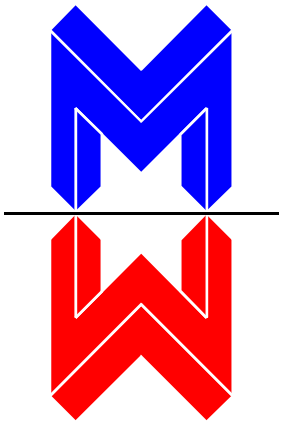
Minix SMP

Una extensión de la arquitectura microkernel a multiprocesadores simétricos



http://webepcc.unex.es/~jalvarez/minixsmp

Jesús María Álvarez Llorente

Departamento de Informática

Universidad de Extremadura

MINIX SMP

1 Introducción

Se presenta en este trabajo Minix SMP, una extensión del sistema operativo Minix 2.0.0 *[23,9]* sobre una arquitectura Intel de multiprocesamiento simétrico (SMP). El principal objetivo es identificar principios de diseño de microkernels POSIX sobre procesadores simétricos y ganar experiencia en su implementación trabajando sobre un caso real. Basándonos en un sistema operativo microkernel sencillo como es Minix, y con un reducido conjunto de cambios y ampliaciones que abarquen el interfaz con el hardware multiprocesador, es fácil obtener un sistema SMP. Nuestra experiencia demuestra que la mayor dificultad estriba en la comprensión de ese hardware subyacente.

Comenzamos este documento con un capítulo introductorio en el que encuadramos el trabajo dentro del ámbito de los sistemas operativos multiprocesador y de Minix. En los capítulos segundo y tercero estudiaremos las bases de partida del trabajo, analizando los principales aspectos de la arquitectura hardware, los sistemas multiprocesador Intel para los que se ha desarrollado Minix SMP, y software, la arquitectura del sistema operativo Minix sobre el que se ha realizado la extensión. En el cuarto capítulo desvelamos los principios de diseño aplicados en el desarrollo de Minix SMP, identificando los diferentes problemas que plantea el nuevo sistema y describiendo las soluciones propuestas, para, en el siguiente capítulo, profundizar en los detalles de implementación de dichas soluciones. Finalmente recogemos una reflexión acerca de las aportaciones logradas mediante el trabajo, así como la continuación prevista del mismo.

Este trabajo ha sido financiado por el proyecto CICYT nº TIC99-0960, titulado "*Diseño e Implementación de Algoritmos de Procesado de Señal de Altas Prestaciones para Reconocimiento de Voz en Condiciones Adversas*".

1.1. Justificación y objetivos

Minix, el sistema operativo compatible con UNIX diseñado por Tanenbaum en 1987 *[25,24]* es, desde hace años, el referente de estudio por excelencia para la docencia de Sistemas Operativos. No en vano, ese fue el motivo de su creación: poder proporcionar al estudiante una visión real de cómo está construido un sistema operativo. Minix, no obstante, es algo más que una herramienta docente. Entre sus características más importantes podemos destacar la siguientes:

1. Es un sistema microkernel, lo que le confiere de partida un diseño fuertemente estructurado, sencillo (dentro de la complejidad de un sistema operativo), fácil de comprender y de modificar.
2. Es conforme POSIX 1003.1a.

3. Se basa en la arquitectura Intel para computadores personales, la más extendida, accesible y asequible hoy en día.
4. Es un sistema pequeño y manejable, libre de las optimizaciones y complicaciones necesarias en los sistemas comerciales.
5. Es de libre distribución, con lo que es muy fácil de obtener y utilizar. Además viene acompañado por un texto que lo describe en profundidad *[23]*.
6. Es un caso de estudio real. No es un simulador ni un sistema idealizado.

En los últimos 4 o 5 años se han venido popularizando y abaratando las arquitecturas SMP (*Symmetric Multiprocessing*, Multiprocesador Simétrico), basadas en múltiples procesadores con memoria compartida con una estructura semejante a la mostrada en la Figura 1. Si bien podemos encontrar referencias a estas arquitecturas en cualquier bibliografía actual sobre sistemas, lo cierto es que es difícil encontrar textos que, como *[18]*, abarquen con cierta profundidad y criterio didáctico lo referente al diseño de sistemas operativos que soporten tales arquitecturas. Es cierto que el estudio de Linux SMP puede ayudar en este sentido, pero consideramos que no es un caso de estudio idóneo debido a la complejidad que se desprende de su diseño monolítico. No es una herramienta adecuada para aprender porque que no es ese el objetivo con el que Linux fue creado.

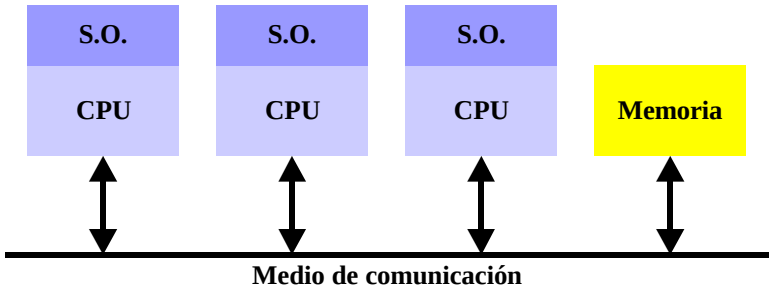


Figura 1. Esquema genérico de un multiprocesador de memoria compartida

De entre las muchas propuestas de arquitectura multiprocesador que encontramos en el mercado, de las cuales trataremos más adelante, la especificación de Intel, lejos de resultar de grandes prestaciones, es la más asequible, por lo que resulta adecuada para investigar sobre ella los principios de la multicomputación. Hoy en día es fácil encontrar todo tipo de textos acerca de la arquitectura Intel, pero sólo en lo relativo a los aspectos tradicionales de dicha arquitectura (arquitecturas del 8086 al 80386). Intel proporciona de forma gratuita los manuales y especificaciones de todos sus productos, pero estos documentos, aun siendo precisos, rigurosos y completos, distan de ser adecuados para aprender, sirviendo más bien como una guía de referencia para quien conoce bien las bases de la arquitectura.

La especificación multiprocesador (MP) de Intel *[8]* está constituida por un relativamente pequeño documento que complementa los tres volúmenes que forman el manual para desarrollo de software para la arquitectura Intel de 32 bits *[6]*. En el tercer volumen de dicho manual, dedicado a la programación de sistemas, ya se introducen algunas nociones sobre multiprocesamiento, centradas en lo relativo a la arquitectura interna al procesador. La especificación MP se centra más en los aspectos relacionados con la configuración del sistema de procesadores. Más compleja es la localización de información referente al resto de elementos que acompañan al sistema multiprocesador, para lo que hubo que recurrir a las especificaciones avanzadas de un chipset concreto *[5]*.

Todas estas dificultades nos conducen a repetir el planteamiento de Tanenbaum: crear nuestro propio sistema operativo multiprocesador, diseñado desde una perspectiva claramente didáctica, y acompañado de una completa documentación, que facilite el aprendizaje al estudiante.

Afortunadamente no es necesario partir de cero. Minix constituye, como veremos, un inmejorable punto de partida sobre el cual construir un sistema SMP, ya que:

1. Permitirá que todo lo aprendido sobre sistemas monoprocesador siga siendo útil al estudiar SMP.
2. Ayudará a comprender las diferencias entre la estructura de un sistema operativo monoprocesador y multiprocesador, proporcionando una valiosa experiencia que transmitir a nuestros alumnos.
3. Proporcionará una base bien diseñada, estructurada, sencilla, fácil de comprender y modificar, pequeña, manejable, accesible y real sobre la que construir, y eso ayudará a que el resultado retenga también todas estas propiedades.

Con estos objetivos, y partiendo de la versión 2.0.0 de Minix *[23,9]*, hemos realizado el trabajo de extender el sistema operativo docente por excelencia a la arquitectura SMP de Intel para obtener lo que llamaremos de ahora en adelante Minix SMP.

