de la CPU y el esquema de protección de memoria en segmentación, los cuales son explicados con las etiquetas de la (1) a la (4).



1. El registro GDTR de la CPU contiene un descriptor de GDT = [GDT\_limit | GDT\_address]. La GDT contiene descriptores de segmento globales de 8 bytes. Cada descriptor de segmento tiene una dirección base de 32 bits, un tamaño límite y un nivel de privilegio de 2 bits.
2. Como la GDT, la LDT también contiene descriptores de segmentos locales de 8 bytes. A diferencia de la GDT, la cual puede estar localizada en cualquier parte en la memoria, el descriptor LDT debe estar colocado en la GDT y cargar el registro LDTR con su selector en la GDT.
3. Una dirección virtual VA = [segment\_selector de 16 bits : offset de 32 bits].
4. El index de 13 bits es usado para acceder a un descriptor de segmento ya sea en la GDT (T=0) o en una LDT (T=1). Un programa se ejecuta en el nivel de privilegio (CPL, Current Privilege Level) de su selector de segmento de código. Cuando un programa usa un selector (RPL, Required Privilege Level) para acceder a un descriptor de segmento de datos (DPL, Descriptor Privilege Level), la CPU hace una prueba de privilegio de acuerdo con la siguiente expresión: max(CPL,RPL)<=DPL. Si el acceso intentado pasa la prueba de privilegio, la dirección física es PA = segment\_base + offset, la cual debe estar dentro del límite del segmento. Un segmento de pila es un segmento de datos R|W en el cual el bit de tipo de segmento E está establecido para que se expanda hacia abajo y una dirección física PA en un segmento de pila debe ser más grande que el límite del segmento. Como en el modo real de 16 bits, en el sistema MTX no usaremos segmentos de pila debido a las restricciones de las variables apuntador en C.

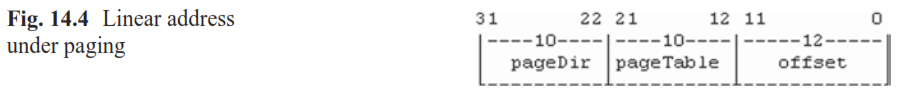
En general, un programa se ejecuta en segmentos de código con el mismo nivel de privilegio. Podría llamar procedimientos en un segmento de código conforme de nivel de privilegio más alto pero se ejecuta en el nivel de privilegio del programa original. Esta característica permite a los programas de modo usuario llamar a procedimientos en segmentos de código conformes de kernel. Un programa solamente puede transferir el control a un segmento de código no conforme de nivel de privilegio más alto a través de puertas en la Tabla de Descriptores de Interrupción (IDT, Interrupt Descriptor Table). Para MTX en modo protegido, supondremos dos niveles de privilegio y usaremos solamente segmentos de código no conformes, los cuales evitan que los programas de modo usuario ejecuten código kernel directamente.

14.2.2 Paginación

En el modo protegido, se puede también administrar la memoria por paginación. En la CPU x86, la paginación es implementada sobre la segmentación. Una dirección lógica primero es mapeada por segmentación a una dirección lineal. Si la paginación no está habilitada, la dirección lineal es también la dirección física. Si la paginación está habilitada, la dirección lineal es adicionalmente mapeada por el hardware de paginación de la CPU a una dirección física. En la mayoría de los sistemas de paginación el término paginación se refiere a paginación pura, el cual no tiene una noción de segmentos. La mayoría de los sistemas tipo Unix usan paginación pura. En la CPU x86 no hay forma de deshabilitar la segmentación, pero hay una forma de darle la vuelta. Cuando se está usando paginación, primero establecemos un modelo de segmento plano en el cual todos los segmentos son de tamaño 4GB, lo cual efectivamente esconde la capa de segmentación, haciéndola transparente. Entonces establecemos tablas de página y activamos la paginación. Con la paginación habilitada, una dirección lineal de 32 bits es tratada por la unidad de administración de memoria de la CPU (MMU, Memory Management Unit) como una

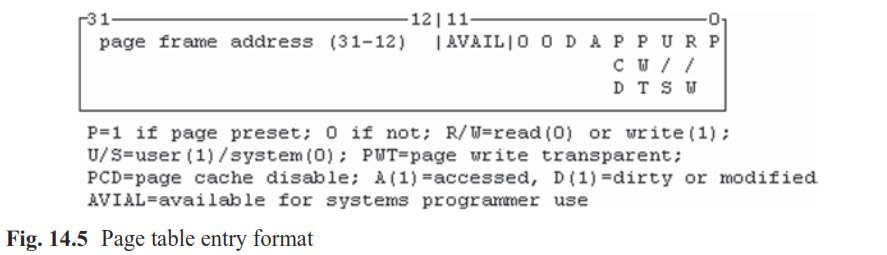
triada = [pageDir, pageTable, offset],

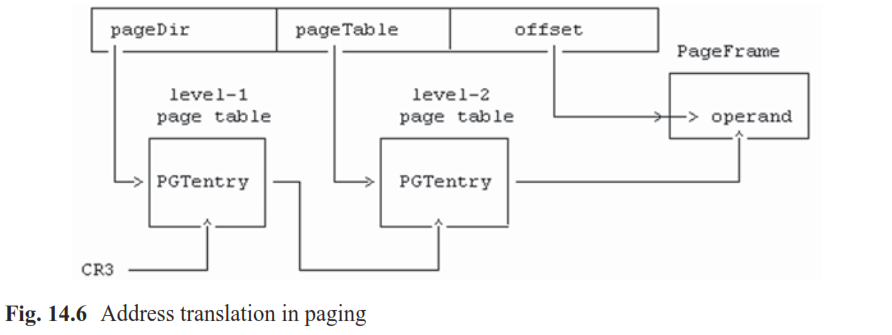
como se muestra en la Fig. 14.4.



