Capítulo 15 (p449, 461/551)

Symmetric Multiprocessing MTX

15.1 Sistemas Multiprocesador

Un sistema multiprocesador consiste de un número múltiple de procesadores, los cuales comparten la memoria principal y los dispositivos de I/O. Si la memoria principal compartida es la única memoria en el sistema, el sistema es llamado un sistema Uniform Memory Access (UMA). Si, además de la memoria compartida, cada procesador tiene una memoria local privada, el sistema es llamado sistema Non-uniform Memory Access (NUMA). Si los roles de los procesadores no son los mismos, por ejemplo, solamente algunos de los procesadores pueden ejecutar código de kernel y otros no pueden, el sistema es llamado un sistema Asymetric MP (ASMP). Si todos los procesadores son funcionalmente idénticos, el sistema es llamado sistema Symmetric MP (SMP). Con la tecnología de procesador multicore actual, SMP se ha convertido virtualmente en sinónimo de MP.

15.2 Sistemas conformes con SMP

Un sistema SMP requiere mucho más que solo un número múltiple de procesadores o cores de procesador. Para soportar SMP, la arquitectura del sistema debe tener capacidades adicionales. La especificación de Multiprocesador de Intel (Intel Multiprocessor Specification 1997) define sistemas conformes con SMP como sistemas PC/AT compatibles con las siguientes capacidades.

1.Soportar enrutamiento de interrupciones e interrupciones inter-procesador. En un sistema conforme SMP, las interrupciones de dispositivos de I/O pueden ser enrutadas a diferentes procesadores para balancear la carga de procesamiento de interrupción. Los procesadores pueden interrumpirse uno a otro por Inter-Processor Interrupts (IPIs) para comunicación y sincronización. En un sistema conforme con SMP, estas son proporcionadas por un conjunto de Advanced Programmable Interrupt Controllers (APICs). Un sistema conforme con SMP usualmente tiene un system wide IOAPIC y un conjunto de local APICs de los procesadores individuales. Juntos, los APICs implementan un protocolo de comunicación inter-procesador, el cual soporta enrutamiento de interrupciones e IPIs.

2.Un BIOS extendido, el cual detecta la configuración del sistema y construye estructuras de datos SMP para uso del SO.

3.Cuando el sistema empieza, uno de los procesadores es designado como el procesador de booteo (BSP), el cual ejecuta el código de booteo para inicializar el sistema. Todos los otros procesadores son llamados Application Processors (APs), los cuales son amtenidos en el estado ocioso (idle) inicialmente pero pueden recibir IPIs desde el BSP para iniciar. Después del booteo, todos los procesadores son funcionalmente idénticos.

15.3 Secuencia de inicio de un sistema SMP (p450, 462/551)

15.3.1 Estructuras de datos SMP (p450, 462/551)

Cuando un sistema SMP inicia, el BIOS detecta la configuración de hardware del sistema y crea un conjunto de estructuras de datos SMP para uso del sistema operativo. Las estructuras de datos incluyen una Floating Pointer Structure (FPS),

(p451, 463/551)

Durante el booteo de un sistema conforme SMP, el BSP debe inicializar el hardware del sistema para operaciones SMP. Esto incluye

Configurar IOAPIC para enrutar interrupciones a local APICs.

Configurar y habilitar el local APIC del BSP.

Enviar INIT y STARTUP IPIs para activar otros APs.

Continuar a inicializar el kernel del SO hasta que esté listo para correr tareas.

Las siguientes secciones describen estas acciones en detalle.

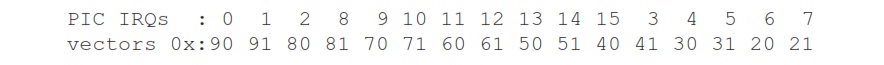
15.3.2 Configurar IOAPIC (p451, 463/551)

1 Establecer los registros de interrupción IOAPIC

Un sistema SMP usualmente tiene solamente un IOAPIC. El IOAPIC tiene 24 registros (de 64 bits), los cuales especifican como enrutar interrupciones y mapear IRQs a vectores de interrupción. Los registros son accedidos indirectamente a través de un par de registros IOREGSEL e IOWIN, los cuales están localizados en 0xFEC00000 y 0xFEC00010, respectivamente. Otros registros son offsets de byte desde el registro IOWIN. Todos los registros IOAPIC deben ser accedidos por lecturas/escrituras de 32-bit. Para acceder a un registro IOAPIC, primero se selecciona el registro escribiendo su byte offset a IOREGSEL, entonces se lee/escribe un valor de 32-bit desde el/al registro IOWIN. Para cada registro de interrupción IOAPIC, toma dos operaciones leer/escribir las dos mitades de 32-bit del registro. Algunas asignaciones estándar de los registros de interrupción IOAPIC son