1.2. Minix

En las primeras distribuciones de UNIX (hasta la versión 6), el código fuente se distribuía libremente, con autorización de AT&T, y se estudiaba frecuentemente, existiendo incluso textos como en de John Lions (de la Universidad de New South Wales, en Australia), que describían su operación línea por línea y que se utilizaban (con permiso de AT&T) como libro de texto en cursos universitarios sobre sistemas operativos *[23]*.

AT&T entregó la versión 7 de UNIX con una licencia que prohibía el estudio del código fuente en cursos con el objeto de evitar poner en peligro su condición como secreto comercial, por lo que muchas universidades descartaron el estudio de UNIX, enseñando sólo teoría, lo que proporciona una visión desproporcionada de lo que en realidad es un sistema operativo, centrándose en cuestiones como los algoritmos de planificación, que en la práctica no son realmente tan importantes, y dejando de lado otros aspectos más importantes, como la E/S y los sistemas de archivo, difícilmente abarcables de forma teórica.

Para remediar esta situación, Andrew S. Tanenbaum, de la *Vrije Universiteit Amsterdam*, decidió escribir desde cero un nuevo sistema operativo que fuera compatible con UNIX pero sin utilizar una sola línea de código de AT&T. Así, este sistema no violaba la licencia de AT&T, de modo que podía utilizarse para la docencia y los estudiantes podían analizar minuciosamente un sistema operativo real. Se trababa de un sistema suficientemente pequeño para que alguien que no fuera un maestro en sistemas operativos pudiese entender la forma en que trabajaba, por lo que se bautizó como Minix (*Mini-UNIX*)

Además de evitar los problemas de la licencia de AT&T, Minix se escribió una década después de UNIX y se estructuró de forma más modular y se diseñó para ser legible en contraposición a UNIX, que se diseñó para ser eficiente.

La primera versión de Minix se diseño para ser compatible con la versión 7 (V7) de UNIX, debido a su simplicidad y elegancia, y al igual que UNIX, se escribió en lenguaje de programación C, con el objetivo de ser fácil de portar a diversas arquitecturas. La implementación inicial se hizo para una arquitectura IBM PC básica, puesto que tenia un uso extenso y asequible.

El libro *Operating Systems, Design and Implementation* (de 1987) *[24]* hace referencias explícitas al código de la Minix, parte del cual se incluye dentro del texto. Sucesivas versiones de Minix se publican con completos manuales de referencia como *[22]*. En 1997 se publica la segunda edición de *Operating Systems, Design and Implementation [23]*, que incluye un CD-ROM con la distribución 2.0.0 de Minix *[9]*. Entre las mejoras de esta nueva versión destaca el hecho de que sea conforme a la norma POSIX 1003.1a. Además, el sistema se ve ampliado con las siguientes características:

- Mayor soporte de periféricos (p.e. CD-ROM).
- Soporte para hardware más moderno (procesadores i386 y superiores), discos de gran tamaño, etc.
- Soporte para red (TCP/IP).

La versión 1.5 de Minix fue portada a diversas arquitecturas, como *Atari*, *Amiga*, *Apple Macintosh y SPARC*. La 2.0.0 *[9]* no ha sido aún portada a todas ellas, aunque sí existe una versión que funciona bajo

SunOS, y también ha sido probado su funcionamiento bajo un simulador de PC llamado *Bochs* disponible para diversas plataformas UNIX.

Minix, además, ha servido como base para diversas extensiones, entre las que podemos destacar:

- *RT-Minix*, una extensión para tiempo real sobre minix 2.0.0 *[12]*.
- *Minix-VMD*, una extensión sobre minix 1.7.0 con soporte para memoria virtual *[13]*.

También podemos encontrar referencias que hablan de sistemas distribuidos basados en Minix *[27,16]*. Todo este trabajo demuestra el interés de utilizar Minix en entornos y arquitecturas actuales. No existe, sin embargo, ninguna extensión de Minix para arquitecturas multiprocesador. Tampoco se ha previsto esta extensión en las últimas versiones publicadas de Minix, la 2.0.2 *[10]* y 2.0.3 *[11]*, en las que sólo se mejoran algunos aspectos del soporte de red, se actualizan algunos controladores y se incluye por primera vez la posibilidad de que Minix funcione bajo el sistema DOS.

Dada la expansión que estamos viviendo de las arquitecturas multiprocesador, a la que ya se están adaptando los sistemas operativos actuales (incluso los que tienen una aplicación que roza lo doméstico, como Windows), resulta interesante ampliar el conjunto de extensiones de Minix a estas plataformas.

1.3. Multiprocesamiento Simétrico

1.3.1. Multiprocesamiento

El multiprocesamiento es una tendencia significativa en el campo de la computación, consistente en configurar un sistema de computación con varios procesadores. Aunque no se trata de un enfoque nuevo, sí que es cierto posee actualmente grandes perspectivas debido a las ventajas que ofrecen sobre los sistemas monoprocesador de altas prestaciones, entre las que podemos destacar la superación de la limitación tecnológica a la que se acercan los actuales procesadores y la fiabilidad, ya que si un procesador falla, los restantes continúan operando. Además, se puede concebir un diseño modular, que proporciona una flexibilidad importante y facilita la expansión de la capacidad. Así, las metas de los sistemas de multiprocesamiento generalmente son la fiabilidad y la disponibilidad muy altas, así como, por supuesto, el incremento del poder de computación.

El principal inconveniente que se presenta es la explotación de esta capacidad, ya que requiere unas técnicas y métodos (paralelismo) diferentes de los de la computación tradicional (secuencial) y difíciles de aplicar. La principal consecuencia es que no es posible aprovechar el software existente para la computación secuencial si lo que queremos es explotar toda la potencia del multiprocesamiento.

1.3.2. Hardware de los multiprocesadores

Desde el punto de vista del hardware, el problema clave es determinar los medios de conexión de los procesadores y las unidades de almacenamiento *[4]*. Un multiprocesador contiene dos o más procesadores con capacidades aproximadamente comparables, todos ellos compartiendo el acceso a un almacenamiento común y a dispositivos de E/S (organización MIMD, *Multiple Instruction Multiple Data*, según la clasificación de *Flynn*). En este sentido se suele hacer distinción con el término *multicomputadores*, donde cada procesador dispone de almacenamiento y espacios E/S propios no compartidos. Entonces se suele hablar de hardware débilmente acoplado, frente al fuertemente acoplado de los *multiprocesadores* propiamente dichos.

Las organizaciones más comunes en multiprocesadores son las siguientes *[4,15]*:

- En bus (Figura 2, también conocido como tiempo compartido). Usa un solo bus de comunicación entre todas las unidades funcionales (procesadores, memoria, E/S). Es una organización económica, simple y flexible pero con una sola vía de comunicación, por lo cual el sistema falla totalmente si

falla el bus y el ancho de banda de transmisiones está limitado por el bus, lo que limita la cantidad de elementos que se pueden conectar *[15]*.

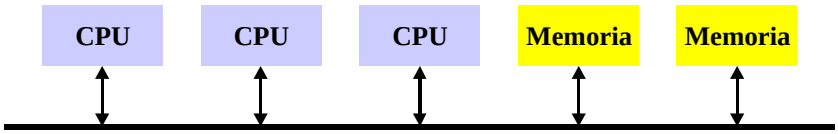


Figura 2. Multiprocesadores en bus

- Matriz de barras cruzadas y conmutadores (Figura 3). Los procesadores y elementos de memoria se conectan mediante una red de conmutadores, un circuito capaz de conectar cada unidad de proceso con cualquier módulo de memoria de forma temporal, para lo cual dispone de una serie de conmutadores que pueden abrir o cerrar determinadas partes de un circuito cambiando así las conexiones establecidas en la red mediante diversas combinaciones válidas. En el caso ideal, cada procesador se puede conectar a cualquier módulo independientemente de los accesos del resto de procesadores (este tipo de red se denomina *crossbar switch*), aunque dos procesadores no pueden acceder nunca simultáneamente al mismo módulo de memoria.