14.2.2.1 Directorio de Página y Tablas de Página

En una dirección lineal, el pageDir se refiere a una entrada en una tabla de página de nivel 1, la pageTable se refiere a una entrada en una tabla de página de nivel 2 y el offset es la dirección relativa en la página. El tamaño de página normal es de 4KB. Con extensión de tamaño de página (PSE, Page Size Extension), algunas CPU x86 también soportan tamaño de superpágina de 4MB. Cuando se está usando paginación la MMU primero usa el registro de control CR3 para localizar el directorio de página, el cual es la tabla de página de nivel 1. Cada entrada de tabla de página tiene el formato mostrado en la Fig. 14.5.





14.2.2.2 Traducción de Dirección con Paginación

La Figura 14.6 ilustra el procedimiento de traducción de dirección con paginación. El registro CR3 de la CPU a la tabla de página de nivel 1. Dada una dirección lineal, la MMU primero usa el pageDir de 10 bits para localizar una entrada en la tabla de página de nivel 1. Suponga que la entrada de tabla de página de directorio está presente y que la prueba de acceso es OK. La MMU usa la dirección de frame de página de 20 bits de la entrada de tabla de página para localizar la tabla de página de nivel 2. Entonces usa el pageTable de 10 bits para localizar la entrada en la tabla de página de nivel 2. Suponga que la entrada de página está presente y que la prueba de acceso es OK también. La entrada de la tabla de página contiene la dirección del frame de página en memoria. La dirección física final es PA = (page frame address << 12) + offset. Dado que la paginación usualmente se apoya en el modelo de segmento plano, la protección a través de revisar los límites de segmento ya no tiene sentido. Con paginación, la protección es forzada por las entradas de tabla de página individuales. Una página está presente o no está presente. Un intento de acceder a una página no presente genera una page fault. Además, una entrada de tabla de página puede ser marcada ya sea como de solo lectura o escribible. Un intento de escribir a una página de solo lectura también genera una page fault. Los bits de acceso (A) y dirty (D) pueden ser usados para implementar remplazamiento de página en demand-paging.

14.2.2.3 Translation Lookaside Buffer (TLB)

Para aumentar la velocidad del proceso de traducción de paginación, la CPU almacena las entradas de tabla de página más recientemente usadas en una memoria cache interna, llamada la TLB. La mayor parte de la paginación se realiza usando los contenidos de la TLB. Se realizan ciclos de bus solamente cuando se usa una página nueva. Siempre que las tablas de página son cambiadas, el kernel del sistema operativo debe vaciar la TLB para evitar usar entradas de antiguas en la TLB. El vaciado de la TLB se puede hacer recargando el registro de control CR3. Entradas individuales en la TLB se pueden vaciar usando la instrucción INVPLG.

14.2.2.4 Full Paging

El esquema de paginación más simple es full paging. En este esquema, todas las páginas de una imagen de proceso son alojadas en frames de página física una vez. Después de cargar una imagen de proceso en frames de página, las páginas están siempre presentes. El esquema full paging puede ser estático o dinámico, dependiendo de cómo son alojados los frames de página. En paginación estática, cada imagen es alojada en una sola pieza de memoria física contigua alineada a frontera de página. El área de memoria es dividida en una secuencia de frames de página, las cuales son usadas como entradas de tabla de página. La principal ventaja de la paginación estática es que es extremadamente fácil de implementar. Primero, la imágenes de proceso pueden ser manejadas como particiones de tamaño variable por el algoritmo simple first fit. Segundo, es muy fácil construir tablas de página dado que todos los frames de página son contiguos. Tercero, no hay necesidad de mantener una estructura de datos separada para manejar los frames de página libres. En la paginación dinámica, las páginas de una imagen son alojadas dinámicamente. La principal ventaja de este esquema es que una imagen puede ser cargada en cualesquiera frames de páginas disponibles, los cuales no tienen que ser contiguos.