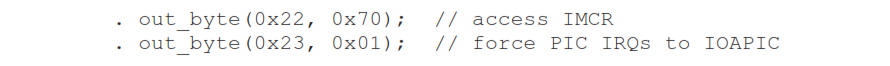
Para las interrupciones IRQ 0-15, deben ser establecidas a edge-triggered y active high. Para interrupciones PCI A – D, deben ser establecidas a level-triggered y active low. En el modo protegido, los primeros 32 vectores de interrupción están reservados. Cada uno de los registros de interrupción IOAPIC puede ser programado con alguno de los restantes 224 vectores. Los cuatro bits más altos de un vector de interrupción contienen la prioridad de interrupción. Los números de vector de interrupción más altos tienen prioridades más altas. Las prioridades de interrupción PIC estándar son IRQ 0-2, 8-15, 3-7, donde IRQ0 tiene la prioridad más alta. Si es necesario, las IRQs pueden ser mapeadas por los registros de interrupción IOAPIC a vectores que preserven sus prioridades. Hay 16 prioridades de interrupción diferentes, 0 a F. Dado que no podemos usar los vectores 0x00-0x1F, esto deja solamente 14 prioridades de interrupción disponibles. Entonces, algunas de las IRQs deben ser mapeadas a vectores con la misma prioridad. Por ejemplo, si remapeamos IRQ0-15 a 0x21-0x2F y programamos los registros IOAPIC 0-15 para los vectores 0x21-0x2F, entonces todas las interrupciones son de la misma prioridad (2). Alternativamente, podríamos elegir vectores 0x20-0x9F para las IRQs de PIC y asignar dos IRQs a cada nivel de prioridad, resultando en la siguiente asignación.



Similarmente para los registros de interrupción 16-19, los cuales mapean interrupciones PCI A—D. Además de los vectores de interrupción, cada registro de interrupción IOAPIC también debe ser programado con modo de envío de interrupción y destino. El modo de envío puede ser físico o lógico. La forma más simple es usar modo de envío lógico y enrutar interrupciones a APICs con la prioridad más baja. La prioridad de un APIC está en el registro de prioridad de tarea de APIC. Alternativamente, podríamos también usar el modo de envío físico y programar los registros IOAPIC para enrutar interrupciones a procesadores específicos para balancear la carga de procesamiento de interrupción.

2 Switchear a modo I/O simétrico

Para enrutar interrupciones a local APICs, el sistema debe switchear a modo I/O simétrico. Para PCs con un Interrupt Mask Control Register (IMCR), esto se puede hacer escribiendo 0x01 al registro de datos de IMCR. La mayoría de las máquinas virtuales no tienen un IMCR, así que este paso es opcional.



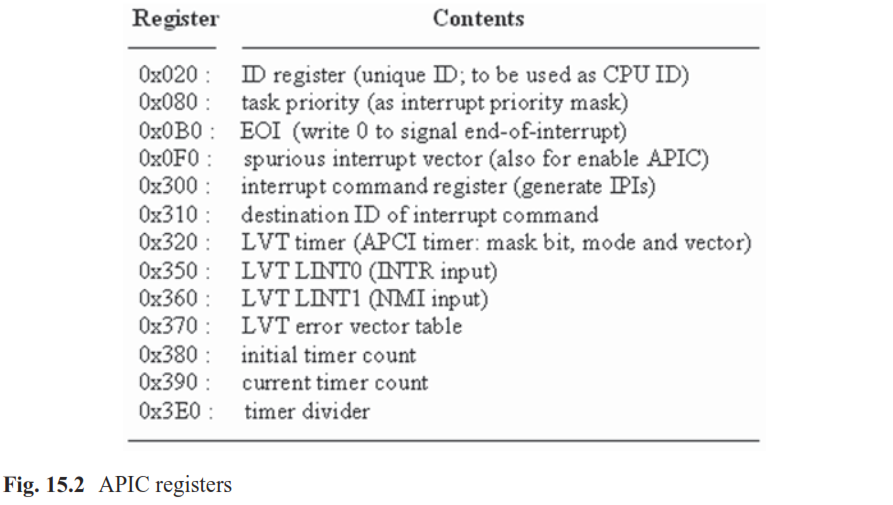
3. Deshabilitar los PICs 8259

Después de ajustar el IOAPIC para enrutar interrupciones, los PICs 8259 deben ser deshabilitados para prevenir múltiples interrupciones de la misma fuente.