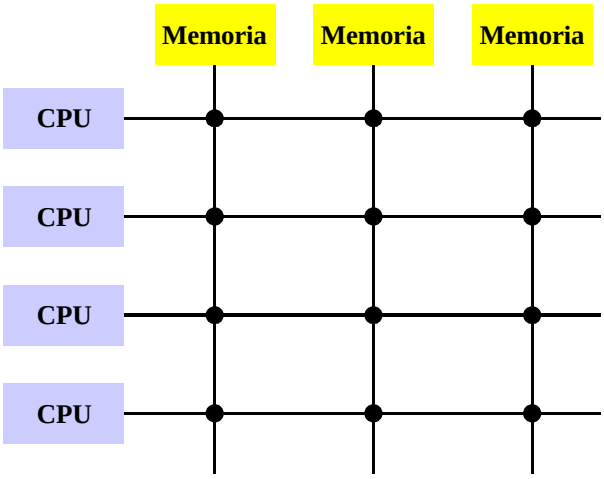


Figura 3. Multiprocesadores con crossbar switch

- Almacenamiento de interconexión múltiple (Figura 4). Cada elemento se conecta directamente con un conjunto de elementos, manipulando su propia lógica de control. Esta organización suele descartarse por ser excesivamente costosa.

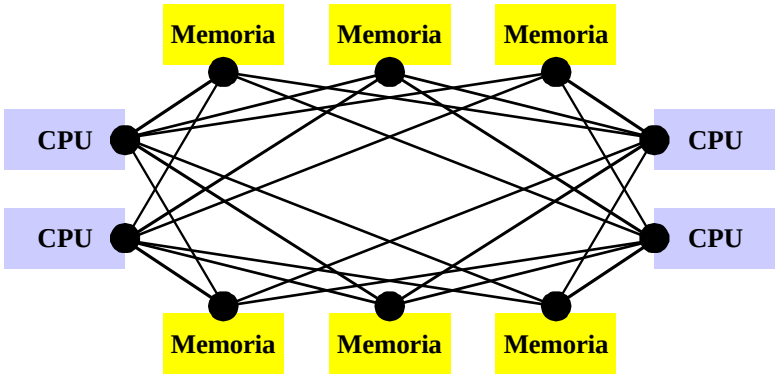


Figura 4. Multiprocesador con almacenamiento de interconexión múltiple

Junto con la estructura de las interconexiones se permiten varias organizaciones de procesadores, entre las que se destacan:

- Maestro/Esclavo (Maestro/Satélite). Un procesador está diseñado como el maestro y los otros como satélites, donde el maestro puede realizar tanto operaciones de E/S como cálculo, mientras que los satélites sólo realizan computaciones. Los procesos limitados por computación pueden ejecutarse con efectividad en los satélites. Los procesos limitados por la E/S ejecutados en los satélites generan frecuentes llamadas de servicios al procesador maestro, pudiendo resultar ineficientes. Si falla un satélite se pierde capacidad computacional pero el sistema no falla. Si falla el maestro el sistema falla al no poder efectuar operaciones de E/S, por lo que un satélite debería asumir las funciones del maestro previo cambio de los periféricos y reinicio del sistema.
- Multiprocesamiento Simétrico. Todos los procesadores tienen las mismas capacidades, por lo tanto todos pueden hacer E/S. Esta capacidad deberá ser explotada correctamente por el software (sistema operativo) para que el multiprocesamiento simétrico sea una realidad.

1.3.3. Software de los multiprocesadores

La explotación de los sistemas multiprocesador pasa por la utilización del software adecuado, empezando por el sistema operativo. Las capacidades funcionales de un sistema operativo de multiprocesadores incluyen:

- Asignación y administración de recursos.
- Protección de tablas y conjuntos de datos.
- Prevención contra el interbloqueo del sistema.
- Terminación anormal.
- Equilibrio de cargas de E/S.
- Equilibrio de carga del procesador.
- Reconfiguración.

Las tres últimas son especialmente importantes en Sistemas Operativos de multiprocesadores, donde es fundamental explotar el paralelismo en el hardware y en los programas. Esta necesidad complica el diseño del sistema introduciendo los mismos problemas que aparecen en la computación paralela al ser necesaria una comunicación y sincronización entre los procesadores y los programas que se ejecutan en éstos.

La principal complicación viene dada por el acceso desde los procesos a los servicios centralizados en el sistema operativo, ya que manipulan estructuras de datos comunes y utilizan recursos compartidos que no pueden ser accedidos de forma simultánea. Para resolver estos inconvenientes se utilizan varias organizaciones del Sistema Operativos, entre la que podemos encontrar las siguientes:

- Maestro/Esclavo.

- Ejecutivos separados.
- Multiprocesamiento Simétrico (SMP).

1.3.3.1. Sistema Operativo Maestro/Esclavo

Es la organización más fácil de implementar. No logra la utilización óptima del hardware dado que sólo el procesador maestro puede ejecutar el Sistema Operativo y el procesador esclavo sólo puede ejecutar programas del usuario. Las interrupciones generadas por los procesos en ejecución en los procesadores esclavo que precisan atención del Sistema Operativo deben ser atendidas por el procesador maestro y por ello pueden generarse largas colas de peticiones pendientes.

1.3.3.2. Ejecutivos separados

Cada procesador tiene su propio Sistema Operativo y responde a interrupciones de los usuarios que operan en ese procesador, de forma parecida a como ocurre en un sistema distribuido. Existen tablas de control en memoria compartida con información global de todo el sistema (por ejemplo, lista de procesadores conocidos por el Sistema Operativo) a las que se debe acceder utilizando exclusión mutua. Es más fiable que la organización anterior. Cada procesador controla sus propios recursos dedicados, por lo que la contención sobre las tablas comunes del Sistema Operativo es mínima. Los procesadores no cooperan en la ejecución de un proceso individual, que habrá sido asignado a uno de ellos.

1.3.3.3. Multiprocesamiento Simétrico (SMP)

Es la organización más complicada de implementar y también la más poderosa y confiable. El Sistema Operativo administra un grupo de procesadores idénticos, donde cada procesador posee capacidades funcionales completas y puede utilizar cualquier dispositivo de E/S y referenciar a cualquier unidad de almacenamiento. El Sistema Operativo precisa código reentrante y exclusión mutua. Es posible equilibrar la carga de trabajo más precisamente que en las otras organizaciones. Todos los procesadores pueden cooperar en la ejecución de un proceso determinado. Es más eficiente que las organizaciones anteriores. Cada procesador puede ejecutar el planificador para buscar el siguiente trabajo a ejecutar, de forma que un proceso determinado se ejecuta en diferentes procesadores en distintos momentos. Utiliza una cola compartida de trabajos y cada procesador puede seleccionar trabajos de ella, con lo cual se equilibra la carga entre los procesadores.

Esta es la organización que se ha implementado en Minix SMP. Cada procesador accederá a la cola de procesos en busca de trabajo. Las peticiones al sistema serán ejecutadas por el mismo procesador que las invoque desde el proceso de usuario que las solicite, y las interrupciones del hardware se repartirán indistintamente a todos los procesadores.

2 Arquitectura Multiprocesador de Intel

Minix SMP se ha desarrollado para la arquitectura multiprocesador que Intel propone en su especificación MP 1.4 *[8]*. Un sistema compatible con la especificación MP 1.4 de Intel es un multiprocesador simétrico de memoria compartida. Los primeros procesadores de la familia Intel capaces de permitir configuraciones en multiprocesador son los 80486DX, utilizando algunos dispositivos adicionales. La siguiente generación de la familia x86, los Pentium están diseñados desde el principio para permitir configuraciones multiprocesador, produciéndose un cambio significativo en algunos aspectos importantes del hardware al tal efecto. El presente trabajo se ha desarrollado exclusivamente para soportar procesadores Intel Pentium y posteriores.