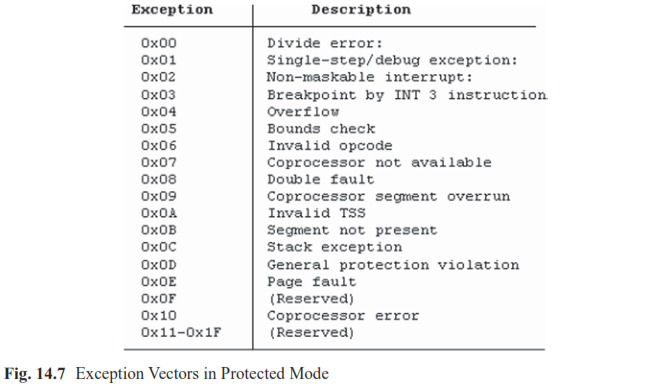
14.2.2.5 Demand-Paging

En demand-paging, las tablas de página de una imagen de proceso son construidas de acuerdo al tamaño de la imagen, pero no todas las páginas son frames de página alojados. Las páginas que no tienen frames de página sonmarcadas como no presentes. La dirección de frame en una entrada de tabla de página ausente podría apuntar a su ubicación en un dispositivo físico, por ejemplo, un número de bloque en un disco swap que contiene la imagen de página. Durante la ejecución, cuando un proceso intenta hacer referencia a una página que no está presente, éste genera una page fault, la cual cae en una trampa hacia el kernel del sistema operativo. El manejador de page fault del kernel del sistema operativo puede alojar un frame de página para la página, cargar la página faltante en el frame de página y cambiar la entrada de tabla de página a presente. Entonces el manejador le permite al proceso continuar con la entrada de tabla de página válida. Demand-paging es la base de la memoria virtual, en la cual el espacio de dirección virtual de un proceso puede ser mucho más grande que la memoria física alojada para este.

14.3 Procesamiento de Interrupciones y Excepciones

14.3.1 Excepciones en el modo protegido

El procesamiento de interrupción y excepción en el modo protegido difiere del que se hace en modo real en dos áreas. Primero, Los primeros 32 vectores de interrupción están reservados para excepciones, las cuales son listadas en la Fig. 14.7. Los vectores de excepción se traslapan con los vectores de interrupción tradicionales de IRQ0 a IRQ7 (0x08 a 0x0F). Los vectores de IRQ deben ser remapeados a diferentes ubicaciones. Segundo, los vectores de excepción ya no son áreas de memoria en el 1KB bajo como en el modo real. En lugar de ello, están definidos como descriptores en una tabla de descriptores de interrupción (IDT).



14.3.2 Tabla de Descriptores de Interrupción (IDT)

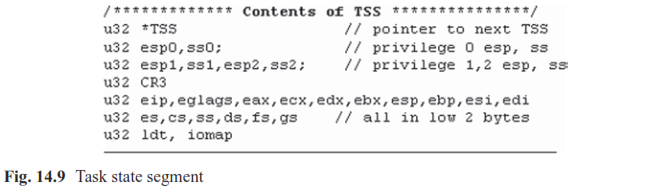
La IDT es una estructura de datos que contiene puertas de interrupción y de trampa. El registro IDTR de la CPU apunta a la IDT. Los contenidos de la IDT son esencialmente descriptores, pero Intel elige llamarlos puertas de interrupción o de trampa. La figura 14.8 muestra el formato de una puerta de interrupción o trampa.



Donde P es el bit de presente, pL es el nivel de privilegio, TYPE=1110 para puertas de interrupción y 1111 para puertas de trampa. La diferencia entre ellas es que invocar una puerta de interrupción automáticamente deshabilita las interrupciones, pero invocar una puerta de trampa no las deshabilita. Dado que las interrupciones y excepciones son procesadas en modo kernel, el nivel de privilegio debe ser establecido a 00. Éste puede ser establecido a 11 para permitir a los programas de modo usuario manejar interrupciones generadas por software. Por conveniencia, definimos los siguientes prototipos de puertas de interrupción y de trampa.



**Los campos de dirección de las puertas de IDT pueden ser establecidos para apuntar a los puntos de entrada de diferentes funciones manejadoras de excepción y de interrupción en el segmento de código del kernel**. Además de puertas de interrupción y de trampa, la IDT también puede contener puertas de call y puertas de task. El llamar a una puerta de tarea o de interrupción podría disparar un “switcheo” de tarea por hardware. Para MTX en modo protegido, no utilizaremos switcheo de tareas por hardware por las siguientes razones. Primero, el switcheo de tareas involucra mucho más que solo switchear el contexto de hardware de las tareas. Segundo, el switcheo de tareas por software es más flexible dado que está bajo el control directo del diseñador del sistema operativo. Por último pero no menos importante, el switcheo de tareas por hardware solamente está soportado en CPUs de 32 bits. Ya no está soportado en CPUs x86 de 64 bits. Para MTX en modo protegido, solo usamos TSS para definir las pilas de modo kernel de proceso. Para hacer esto, definimos un TSS en la estructura PROC, colocamos el descriptor de TSS en la GDT y hacemos que el registro de tarea de la CPU (TR) apunte al descriptor del TSS en la GDT. La figura 14.9 muestra la estructura de datos TSS.

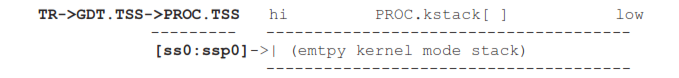


14.3.3 Task State Segment

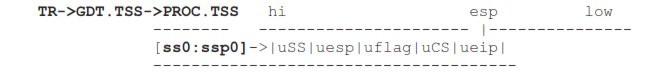
En la estructura TSS, los campos de eip a gs son registros de CPU guardados durante el switcheo de tarea por hardware. Dado que asumimos solo dos niveles de privilegio sin switcheo de tareas por hardware, los campos más importantes son esp0 y ss0, los cuales definen la pila de interrupción de la CPU. Cuando ocurre una interrupción, la CPU automáticamente usa [ss0:esp0] como la pila de interrupción. Por lo tanto, [ss0:esp0] debe apuntar a la pila de modo kernel del proceso corriendo actualmente. Las siguientes discusiones ayudarán a clarificar este importante punto.