15.3.3 Configurar Local APICs (p453, 465/551)

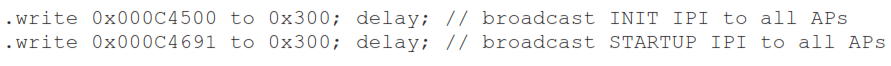
Cada procesador tiene un local APIC en la misma dirección base 0xFEE00000. Los registros de local APIC son offsets desde la dirección base APIC. Los registros APIC pueden ser accedidos directamente con lectura/escritura de 32-bit. Algunos de los registros APIC y sus usos son listados en la Fig. 15.2



Para habilitar el local APIC, escriba 0x010F al registro de interrupción espuria (0x0F0). Esto también establece 0x0F como el vector por defecto para interrupciones espurias.

15.3.4 Enviar IPIs para activar APs (p453, 465/551)

Después de habilitar su local APIC, el BSP debe activar otros APs enviándoles INIT y STARTUP IPIs. Esto se puede hacer escribiendo a los registros de comando de interrupción (Interrupt Command Register) (0x310 = dest, 0x300 = IPI) para enviar IPIs a cada AP individual o transmitirlas a todos los APs. En el último caso, los IPIs son lanzados como sigue, donde el retardo es de aproximadamente 200 msec.



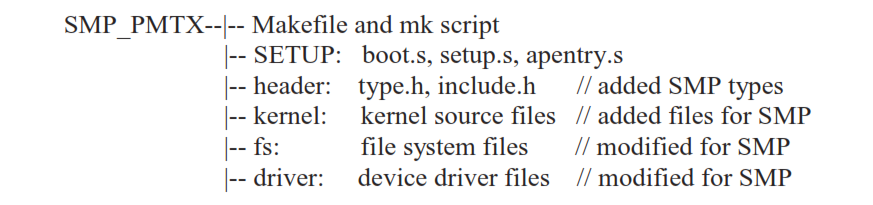
15.3.5 Código Trampolín de APs (p454, 466/551)

Cada AP despierta para ejecutar una pieza de código trampolín en el modo real de 16-bits. El código trampolín debe comenzar en una frontera de página de 4 KB en memoria de modo real. La ubicación es determinada por el número de vector en el STARTUP IPI. En el ejemplo de arriba, el valor del vector es 0x91, el cual le dice a los APs que comiencen su ejecución desde la memoria 0x91000 en modo real. Cada AP debe switchear al modo protegido, establecer tablas de páginas y configurar local APIC antes de entrar al kernel del SO (before entering the OS kernel). En SMP, un procesador podría usar IPIs para sincronizarse con otros procesadores, tal como hacer flush sus TLBs e invalidar entradas de tablas de páginas, etc.

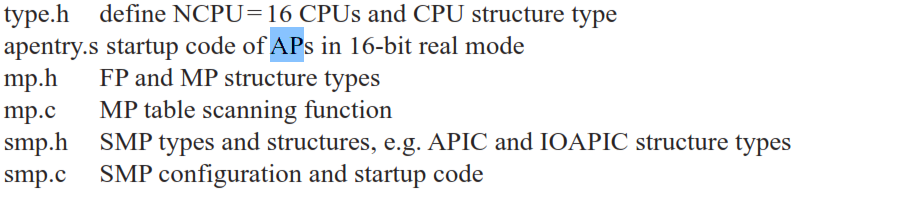
15.4 De UP a SMP

15.5.7 Organización del kernel SMP\_PMTX

Siguiendo los principios de arriba, hemos adaptado el kernel UP de PMTX para SMP. El kernel SMP\_PMTX consiste del siguiente árbol de archivos, el cual es el mismo de PMTX pero con características agregadas para soportar SMP.