Los 80486DX son los primeros Intel en incorporar sistemas de cache en el propio procesador. Se trata de un detalle importante, ya que en un sistema compatible con la especificación MP 1.4 de Intel se exige que la coherencia entre la memoria principal y los distintos sistemas de cache quede resuelta por el hardware y sea transparente al software. Este detalle va a facilitar enormemente la construcción del sistema multiprocesador, ya que apenas vamos a preocuparnos por la existencia de una memoria cache distribuida entre los procesadores, como veremos en el apartado 2.5.

A lo largo de la especificación MP encontramos la idea constante de compatibilidad entre el sistema multiprocesador y un sistema de similares prestaciones con un solo procesador. Un sistema compatible con la especificación MP se comportará como un monoprocesador hasta el mismo instante en que intentemos explotar las capacidades de multiprocesamiento. Así, sólo tendremos que preocuparnos por estas nuevas capacidades y podremos aprovechar la práctica totalidad del software creado para monoprocesador.

Una de las claves de la arquitectura Intel MP reside en la ampliación del sistema de interrupciones. En las arquitecturas monoprocesador las interrupciones de los dispositivos E/S se manejaban mediante el PIC (*Programmable Interrupt Controller*) i8259, conectado directamente al procesador. Para el procesamiento simétrico de interrupciones se ha diseñado un sistema distribuido de controladores formado por unos circuitos llamados APIC (*Advanced Programmable Interrupt Controller*) conectados entre sí mediante un bus dedicado (ICC, *Interrupt Controller Communications*), como muestra la Figura 5. Existen dos tipos de APIC:

1. Cada procesador del sistema posee un APIC llamado local, integrado en el propio procesador, y conectado al bus dedicado.
2. En el conjunto del sistema existe al menos un APIC de entrada/salida (I/O APIC), conectado al bus dedicado y a las distintas fuentes de interrupción del sistema (buses).

En la secuencia de arranque, el BIOS debe configurar el sistema para que funcione como un monoprocesador tradicional. Esto implica que sólo uno de los procesadores del sistema estará activo (BSP, *Bootstrap Processor*), mientras los demás (AP, *Application Processor*) esperan inactivos. El control de interrupciones debe quedar configurando de modo que responda de forma compatible con el PIC i8259. Así, el antiguo PIC sigue funcionando en colaboración con el sistema de APIC. Las distintas fuentes de interrupción se conectan tanto al PIC como al I/O APIC. El PIC, a su vez, se conecta directamente a todos los APIC locales de cada procesador, mientras el I/O APIC se comunica con los APIC locales a través del bus ICC. Inicialmente el I/O APIC se encuentra inhabilitado, de manera que las fuentes de interrupción externa sólo afectan al PIC. Éste hace llegar a todos los procesadores las señales de interrupción, pero deben pasar a través del APIC local de cada uno. Inicialmente sólo el APIC local del BSP se encuentra programado para trasladar las señales de interrupción del PIC hasta el procesador.

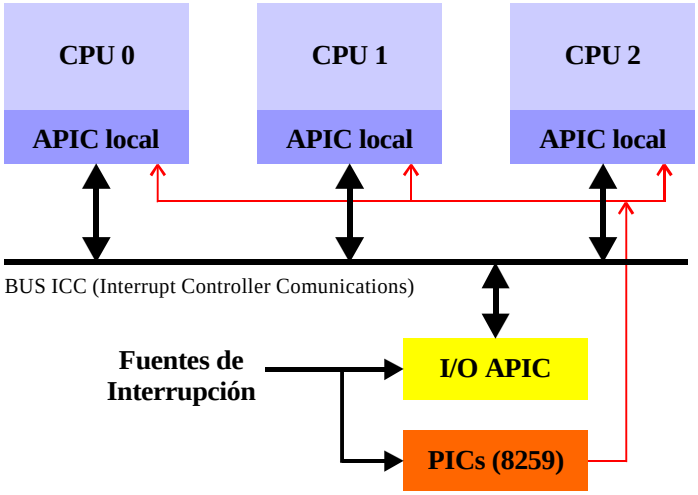


Figura 5. Sistema distribuido de controladores de interrupciones (APIC).

Las funciones del sistema distribuido de APIC frente al PIC tradicional son principalmente dos:

1. Permitir la distribución inteligente entre los procesadores de las señales de interrupción procedentes del I/O APIC, es decir, de las fuentes externas de interrupción. Esto permite la programación de distintos modelos de procesamiento simétrico mediante diferentes estrategias de reparto de interrupciones, por ejemplo, entrega al procesador menos ocupado, al menos prioritario, a todos los procesadores, etc.
2. Distribuir señales de interrupción procedentes de cualquiera de los APIC locales (IPI, *Inter-Processor Interrupts*), lo que representa una forma de comunicación entre los procesadores, necesaria, entre otras razones, para la iniciación y detención de los AP, redistribución de interrupciones, etc.

La primera de estas funciones requiere la inhabilitación del PIC y su sustitución por el uso del I/O APIC, que es capaz de distribuir las interrupciones de forma programada. Si no deseamos aprovechar esta facilidad, podemos seguir utilizando el PIC, que distribuirá las interrupciones a todos los procesadores de manera que alguno de ellos reconocerá y aceptará cada interrupción (pero no tenemos ningún control sobre la forma en que esto va a ocurrir).

Los APIC son dispositivos E/S mapeados en memoria. El acceso a estos dispositivos se hace a través de direcciones no fijadas por la especificación MP. Tampoco se establece la configuración exacta del sistema de APIC: número, identificación, estructura, etc. Por esta razón es necesario que el sistema provea una cierta información acerca de su configuración. Esto es tarea del BIOS: preparar una zona de memoria con una estructura (*MP Configuration Table*) con toda la información sobre el conjunto de procesadores, APIC, buses, etc., necesaria para que el sistema operativo pueda tomar el control del multiprocesador.

Cada APIC (tanto local como I/O) se identifica con un número único en el sistema. Este número permite además identificar los distintos procesadores (mediante la identificación de su APIC local). La numeración, que se puede programar por software, no tiene por qué ser consecutiva ni seguir ningún tipo de patrón, aunque lo habitual es que el BSP esté identificado con el APIC 0, los AP con valores consecutivos 1, 2, etc., y los I/O APIC con los valores siguientes.

Por último, la arquitectura Intel proporciona otras facilidades necesarias para la implementación de sistemas multiprocesador, como instrucciones atómicas, bloqueo del bus de datos para determinadas instrucciones (accesos exclusivos a direcciones de memoria compartida), etc. Veremos todas estas cuestiones de forma detallada en las siguientes secciones.

2.1. Tabla de configuración MP

Es función del BIOS preparar, durante la secuencia de arranque, una zona de memoria con información acerca de la configuración multiprocesador concreta del computador. Esta información se divide en dos bloques: la *Floating Pointer Structure* (FPS) y la tabla de configuración MP (MPCT, *MP Configuration Table*). La primera contiene información sobre la versión de la especificación MP y un puntero a la zona de memoria donde encontrar la segunda estructura, como indica la Figura 6.