14.3.4 TSS y la pila de interrupción de proceso

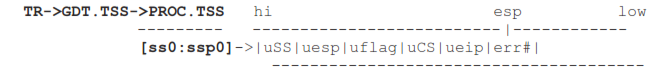
Suponga que la CPU está ejecutando un proceso en modo usuario. En este momento la pila de modo kernel del proceso está vacía. El registro TR de la CPU apunta a un selector de TSS en al GDT el cual apunta al TSS del proceso, en el cual [ss0:esp0] apunta al extremo alto de la pila de kernel del proceso como se muestra en el siguiente diagrama.



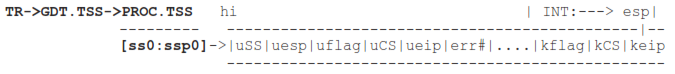
Cuando ocurre una interrupción, la CPU guarda uSS, uSP, uflags, uCS y uiep en la pila de interrupción la cual se convierte en



donde el prefijo u denota registros de modo usuario, y los uSS, uesp guardados son la pila de modo usuario en el punto de interrupción. Si ocurre una excepción en el modo usuario, la situación es exactamente la misma, excepto que para algunas excepciones la CPU también “pushea” un número de error err#, sobre la pila de interrupción, lo cual convierte a la pila en



Mientras que en modo kernel, si otra interrupción o excepción ocurre la CPU continúa usando la misma pila de interrupción para “pushear” una o más capas de contextos interrumpidos. Si la CPU ya está en modo kernel, reentrar al modo kernel ya no involucra cambio de privilegio. En este caso, el contexto guardado solamente tiene a |kflags|kCS,keip| de modo kernel, como se muestra en el siguiente diagrama.

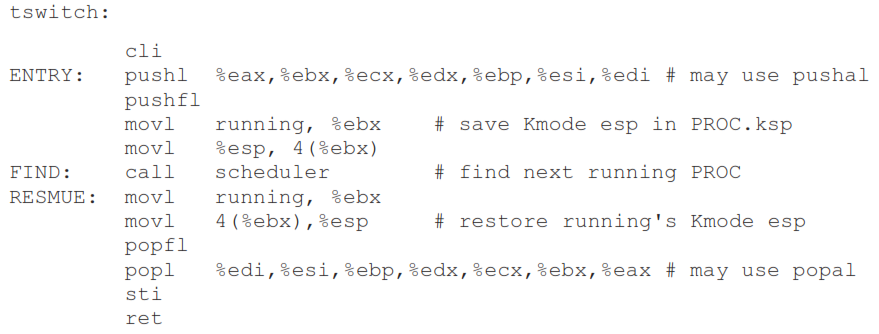


Cuando se regresa de un manejador de interrupción/excepción la operación iret soporta diferentes dtipos de acciones en general. Solamente consideramos el caso de dos niveles de privilegio sin switcheo de tareas por hardware. En este caso, iret checa los CPLs de los segmentos de código actual y siguiente. Si los CPLs son los mismos, i.e., de kernel de regreso a kernel, ésta solamente extrae de la pila los registros de modo kernel guardados [keip,kCS,kflags]. Si el CPL del siguiente segmento de código es más grande, i.e. si se está regresando del modo kernel al modo usuario, ésta extrae de la pila los registros de modo usuario guardados, los cuales incluyen el uesp y uSS guardados.

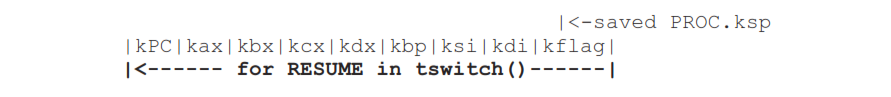
14.3.5 Pila de kernel de proceso en Modo Protegido

(1)Cambio de contexto: Como en el modo real, el cambio de contexto del MTX en modo protegido

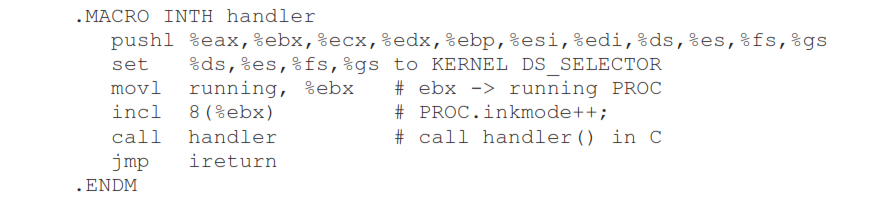
es a través de la función tswitch(), la cual se muestra abajo en un código de pseudo ensmblador de 32 bits.