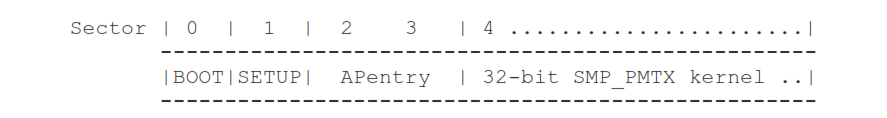


Las principales características en el soporte de SMP están en los siguientes archivos.



15.5.8 Imagen SMP\_PMTX booteable

Una imagen de SMP\_PMTX booteable consiste de las siguientes piezas.



Durante el booteo, el booter carga BOOT+SETUP en el block 0 a 0x90000, APentry en el bloque 1 a 0x91000 y el kernel MTX a 0x10000. Para kernels MTX de modo protegido, la última palabra en el sector BOOT contienen la firma boot ‘PP’. Cuando el booter detecta esto, salta a 0x90200 para correr SETUP, el cual lleva el sistema al modo protegido de 32-bits. Cuando el kernel SMP PMTX empieza, el procesador de booteo (BSP) primero llama a findmp() para buscar el FPS y la tabla MP. Escanea la tabla MP y devuelve el número de CPUs en el sistema. Actualmente, el kernel SMP\_PMTX soporta hasta 16 CPUs. Si se desea, fácilmente puede ser extendido a más CPUs. Si el sistema tiene solamente una CPU, el kernel cae al modo UP. En caso contrario, llama a smp() para configurar el sistema para operaciones SMP y levantar el sistema en el modo SMP. Después de inicalizar el kernel, el BSP transmite INIT y STARTUP IPIs a los APs con el vector 0x91, lo cual corresponde a la dirección de carga de APentry. Cada AP comienza ejecutando APentry en el modo real de 16-bits.

15.7 SMP y Sistemas Operativos en Tiempo Real

Un sistema operativo de tiempo real (Dietrich y Walker, 2015) es un SO que se construye para aplicaciones de tiempo real. Las aplicaciones de tiempo real usualmente tienen dos requerimientos de tiempo muy estrictos: tiempo de respuesta rápido y tiempo de terminación garantizado. Específicamente, un SO de tiempo real debe ser capaz de

* Responder a peticiones de interrupción de eventos de tiempo real dentro de un tiempo límite muy corto.
* Completar un servicio pedido dentro de un cierto tiempo límite, conocido como deadline de tarea.

Si un sistema siempre puede cumplir estos requerimientos críticos, entonces el sistema es llamado **sistema de tiempo real estricto**. Si un sistema solo puede cumplir estos requerimientos la mayor parte del tiempo, pero no siempre, entonces el sistema es llamado **sistema de tiempo real suave**. Para poder cumplir con esos requerimientos de tiempo críticos, un sistema de operativo de tiempo real usualmente es diseñado con las siguientes capacidades.

* Latencia de respuesta de interrupción y tiempo de cambio de tarea (task switch) mínimos: un kernel de SO de tiempo real no debe enmascarar interrupciones por periodos de tiempo largos. El código usado para implementar regiones críticas y cambio de tatreas (task switching) debe ser muy corto. Todas las regiones críticas deben ser tan cortas como sea posible.
* Un scheduler de tareas avanzado: el scheduler debe soportar planificación de tareas expulsiva con un algoritmo de planificación adecuado basado en deadline, tal como el algoritmo Earliest Deadline First (EDF) (Liu y Layland, 1973). La planificación (o scheduling) expulsiva permite que tareas de prioridad más alta expulsen tareas de prioridad más baja en cualquier momento. Este tipo de planificación es una condición necesaria, aunque no suficiente, para operaciones de tiempo real. Sin planificación expulsiva sería imposible cumplir con los deadlines de tareas.
* Para asegurar respuesta rápida, las tareas en un SO de tiempo real usualmente no tienen un modo de usuario separado. Todas las tareas corren en el mismo espacio de direcciones del kernel del SO.

Debido a sus requerimientos y objetivos únicos, el diseño e implementación de sistemas operativos de tiempo real difiere del de sistemas operativos de propósito general. Sin embargo, en ambos casos, también existen muchas cosas en común, especialmente en el área de sincronización de procesos, tales como, regiones críticas, protección de estructuras de datos para soportar ejecuciones concurrentes, prevención de deadlocks y de condiciones de carrera, etc. Por lo tanto, algunos de los principios de diseño y técnicas de implementación de los kernels SMP también son aplicables a los kernel de SO de tiempo real.