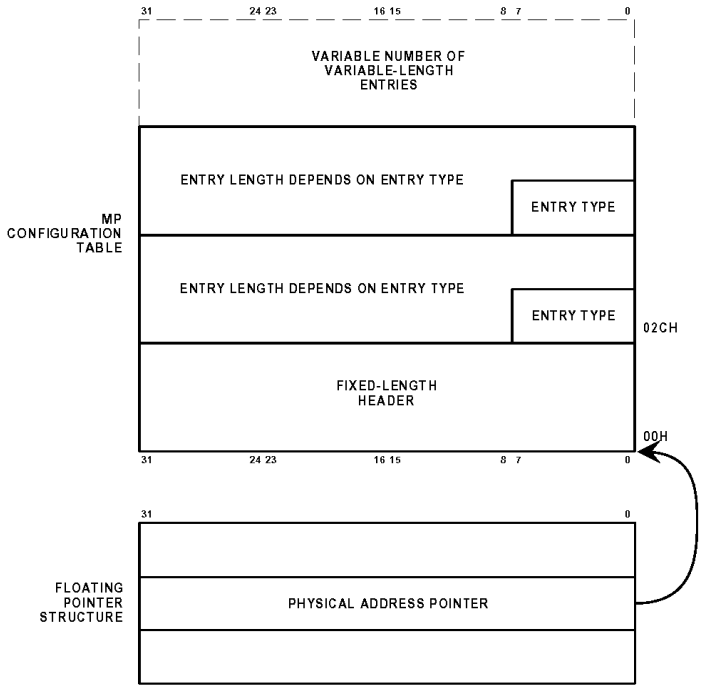


Figura 6. Estructuras de datos de la configuración MP

2.1.1. Floating Pointer Structure

La FPS no se encuentra en una dirección de memoria fija. Según la especificación de Intel se puede encontrar:

- dentro del primer kilobyte del EBDA (*BIOS Data Area*), o
- dentro del último kilobyte de memoria base (639K-640K o 511K-512K, según la memoria del sistema), en el caso de que el EBDA no se encuentre definido, o
- en el área de ROM BIOS, entre las direcciones 0F0000h y 0FFFFFh.

La Tabla de Configuración MP es opcional. La FPS contiene un byte que puede especificar una configuración estándar o bien una configuración especial. Si la configuración es estándar, la Tabla de Configuración no es necesaria, ya que toda la información que contendría está recogida en el estándar.

La FPS (Figura 7, Tabla 1) es una estructura de 16 bytes, alineada en párrafos de 16 bytes. Para encontrarla en memoria dispone de un campo con una firma (*SIGNATURE*) y otro de verificación (*CHECKSUM*).

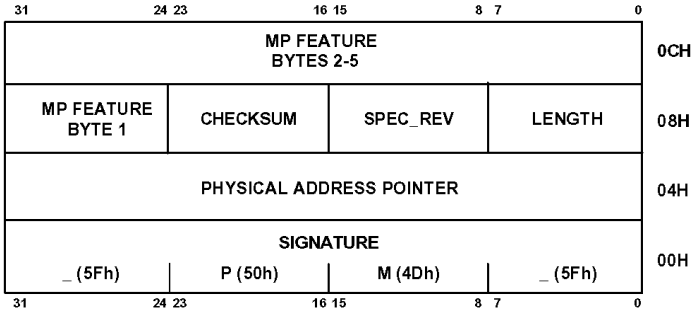


Figura 7. Estructura de la FPS

Campo	Offset (bytes:bits)	Longitud (bits)	Descripción
SIGNATURE	0	32	Una cadena ASCII con el contenido "_MP_" que sirve para localizar la estructura en la memoria.
PHYSICAL ADDRESS POINTER	4	32	Un puntero que indica la dirección física de la Tabla de Configuración MP.
LENGHT	8	8	Longitud de la FPS en párrafos de 16 bytes. Tiene el valor 01h.
SPEC_REV	9	8	Revisión de la especificación MP. Un valor 04h significa v1.4.
CHECKSUM	10	8	Checksum de la estructura completa, de manera que la suma de todos los bytes de la estructura, incluido éste, debe sumar 0.
MP FEATURE INFORMATION BYTE 1	11	8	Tipo de configuración estándar. Vale cero si la configuración no es estándar y en ese caso existe una Tabla de Configuración MP.
MP FEATURE INFORMATION BYTE 2	12:0 12:7	7 1	Los bits 0-6 están reservados. El 7 (*IMCRP*) indica la presencia del IMCR, un registro que permite configurar diferentes modos de funcionamiento del sistema de PIC y APIC.
MP FEATURE INFORMATION BYTES 3-5	13	24	Reservado para uso futuro.

Tabla 1. Campos de la FPS

En el caso de que se trate de una configuración estándar, habrá que consultar la especificación de Intel y tendremos todos los datos referentes a la organización de la máquina. Caso contrario habrá que analizar la Tabla de Configuración MP que encontraremos en la dirección indicada por el puntero *PHYSICAL ADDRESS POINTER*.

2.1.2. Tabla de Configuración MP

La Tabla de Configuración MP consta de tres partes, una cabecera (Figura 8, Tabla 2) localizada en la dirección indicada por el puntero de la FPS, seguida (en memoria) de una conjunto de entradas de longitud variable dependiendo de la configuración hardware, seguida a su vez de una extensión (*extended table*) consistente otro conjunto opcional de entradas de tamaño variable. La cabecera más el primer conjunto de entradas (no extendidas) forman la llamada *tabla base*.

2.1.2.1. Cabecera

Al igual que en la FPS, en la cabecera encontramos un campo *SIGNATURE* y un campo *CHECKSUM* que nos ayuda a identificar la estructura. Los campos más importantes de esta estructura son dos: *ENTRY COUNT*, que nos indica cuántas entradas de información encontraremos en la tabla base, y *ADDRESS OF LOCAL APIC*, que nos indica cuál es la dirección física en la que se efectúa el acceso E/S al hardware del APIC local de cada procesador (todos los procesadores acceden en la misma dirección). Debemos anotar esta dirección para acceder posteriormente a los servicios del APIC local.

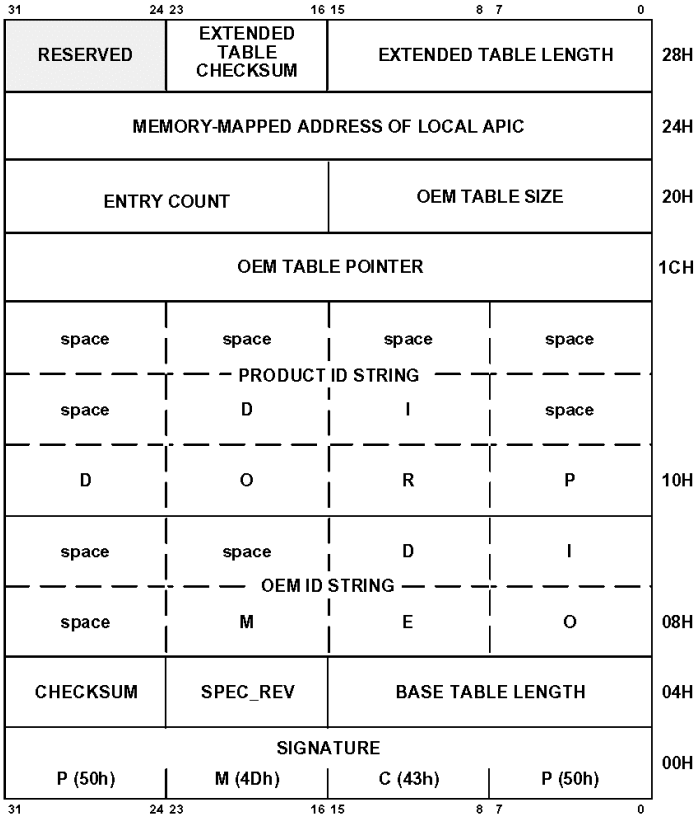


Figura 8. Cabecera de la Tabla de Configuración MP

Campo	Offset (bytes:bits)	Longitud (bits)	Descripción
SIGNATURE	0	32	Una cadena ASCII con el contenido "PCMP" que sirve para localizar la estructura en la memoria.
BASE TABLE LENGHT	4	16	Longitud de la estructura base en bytes (incluyendo la cabecera y el conjunto de entradas normales, no extendidas).
SPEC_REV	6	8	Revisión de la especificación MP. Un valor 04h significa v1.4.
CHECKSUM	7	8	Checksum de la estructura base, de manera que la suma de todos los bytes de la estructura, incluido éste, debe sumar 0.
OEM ID	8	64	Una cadena ASCII de 8 bytes rellenada con espacios que identifica al fabricante.
PRODUCT ID	16	96	Una cadena ASCII de 12 bytes rellenada con espacios que identifica al producto.
OEM TABLE POINTER	28	32	Un puntero a una estructura de datos opcional dependiente del fabricante (cero si no existe).
OEM TABLE SIZE	32	16	Tamaño en bytes de la estructura de datos del fabricante.
ENTRY COUNT	34	16	Número de entradas de la tabla base.
ADDRESS OF LOCAL APIC	38	32	Dirección física de acceso al APIC local.
EXTENDED TABLE LENGHT	40	16	Tamaño en bytes de la tabla extendida opcional, cero en caso de no existir.
EXTENDED TABLE CHECKSUM	42	8	Checksum de la estructura extendida, de manera que la suma de todos los bytes de la estructura, incluido éste, debe sumar 0.