Como en el modo real, no hay necesidad de guardar los registros de segmento del modo kernel en tswitch() dado que son siempre lo mismo. Correspondiendo a los códigos ENTRY y RESUME de tswitch(), el frame de pila de kernel para que un proceso retome su ejecución es de la forma



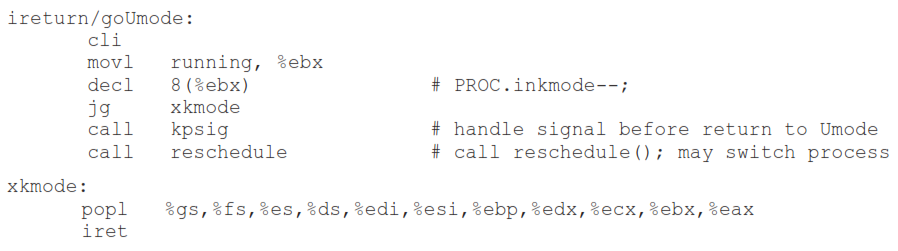
(2).Los códigos de entrada y salida de los manejadores de interrupción: Como en el modo real, los puntos de entrada de los manejadores de interrupción, incluyendo llamadas al sistema (system call), también son definidas por una macro INTH, la cual se muestra abajo en pseudocódigo.



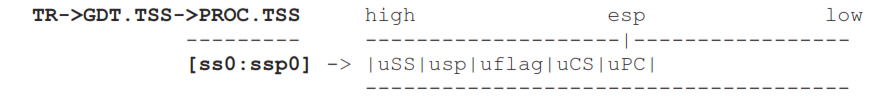
Los puntos de entrada de los manejadores de interrupción son instalados como llamadas a la macro INTH, por ejemplo



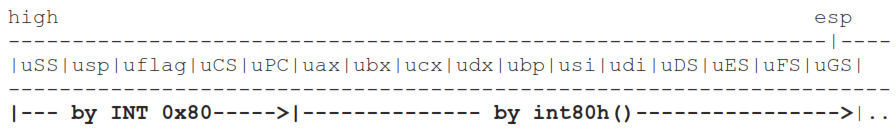
Cada manejador de interrupción llama a una función manejadora en C, la cual realmente maneja la interrupción. Cuando la función manejadora en C termina, ésta lanza una instrucción EOI y regresa a ejecutar el código ireturn/goUmode. Si la interrupción ocurrió en modo kernel, el proceso restaura los registros de la CPU guardados y hace IRET de regreso al punto de interrupción directamente. En caso contrario, maneja la señal primero y podría switchear de proceso antes de regresar al modo usuario. Lo siguiente muestra el pseudocódigo de ireturn/goUmode.



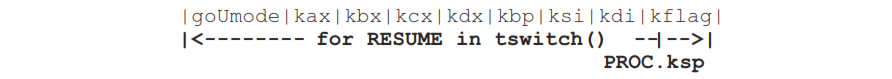
Como en el modo real, usamos un contador PROC.inkmode para mantener registro del número de veces que un proceso entra al modo kernel. En el modo real, cuando un proceso hace una llamada al sistema, la CPU guarda [uflag, uCS, uPC] en la pila de modo usuario del proceso. El código de entrada del manejador de interrupción, int80h(), guarda los otros registros, también en la pila de modo usuario. Cuando el manejador de llamada al sistema termina, el proceso regresa al modo usuario a través del frame de pila de interrupción en la pila de modo usuario. La situación cambia solo ligeramente en el modo protegido. En lugar de la pila de modo usuario, todos los frames de pila están en la pila de modo kernel del proceso. En el modo protegido, implementaremos syscall a través de INT 0x80, como se hace en Linux y en xv6. Cuando un proceso hace una syscall, la CPU guarda [uSS, usp, uflag, uCS, uPC] en la pila de kernel del proceso, la cual se convierte en



El código de entrada del manejador de syscall, int80h(), guarda los otros registros de la CPU en la pila de kernel de proceso, la cual se convierte en



Mientras que en modo kernel, si el prceso cede la CPU llamando a tswitch(), éste agrega un marco de pila de retorno en la cima de la pila de kernel de proceso. Por lo tanto, para permitir a un proceso regresar al modo usuario inmediatamente cuando este retoma su ejecución, podemos ajustar su pila de kernel para contener dos frames; un frame de pila de interrupción como el mostrado arriba, seguido por un frame de pila de retomar ejecución (resume stack frame), como se muestra abajo.



Esta técnica será usada en todas las versiones de MTX en modo protegido.

14.4 GCC en Modo de 32 bits

Para MTX en modo protegido de 32 bits, usaremos el ensamblador, compilador, enlazador de GCC para generar código de 32 bits. En Linux de 32 bits, GCC genera código de 32 bits por default. En Linux de 64 bits, debemos usar gcc –m32 para generar código de 32 bits.

14.4.1 Ensamblador Inline

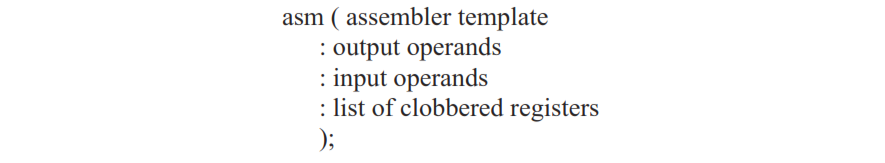
El compilador de C de GCC soporta ensamblador inline, lo cual se usa frecuentemente en código de kernel de SO por conveniencia. El formato básico de ensamblador inline es