15.7.1 SMP\_MTX para procesamiento de tiempo real

El kernel SMP\_MTX soporta planificación de tareas expulsiva. Podríamos adaptar el kernel SMP\_MTX para operaciones de tiempo real como sigue. Divida los procesos en dos clases; de tiempo real y ordinarios. Asigne una prioridad alta fija, por ejemplo 256, para los procesos de tiempo real, y prioridades más bajas para los procesos ordinarios. Modifique el scheduler de tareas para correr primero los procesos de tiempo real con una rebanada de tiempo corta con un esquema round-robin hasta que no haya procesos de tiempo real ejecutables (READY). Entonces corra los procesos ordinarios, los cuales son planificados por prioridades de proceso dinámicas. Siempre que un proceso de tiempo real cambie su estado a READY, este debe expulsar a cualquier proceso ordinario. Dado que el sistema solamente garantiza inicio rápido y compartición equitativa de tiempo de CPU para los procesos de tiempo real pero no el cumplimiento de sus deadlines de terminación, se trata por lo tanto, de un SO de tiempo real suave. Este es el mismo enfoque usado en (Dietrich y Walker) para procesamiento de tiempo real en Linux.

15.8 SMP en el Modo Real de 16 bits

La especificación MP de Intel define los sistemas conformes a SMP como PCs con procesadores x486 y superiores, lo cual parece implicar que SMP está pensado solamente para el modo protegido. Sin embargo, una PC en modo real también tiene las siguientes capacidades.

* La CPU x86 en modo real de 16 bits soporta instrucciones atómicas, tales como xchg y locked inc/dec, etc. lo cual es esencial para SMP.
* Si es necesario, IRQ0 a IRQ15 pueden ser remapeadas en modo real.
* En SMP, todas las CPUs comienzan en modo real. Debe ser fácil integrar los APs en un kernel de modo real.
* Usando las instrucciones del BIOS INT 15-87, una PC en modo real puede leer/escribir los registros de los IOAPIC y APIC en el rango de direcciones arriba de 0xFEC00000.

Basándose en lo anterior, hemos tratado de extender el sistema RMTX de modo real a SMP. El resultado es un éxito calificado. En esta sección, describimos la adaptación de RMTX a SMP\_RMTX y discutimos algunos de los problemas y posibles remedios.

15.8.1 Secuencia de inicio de SMP\_RMTX en modo real

1. Durante el booteo, la CPU0 es el procesador de booteo (BSP), el cual ejecuta el código del booter. Como en RMTX, SMP\_RMTX comienza su ejecución desde el código de ensamblador ts.s, el cual llama a main() en tc.c. Mientras la ejecuión está en main(), primero se llama a init() para inicializar el kernel, crear y correr el proceso inicial P0, el cual llama a kfork(“/bin/init”) para crear el proceso INIT P1 y colocarlo en la readyQueue de la CPU0.

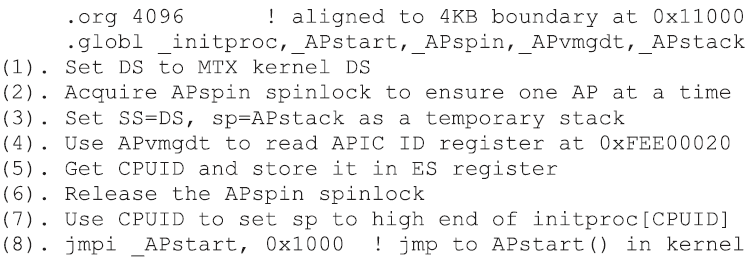
2. P0 llama a findmp() para obtener el número de CPUs en el sistema. Escanea la tabla MP y devuelve el número de CPUs en el sistema. Si el número de CPUs es 1, la ejecución regresa al modo UP. En caso contrario, se llama a smp() para configurar la PC para operaciones SMP.