Tabla 2. Campos de la cabecera de la Tabla de Configuración MP

En los bytes siguientes a la cabecera encontramos la secuencia de entradas de información de la tabla base. Existen varios tipos de entrada, cada uno con un tamaño diferente. En todos ellos existe un campo de 8 bits al principio de la estructura indicando el tipo de entrada. Los tamaños de la estructura para cada tipo de entrada son:

- Procesador: 20 bytes.
- Bus: 8 bytes.
- I/O APIC: 8 bytes.
- Asignación de interrupciones E/S: 8 bytes.
- Asignación de interrupciones locales: 8 bytes.

2.1.2.2. Entrada de procesador

Existirá una entrada de procesador por cada CPU presente en el sistema. La entrada (Figura 9, Tabla 3) recoge información, entre otras cosas, sobre el tipo de CPU (*CPU SIGNATURE*), si está habilitada (*CPU FLAGS:EN*) o no, si es el procesador *principal* o no (*CPU FLAGS:BP*), y un número identificativo del APIC del procesador (*LOCAL APIC ID*). En todo sistema existe un procesador *principal* BSP (*Bootstrap Processor*), que es el que ejecuta la secuencia de arranque, y procesadores de aplicación (AP, *Application Processor*), que deben ser arrancados por el BSP.

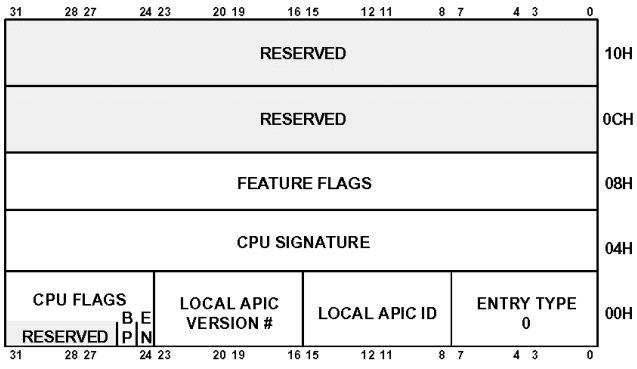


Figura 9. Entrada tipo CPU

Campo	Offset (bytes:bits)	Longitud (bits)	Descripción
ENTRY TYPE	0	8	Tipo de entrada (0h).
LOCAL APIC ID	1	8	Número de identificación del APIC local de este procesador.
LOCAL APIC VERSION #	2	8	Versión del APIC
CPU FLAGS:EN	3:0	1	Vale cero en caso de que el procesador exista pero no pueda utilizarse.
CPU FLAGS:BP	3:1	1	Vale 1 para el BSP y 0 para los AP.
CPU SIGNATURE	4	12	3 campos de 4 bits indicando la versión del procesador (*STEPPING*, *MODEL*, *FAMILY*).
CPU FEATURE FLAGS	8	32	Información semejante a la proporcionada por la instrucción *CPUID*.

Tabla 3. Campos de la entrada tipo CPU

2.1.2.3. Entrada de Bus

Existirá una entrada (identificada como 1h) por cada bus que contiene el sistema. La información de esta entrada no es relevante para la versión actual de Minix SMP, por lo que remitimos a la especificación MP *[8]* para más información.

2.1.2.4. Entrada de I/O APIC

Existirá una entrada de este tipo (identificada como 2h, como vemos en la Figura 10 y en la Tabla 4) por cada unidad I/O APIC que exista en el sistema (al menos una). Contiene información importante como el número de identificación (*I/O APIC ID*) y la dirección de memoria donde se accede (*I/O APIC ADDRESS*).

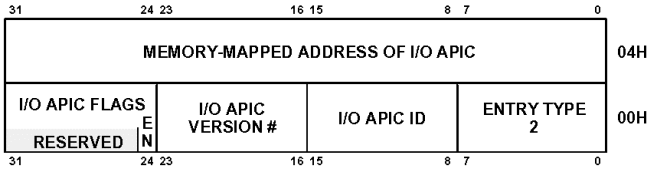


Figura 10. Entrada tipo I/O APIC

Campo	Offset (bytes:bits)	Longitud (bits)	Descripción
ENTRY TYPE	0	8	Tipo de entrada (2h).
I/O APIC ID	1	8	Número de identificación de este I/O APIC en el sistema.
I/O APIC VERSION #	2	8	Versión del APIC.
I/O APIC FLAGS:EN	3:0	1	Vale cero en caso de que el APIC exista pero no pueda utilizarse.
I/O APIC ADDRESS	4	32	Dirección física de acceso a este I/O APIC.

Tabla 4. Campos de la entrada tipo I/O APIC

2.1.2.5. Entrada de asignación de interrupciones E/S

Este tipo de entrada (identificada con 3h) informa sobre la configuración de conexiones entre las fuentes de interrupciones de E/S y el conjunto de I/O APIC, y resulta de utilidad si se utiliza el I/O APIC como controlador de interrupciones. Puesto que en la versión de Minix SMP actual no se hace así, la información de este tipo de entradas no se tiene en cuenta. Remitimos a la especificación MP *[8]* para más información.

2.1.2.6. Entrada de asignación de interrupciones locales

Este tipo de entrada (Figura 11, Tabla 5) identifica las conexiones entre las fuentes de interrupción y las líneas de entrada de los APIC locales. A través de la información de estas entradas podemos asegurarnos de si las fuentes de interrupción del PIC están o no conectadas al conjunto de procesadores.

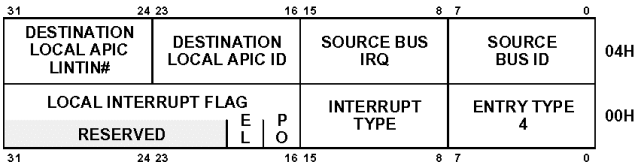


Figura 11. Entrada tipo asignación de interrupciones locales

Campo	Offset (bytes:bits)	Longitud (bits)	Descripción
ENTRY TYPE	0	8	Tipo de entrada (4h).
INTERRUPT TYPE	1	8	Tipo de interrupción, según los valores de la Tabla 6.
LOCAL INTERRUPT FLAG:PO	2:0	1	Polaridad de la interrupción.
LOCAL INTERRUPT FLAG:EL	2:1	1	Tipo de disparo (*trigger*) de la interrupción.
SOURCE BUS ID	4	8	Identificación del bus de origen de la interrupción.
SOURCE BUS IRQ	5	8	IRQ de la interrupción.
DESTINATION LOCAL APIC ID	6	8	Identificación de APIC local al que se dirige la interrupción. Un valor 0FFh indica que está dirigido a todos los APIC locales (a todos los procesadores).
DESTINATION LOCAL APIC LINTIN#	7	8	Número de línea de entrada LINTIN del APIC local (0 o 1) al que se conecta la interrupción.

Tabla 5. Campos de la entrada tipo asignación de interrupciones locales

Tipo de interrupción	Descripción	Comentario
0	*INT*	Interrupción vectorizada, vector proporcionado por la tabla de redirección del APIC
1	*NMI*	Interrupción no enmascarable (*nonmaskable*)
2	*SMI*	Interrupción de administración del sistema (*System Management Interrupt*)
3	*ExtINT*	Interrupción vectorizada, vector proporcionado por el PIC (8259).