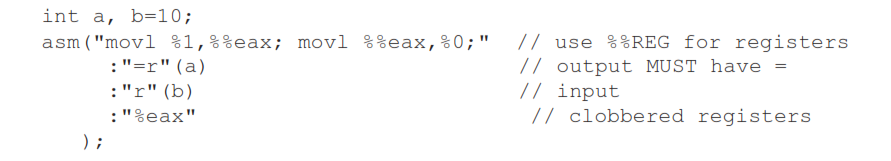
Si el código de ensamblador tiene más de una línea, las sentencias pueden ser separadas por \n\t; como en



En el código de ensamblador inline también se puede especificar operados. La plantilla de tal código de ensamblador inline es



Las sentencias de ensamblador pueden especificar operandos de salida y de entrada, los cuales son referenciados como %0, %1. Por ejemplo, en el siguiente segmento de código,



%0 se refiere a la variable a y %1 se refiere a la variable b, %%eax se refiere al registro eax. El operador de restricción “r” significa usar un registro para el operando. El código de arriba es equivalente a



Esto también le dice al compilador de GCC que el registro eax será contaminado (clobbered) por el código inline. Aunque se podría insertar código ensamblador inline más o menos complejo en un programa en C, abusar de ello podría comprometer la legibilidad del programa. En la práctica, e ensamblador inline debe ser usado solamente si el código es muy corto, por ejemplo, una sola instrucción de ensamblador o si la operación involucra un registro de control de la CPU. En tales casos, el código ensamblador inline no solo es claro sino también más eficiente que llamar a una función de ensamblador.

14.4.2 Constantes, Atributos Volatile y Packed

En un programa en C, las variables o parámetros de función que deban ser de solo lectura pueden ser declaradas con un calificador const para evitar que sean modificadas accidentalmente. Si el código del programa intenta modificar un elemento const, esto será detectado como un error en tiempo de compilación. Para constantes estáticas, tales como las cadenas en char \*p=”abcd”; printf(“%d %s\n”,….); etc. el compilador de C usualmente las pone en una sección de datos de solo lectura. Durante la ejecución, tratar de escribir en memoria de solo lectura generará una segmentation fault. El GCC es un compilador optimizante. Cuando se está escribiendo drivers de dispositivos, manejadores de interrupción, o código multihilo en C, frecuentemente es necesario declarar variables como volatile, lo cual evita que el compilador de GCC haga optimización de código que involucre tales variables. De la misma forma, el atributo packed de una estructura de C le dice al compilador de GCC que no agregue pad a la estructura con bytes extra para lograr alineamiento de memoria.

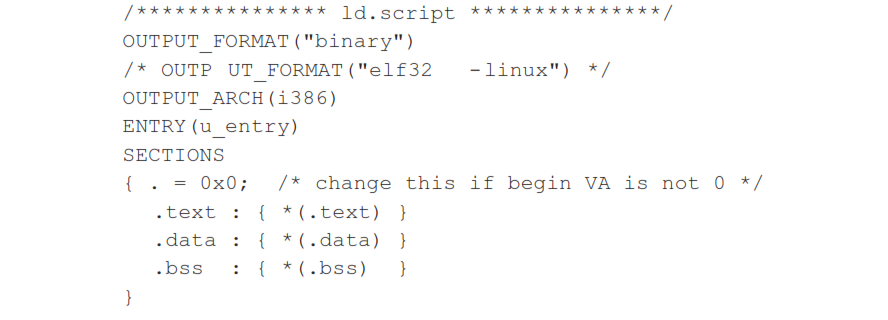
14.4.3 GCC Linker and Executable File Format

El linker de GCC genera archivos ejecutables en varios formatos diferentes. En Linux, el formato de archivo ejecutable por defecto es ELF (Tool 1995,Youngdale 1995), el cual está mejor acondicionado para enlace dinámico. Dado que MTX aún no soporta enlace dinámico, usaremos el formato ELF solamente para archivos enlazados estáticamente.

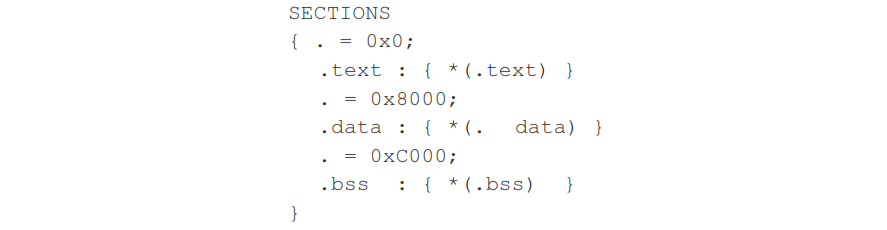
(1).Linker Script: Un linker script es un archivo de texto el cual especifica opciones de enlazado para el linker. Su uso es



Como un ejemplo, lo siguiente muestra un linker script simple, el cual podría ser usado para generar ya sea un viario plano o un archivo ELF ejecutable.