3. Leer/Escribir registros IOAPIC y APIC: En modo real, usamos BIOS INT 15-87 para leer/escribir memoria alta. El kernel MTX es cargado en 0x10000. Usamos la palabra larga (long word) en 0x0F000 como un área de datos intermedia para leer/escribir registros APIC por las funciones

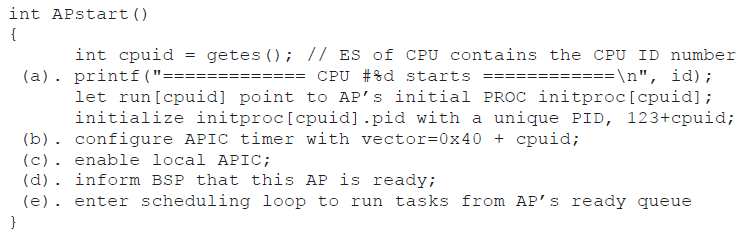


La función apic\_read() establece una GDT (Global Descriptor Table) con dirección fuente = 0xFEE00000 + apic\_reg y dirección destino = 0x0000F000. Ejecuta una instrucción INT 15-87 para leer el registro APIC de 32 bits a 0x0F000 como el valor de retorno. Similarmente para apic\_write(), la cual escribe una palabra larga (long word) a 0x0F000 y entonces ejecuta una instrucción INT 15-87 para escribirla a un registro APIC. Cambiando la dirección de memoria alta a 0xFEC00000 y 0xFEC00010, esas funciones también pueden leer/escribir registros IOAPIC.

4. Habilitar los local APIC de BSP y comenzar los APs: Durante el booteo, el procesador de booteo es establecido por el BIOS para recibir todas las interrupciones de los PICs 8259 (Programmable Interrupt Controller). Para SMP en modo real, nosotros no remapeamos las IRQs y configuramos el IOAPIC para enrutar las interrupciones. En lugar de eso, permitimos que el BSP (procesador de booteo) maneje todas las interrupciones de PIC como en el modo real. Esto ayuda a reducir los cambios necesarios en el kernel RMTX. Cuando el BSP arranca (boots up) habilitamos su local APIC y transmitimos (broadcast) INIT y STARTUP IPIs a otros APs. Cuando un AP comienza, empieza a ejecutar el código trampolín, el cual debe estar en una frontera de página de 4KB en la memoria de modo real correspondiente al número de vector (0x11) en el STARTUP IPI. Para cumplir con esto, agregamos el siguiente segmento de código al archivo de ensamblador ts.s. Dado que el kernel SMP\_RMTX es cargado en 0x10000, el segmento de código agregado está en la página alineada a 4KB 0x11000. En lugar del código ensamblador real, solamente mostramos el algoritmo de APentry.



Previo a activar los APs, el BSP (BootStrap Processor) inicializa el APvmgdt global para leer el registro ID de APIC en 0xFEE00020 por BIOS 15-87. Cuando un AP comienza, primero establece DS al segmento de datos del kernel SMP\_RMTX. Entonces lee el registro ID de APIC para obtener su CPUID y lo almacena en el registro ES. El AP usa el kstack de initproc[CPUID] para introducir APstart() en el kernel RMTX. El algoritmo de APstart() es



5. Timer de APIC local de APs: En el sistema SMP de modo real, necesitamos el timer del AP por la siguiente razón. Dado que las CPU0 recibe y maneja todas las interrupciones del PIC, los APs no tienen interrupciones. Para correr tareas, cada AP debe ser capaz de examinar su cola de procesos ready para hacer planificación de tareas. Hay varias formas posibles para hacer esto, tales como

- Por IPI: la CPU0 podría lanzar IPIs para informar a otros APs para comenzar una acción.

- Por memoria compartida: los APs no pueden monitorear algunos contenidos de memoria que son cambiados por la CPU0, sino que esto requiere usar polling por cada AP.

- Por un timer local, el cual interrumpe periódicamente, permitiendo a cada AP buscar trabajo por sí mismo.

Entre estas, un timer local es el más simple de implementar. Si una CPU no encuentra trabajo que hacer, corre un proceso idle con interrupciones habilitadas. Una interrupción de timer causará que la CPU se levante para manejar la interrupción y entonces tratar de correr una tarea otra vez.

6. Después de ajustar el timer local, cada AP incrementa una variable global go\_smp en 1 para informar al BSP (BootStrap processor) que el AP está listo. Entonces entra al scheduler para correr tareas.

7. El BSP espera hasta que todos los APs están listos. Entonces entra en el scheduler para correr tareas también.

Después de todos los pasos de arriba, RMTX está corriendo SMP en el modo real de 16 bis. Como en SMP\_MTX, cada CPU trata de correr tareas de su propia ready queue. Los procesos listos para ejecución son distribuidos entre las ready queues para balancear la carga de procesamiento de las CPUs.