Tabla 6. Tipos de interrupción

2.1.2.7. Entradas extendidas

Las entradas extendidas recogidas en la versión 1.4 de la especificación MP se refieren a información sobre direccionamiento e interconexión de buses, y no han sido utilizadas en el desarrollo de la versión actual de Minix SMP, por lo que remitimos a la especificación MP *[8]* para más información.

2.2. APIC local

La descripción de este elemento no se encuentra dentro de la especificación MP, sino dentro de los manuales de la arquitectura Intel (*Intel Architecture Software Developers Manual V3 System Programming [6]*), eso sí, dentro de un capítulo referido a la manipulación de múltiples procesadores.

El APIC local aparece por primera vez como un elemento externo en los procesadores 80486DX, y se integra dentro del procesador a partir de los modelos Pentium, sufriendo una notable modificación en su estructura y funcionamiento. En lo sucesivo haremos referencia al modelo de APIC incorporada en los procesadores Pentium y posteriores.

El APIC local es un elemento que se antepone a las líneas de control de interrupciones del procesador (*EXTINT, INTR, INTA*) y las fuentes de interrupción, de manera que sirve como puerta de paso de las interrupciones hacia el procesador. Puede recibir interrupciones procedentes del bus ICC (bus que conecta todos los APIC del sistema) o de dos fuentes externas (interrupciones locales, *LINTINT#0* y *LINTIN#1*) normalmente conectadas al PIC 8259. Estas señales de interrupción pueden o no llegar hasta el procesador dependiendo de la programación del APIC. El APIC también puede generar interrupciones que se pasan al bus ICC para que lleguen a otros APIC o incluso al mismo APIC.

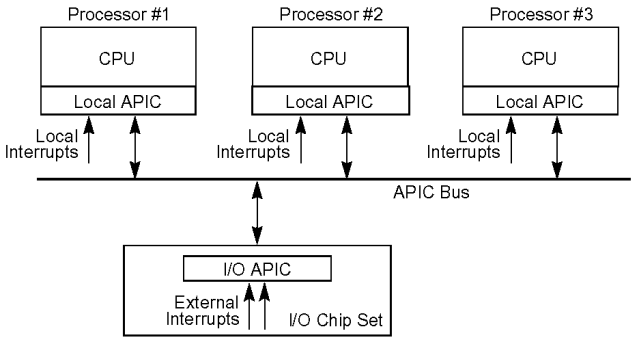


Figura 12. Esquema de conexión de los APIC del sistema

Desde el punto de vista de la programación, el APIC local se presenta como un conjunto de registros de 32 bits accesibles directamente mediante E/S mapeada en memoria a partir de una dirección que puede obtenerse de la Tabla de Configuración MP. Esta dirección, habitualmente FEE00000H, puede cambiarse para facilitar su acceso desde el sistema operativo. En total ocupa 4 kilobytes que deben estar alineados físicamente en páginas de 4 Kb. Los registros deben ser leídos y escritos un una única operación de memoria.

Dirección	**Registro**
FEE0 0000H - FEE0 0010H	Reservado
FEE0 0020H	*Local APIC ID Register*
FEE0 0030H	*Local APIC Version Register*
FEE0 0040H - FEE0 0070H	Reservado
FEE0 0080H	*Task Priority Register*
FEE0 0090H	*Arbitration Priority Register*
FEE0 00A0H	*Processor Priority Register*
FEE0 00B0H	*EOI Register*
FEE0 00C0H	Reservado
FEE0 00D0H	*Logical Destination*
FEE0 00E0H	*Destination Format Register*
FEE0 00F0H	*Spurious-Interrupt Vector Register*
FEE0 0100H - FEE0 0170H	ISR 0-255
FEE0 0180H - FEE0 01F0H	TMR 0-255
FEE0 0200H - FEE0 0270H	IRR 0-255
FEE0 0280H	*Error Status Register*
FEE0 0290H - FEE0 02F0H	Reservado
FEE0 0300H	*Interrupt Command Register 0-31*
FEE0 0310H	*Interrupt Command Register 32-63*
FEE0 0320H	*Local Vector Table (Timer)*
FEE0 0330H	Reservado
FEE0 0340H	*Performance Counter LVT*
FEE0 0350H	*Local Vector Table (LINT0)*
FEE0 0360H	*Local Vector Table (LINT1)*
FEE0 0370H	*Local Vector Table (Error)*
FEE0 0380H	*Initial Count Register for Timer*
FEE0 0390H	*Current Count Register for Timer*
FEE0 03A0H - FEE0 03D0H	Reservado
FEE0 03E0H	*Timer Divide Configuration Register*
FEE0 03F0H	Reservado

Tabla 7. Registros del APIC local

2.2.1. Identificación de destinos de interrupciones

Una de las capacidades del APIC local es la generación de interrupciones dirigidas a otros procesadores (APIC), llamadas Interrupciones Inter-Procesador (IPI, *Inter-Processor Interrupts*). El destino de una IPI puede ser un procesador, un conjunto de procesadores, o todos los procesadores. Para especificar este destino, existen dos modos: destino físico y destino lógico (*physical destination mode* y *logical destination mode*).

En el modo de destino físico, se indica el procesador de destino a través del número de su APIC (0 a 14) o se puede utilizar el valor 15 para indicar una difusión (*broadcast*) a todos los procesadores. El número de identificación del APIC se asigna por el BIOS durante el arranque, aunque puede cambiarse utilizando el registro *Local APIC ID* (Figura 13), con la condición de que no se repitan números de identificación en el sistema. El número de APIC de destino se especifica en el comando que eleva la interrupción.

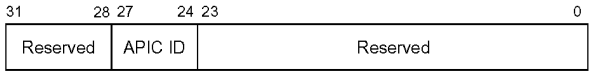


Figura 13. Registro Local APIC ID (IDR)

En el modo de destino lógico se pueden utilizar dos variantes: modelo *flat* y modelo *cluster*. Estos modelos se programan en el registro *Destination Format Register* (Figura 14), estableciendo el campo *MODEL* a 1111 (*flat model*) o a 0000 (*cluster model*).

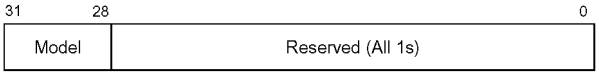


Figura 14. Registro Destination Format (DFR)

En cualquiera de los modelos (*flat* o *cluster*) el destino (MDA) se especifica en el registro *Logical Destination Register* (Figura 15), dentro del campo *LOGICAL APIC ID*, de 8 bits.

- En el modelo *flat*, la dirección de destino (MDA) se interpreta como un mapa de 8 bits que identifican otros tantos APIC. La interrupción se entrega al procesador cuyo bit correspondiente esté establecido a 1 (así sólo se pueden direccionar hasta 8 procesadores).
- En el modelo *cluster*, la dirección de destino (MDA) se divide en 2 partes de 4 bits cada una. La de mayor peso (bits 28-31) identifica un conjunto (*cluster*) y la de menor peso (bits 24-37) un mapa de bits de hasta 4 procesadores. Los bits del conjunto identifican hasta 15 conjuntos (0 a 14) o una difusión a todos los conjuntos (15). Para hacer que un procesador pertenezca a un conjunto hay que indicarlo en su registro *Logical Destination Register*, dentro del campo *Logical APIC ID*. Los 4 bits de mayor peso (bits 28-31) identifica el conjunto (*cluster*) al que pertenece el procesador, y los de menor peso (bits 24-37) numeran el procesador dentro de un determinado conjunto.