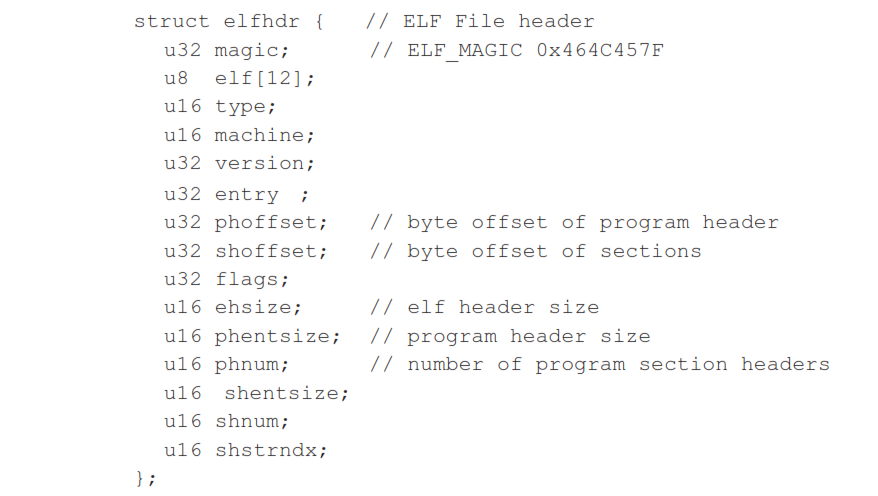


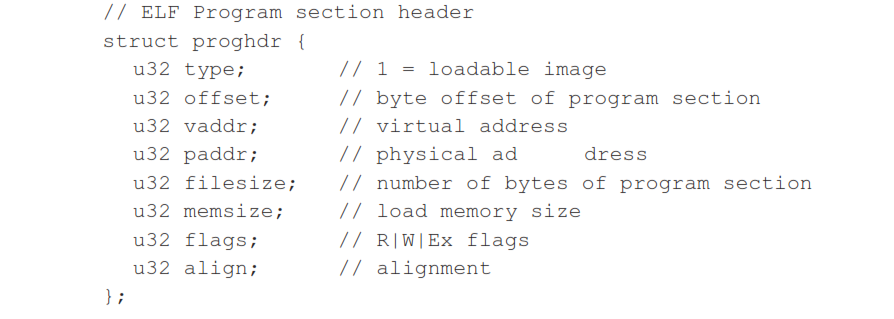
Un archivo ejecutable binario plano es una sola pieza de código binario, el cual no tiene secciones de código y de datos separadas. Cambiando el formato de archivo de salida a “elf32-linux”, el linker generará un archivo ejecutable ELF. Para este ejemplo, el archivo ELF resultante tiene solamente una sección de programa (cargable (loadable)). Si se desea, el linker script puede ser modificado para generar un ejecutable ELF con secciones de código, datos, y bss separadas, como en



El archivo ELF resultante tendrá secciones code, data y bss separadas cada una con una dirección virtual de inicio diferente.

(2).Formato de archivo ejecutable ELF: Un archivo ejecutable ELF comienza con una cabecera elf, seguida por una o más cabeceras de sección de programa, los cuales están definidos como

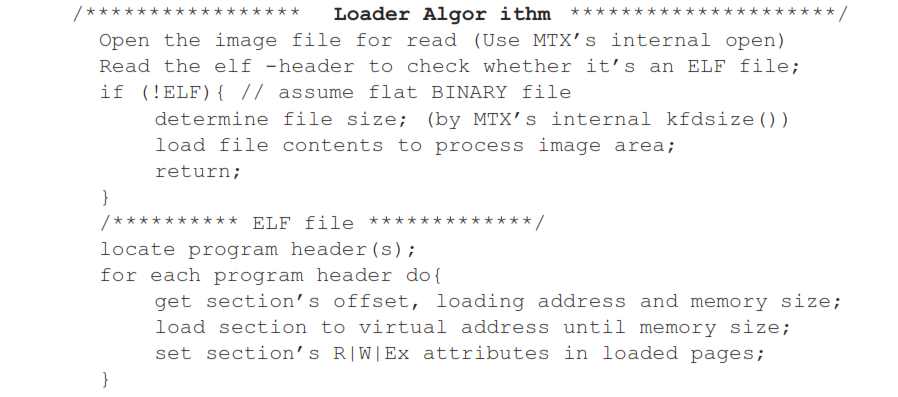




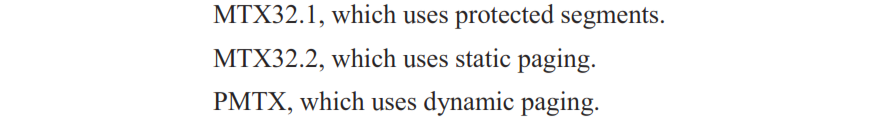
(3).Cargador de archivo ejecutable ELF: Cuando se está cargando un archivo ELF el cargador debe cargar las varias secciones de un archivo ELF a sus direcciones virtuales especificadas. Además, cada sección cargada debe ser marcada con atributos apropiados R|W|Ex para protección.

Por ejemplo, las páginas de la sección de código deben ser marcadas como REx, las páginas de la sección de datos deben ser marcadas como RW, etc.

(4).Program Loader in Protected Mode: El programa cargador de MTX en modo protegido puede cargar archivos ya sea binario plano o imagen ELF. El algoritmo del cargador es como sigue.