15.8.2 Kernel SMP\_RMTX

Para soportar SMP, el kernel RMTX debe ser modificado. Lo siguiente describe los cambios que son específicos para el kernel RMTX de modo real.

1.CPU ID y PROCs en ejecución: En el modo real de 16 bits, permitimos que el registro ES lleve el número de CPU ID. Para identificar los PROCs corriendo en diferentes CPUs, definimos PROC \*run[NCPU] y hacemos que run[i] apunte a la PROC que está actualmente ejecutándose sobre CPUi. Entonces definimos running como

#define running run[cpuid()]

Esto permite que el símbolo running sea usado en el código C del kernel sin cambio alguno. En el código de ensamblador, simplemente usamos ES como un índice para acceder a la PROC corriendo sobre la CPU.

2.Cambios al kernel RMTX para SMP

2.1 tswitch() en el archivo ts.s: En SMP, podrían haber varios procesos ejecutándose en paralelo. Cuando un PROC corriendo sobre una CPU llama a tswitch() para cambiar de proceso, debemos saber cuál es el PROC que está llamando para guardar su contexto. Así que modificamos tswitch() a tswitch(running), pasando como parámetro el apuntador a PROC running. Después de guardar su contexto, el proceso llama a scheduler(), el cual devuelve un apuntador a la siguiente estructura PROC a ejecutar. De forma similar a SMP en el modo protegido, durante el cambio de tarea el proceso running actual debe adquirir el spinlock de la CPU, el cual es liberado por la siguiente estructura PROC a ejecutar cuando esta retoma su ejecución.

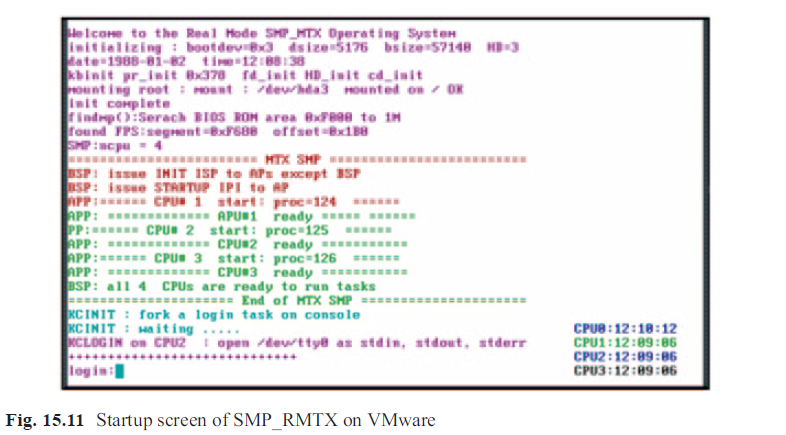
2.2 Rutinas de entrada y salida de interrupción: La CPU0 maneja todas las interrupciones de PIC (Programmable Interrupt Controller). Las rutinas de entrada y salida de interrupción no necesitan cambiarse para CPU0. Dado que las PROCs corriendo sobre APs también hacen syscalls y manejan las interrupciones de local timers, ambos la macro INTH y el código ireturn son modificados ligeramente usando el registro ES para identificar la PROC corriendo sobre una CPU.

2.3 Otros cambios al kernel MTX: Los cambios de arriba son específicos al kernel RMTX de modo real. Otros cambios al kernel RMTX son exactamente los mismos como aquellos en el modo protegido. Por lo tanto, no los repetiremos aquí.

3.Debido al límite de segmento de código de 64KB, el kernel SMP MTX de modo real no tiene suficiente espacio para usar algoritmos paralelos. Así que este kernel usa los algoritmos UP (Uni Processor) protegidos por spinlocks. También, la planificación de procesos es solamente por rebanada de tiempo.

15.8.3 Sistema de demostración SMP\_RMTX

Sobre el CD de instalación de MTX, SMP\_RMTX es SMP MTX en el modo real de 16 bits. Corre sobre máquinas virtuales VMware con múltiples CPUs. La Figura 15.11 muestra la pantalla de SMP\_RMTX sobre VMware con cuatro CPUs.



15.8.4 Limitaciones y Trabajo Futuro

El sistema SMP MTX parece funcionar pero hay algunos problemas menores. Invitamos a los lectores interesados a revisar esos asuntos adicionales.