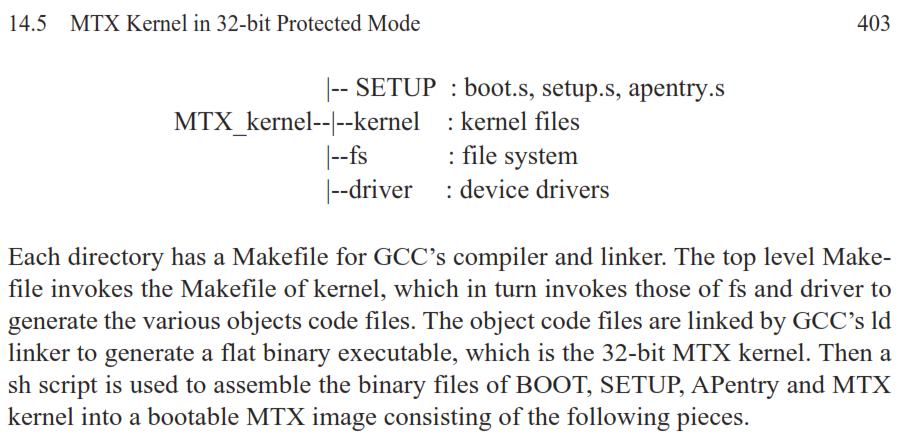
En resumen, para operaciones de modo protegido debemos **ajustar la GDT, remapear los vectores IRQ, instalar manejadores de interrupción y excepción en la IDT y definir la(s) pila(s) de interrupción de CPU a través de un TSS en la GDT**. Como en el modo real, conocer el contenido de la pila de modo kernel es esencial para entender como establecer un proceso para ejecución en modo protegido. En las secciones siguientes (14.5, …, 14.?) describiremos el diseño e implementación de MTX en el modo protegido de 32 bits. Para ilustrar las diferentes capacidades del hardware de administración de memoria de la PC, presentaremos tres versiones diferentes de MTX ene l modo protegido de 32 bits, cada una usa un esquema de administración de memoria distinto. Las tres versiones de MTX en el modo protegido de 32 bits son

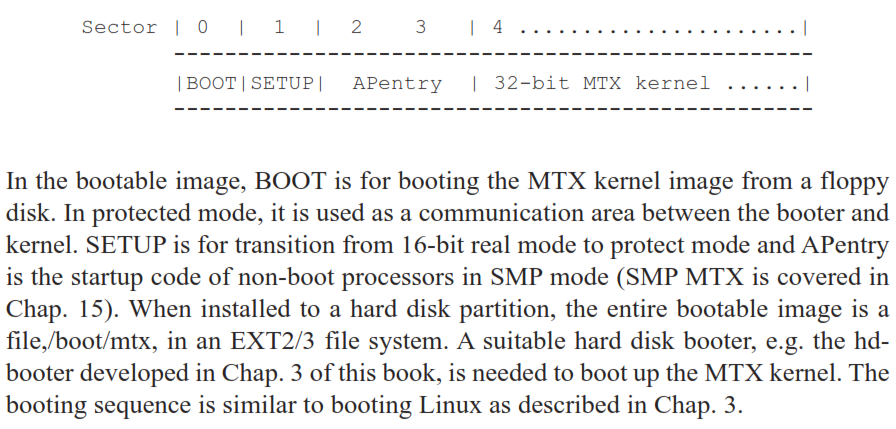


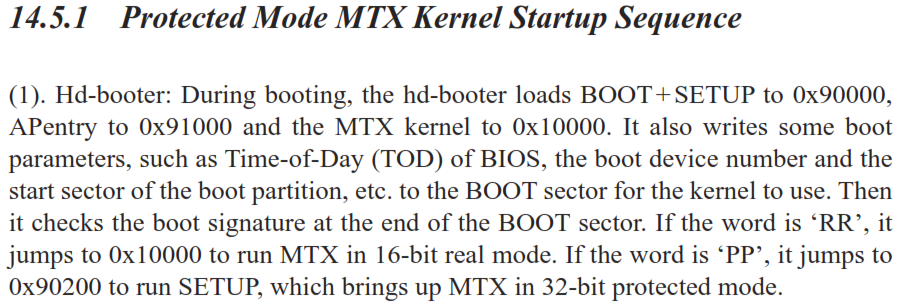
De estos, MTX32.1 y MTX32.2 son intentados solamente para demostración. PMTX es la versión final de MTX en el modo protegido de 32 bits para sistemas uniprocesador.

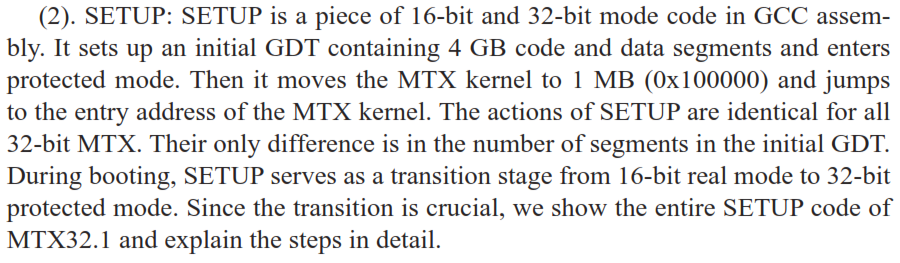
Pag 402, [Wang, K.C.]











PAG. 404 de [Wang, K. C.]

14.<algo>

REF.

Wang., K.C. Design and Implementation of the MTX Operating System. Editorial Springer Nature, 2015. Capítulo 14.

K. C. Wang (auth.) - Design and Implementation of the MTX Operating System-Springer International Publishing (2015).pdf