1.En el sistema, CPU0 maneja todas las interrupciones PIC. Cada AP solamente maneja sus interrupciones timer de local APIC. Estrictamente hablando, este no es un sistema SMP en el verdadero sentido. Para permitir que los APs manejen otras interrupciones, debemos configurar el IOAPIC para enrutar las interrupciones a diferentes CPUs. No está claro si esto es posible en modo real.

2.Incialmente, el sistema tenía un problema de rompecabezas (puzzling) de pérdida de EOIs. Después de correr por unos cuantos minutos, todos los timers APIC se detendrían, causando que el sistema o se colgara o callera de regreso al modo UP. El problema fue eventualmente trazado a la rutina BIOS INT15-87. El kernel SMP\_RMTX se apoya en el registro ES para identificación de la CPU pero INT 15-87 también usa ES. Aunque guardamos ES antes de llamar a INT 15-87 y restauramos ES después, parece que la rutina INT 15-87 habilita las interrupciones tan pronto como ésta termina. Si otra interrupción de timer APIC ocurre antes de que ES sea restaurado, el manejador de interrupción fallaría, resultando en EOIs perdidos. Así que tratamos de usar CLI para deshabilitar interrupciones inmediatamente después de llamar a INT 15-87. A pesar de esto, las interrupciones aun ocurrían, aunque muy infrecuentemente, en el hueco de tiempo entre el final de la rutina INT 15-87 y la instrucción CLI. Dado que no podemos controlar el comportamiento de la rutina de BIOS INT 15-87, tenemos que usar valores de contador timer de APIC grandes y también revisar ES en el código de entrada del manejador de interrupción. Si el contenido de ES no es un CPU ID válido, esto debe ser debido a una interrupción de timer de APIC. En ese caso, nosotros simplemente ejecutamos un EOI pero imprimimos un mensaje EOI\_alert. El sistema trabaja normalmente solamente después de estos parches. Ahora puede correr por díassin perder algún EOI. Podría ser posible evitar este problema no usando la rutina BIOS. Por ejemplo, switchear la CPU al modo protegido, escribir al registro APIC y switchear la CPU de regreso al modo real. Pero entonces la pregunta es: si switcheamos la CPU al modo protegido, podríamos correr SMP en modo protegido, así que, ¿por qué molestarse en switchear la CPU de regreso?

3. El sistema corre muy bien sobre VMware, más recientemente sobre VMware-player 6.0.2 bajo Slackware Linux 14.1. Este debe también correr sobre PCs reales multicore, pero esto no está confirmado dado que el autor del libro [Wang], K. C. Wang, no pudo encontrar una alguna PC multicore que aun soporte drives IDE. Sin embargo, **no corre sobre QEMU**. Parece que los APs en el ambiente SMP de QEMU no responden a algunos IPI si el BSP está en el modo real, así que el sistema solo se cuelga.

A pesar de esos problemas menores, el sistema SMP\_RMTX demuestra que sin duda es posible usar SMP en el modo real de 16-bits. Hasta donde el autor del libro [Wang], (K.C. Wang) conoce, este sistema es probablemente el único sistema SMP en modo real de 16 bits en existencia. Si el sistema es útil o no, aún está por verse.

Abreviaturas

AP – Application Processor

APIC – Advanced Programmable Interrupt Controller

ASMP – Asymmetric MultiProcessor

BSP – BootStrap Processor

FPS – Floating Pointer Structure

IPI – Inter-Processor Interrupt

MP - MultiProcessor

MTX – MultiTasking eXecutable

NUMA – Non-Uniform Memory Access

PMTX – Protected Mode MTX

SMP – Symmetric MultiProcessor

SO – Sistema Operativo

UMA – Uniform Memory Access

UP - UniProcessor

Referencias

[Dietrich y Walker] Dietrich, S., y Walker, D., “The evolution of Real-Time Linux”, <http://www.cse.nd.edu/courses/cse60463/www/amatta2.pdf>, 2015.

[Liu y Layland] Liu, C.L., Layland, J.W., “Scheduling Algorithm for Multi-programming in a Hard Real-Time Environment,” J. ACM, Vol. 20, pp. 40-61, 1973.

[Wang] K.C. Wang, Design and Implementation of the MTX Operating System. Springer International Publishing, 2015.