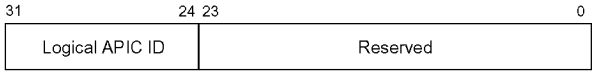


Figura 15. Registro Logical Destination (LDR)

2.2.2. Tabla de vectores locales

La tabla de vectores locales (LVT) es un conjunto de 5 registros (Figura 16) que controlan la forma en que se entregan al procesador las señales de interrupción que llegan al APIC local por sus líneas locales

(LINT), las señales de error en el *Error Status Register*, etc. Por ejemplos, las señales de interrupción que llegan al APIC por las líneas LINT#0 y LINT#1 procedentes del PIC 8259, pueden trasladarse al procesador de diferentes formas, o incluso filtrarse para que no lleguen.

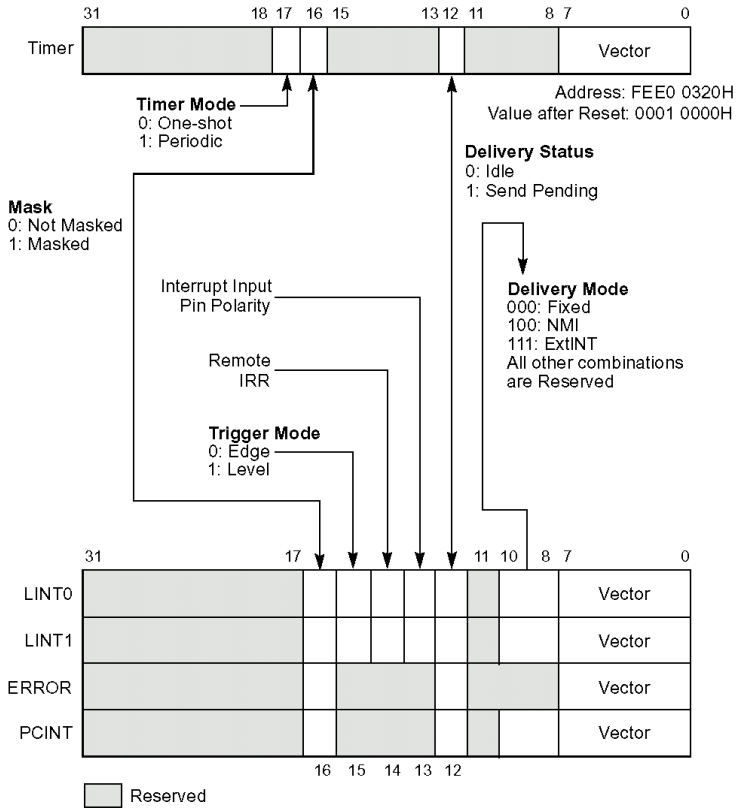


Figura 16. Tabla de vectores locales (LVT)

Los 5 registros de la LVT tienen una estructura semejante, descrita en la Tabla 8.

Campo	Significado
Vector	Número de vector de interrupción
Delivery Mode	Indica el modo en que debe trasladarse la interrupción al procesador. Los valores válidos son: 000 (*Fixed*). Traslada al procesador una interrupción del mismo tipo que la entrada de la LVT correspondiente. 100 (*NMI*). Traslada al procesador una interrupción de tipo NMI (no enmascarable) 111 (*ExtINT*). Traslada al procesador una interrupción como procedente de una fuente externa como el PIC 8259.
Delivery Status	(Sólo lectura) Indica si hay un envío pendiente (1) o si no hay actividad (0).
Interrupt Pin polarity	Indica la polaridad de la interrupción: 0 es activo a nivel alto y 1 es activo a nivel bajo.
Remote Interrupt Request Register Bit	Indica en una interrupción por nivel (*level*) con un 1 si el APIC ha aceptado una interrupción pero todavía no ha recibido el comando EOI desde el procesador.
Trigger Mode	Especifica el modo de disparo de la interrupción tipo *Fixed*: 0 por flancos (*edge*), 1 por niveles (*edge*).
Mask	Enmascara las interrupciones de manera que no pasen al procesador (1).
Timer Mode	Sólo para el registro *Timer*, indica interrupciones únicas (0) o periódicas (1).

Tabla 8. Campos de los registros de la LVT

De todos ellos, nos fijamos en los registros LINT0 y LINT1. Éstos controlan la entrega al procesador de la las señales de interrupción que llegan al APIC por las líneas LINT#0 y LINT#1 procedentes del PIC

8259. Es necesario desenmascarar estos registros para que las interrupciones de los dispositivos de E/S lleguen al procesador.

2.2.3. Interrupciones Inter-Procesador

Un procesador puede elevar una interrupción en otro u otros procesadores mediante una Interrupción Inter-Procesador (IPI, *Inter-Processor Interrupt*), a través del APIC local y el bus ICC de interconexión de los APIC. Para ello hay que invocar un comando mediante el uso del registro *Interrupt Command Register* (ICR). El ICR (Figura 17, Tabla 9) tiene un tamaño de 64 bits dividido en dos registros (*ICR-low* bits 0-31 e *ICR-high* bits 32-63). La interrupción se envía al escribir algún valor en el registro *ICR-low*.

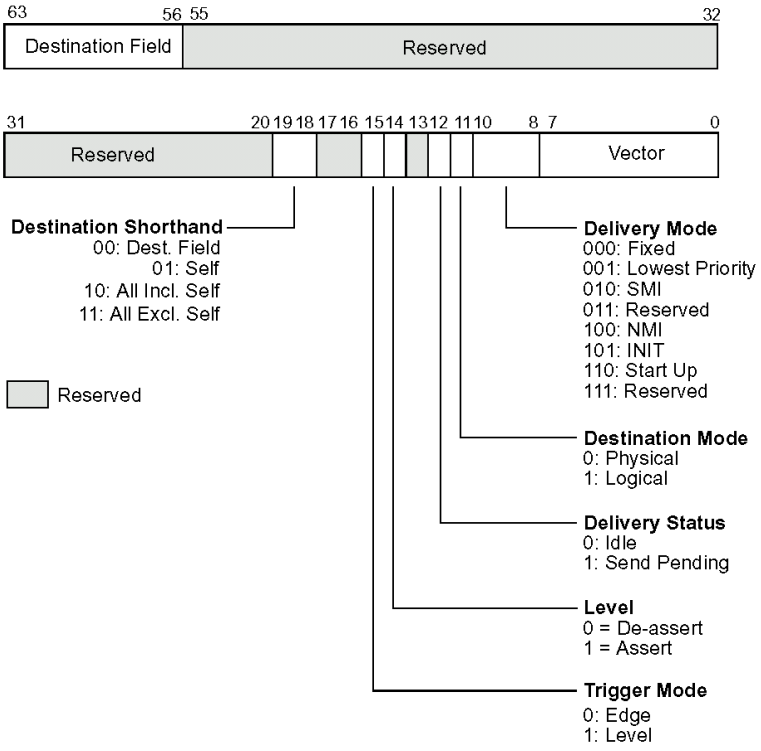


Figura 17. Registro Interrupt Command (ICR)

Campo	Significado
Vector	Indica el vector de interrupción que hay que enviar en la IPI.
Delivery Mode	Indica el modo en que debe reaccionar el APIC receptor. Los valores válidos son: 000 (*Fixed*). Entrega el vector de interrupción indicado a los APIC de destino. 001 (*Lowest Priority*). Entrega el vector de interrupción indicado al APIC que menor nivel de prioridad tenga programado en sus registros *Task Priority Register* y *Arbitration Priority Register*. 010 (*SMI*). Entrega una interrupción de administración del sistema (*System Management Interrupt*) ignorando el vector. 011 (Reservado) 100 (*NMI*). Entrega una interrupción no enmascarable (*Nonmaskable Interrupt*) ignorando el vector. 101 (*INIT*). Entrega una señal de inicio (*INIT*) ignorando el vector. Esto sitúa el APIC en estado de inicio. 110 (*INIT Level De-assert*). Entrega una señal de sincronización a todos los APIC del sistema ignorando el vector. Restablece parte de la configuración del APIC. 111 (*Start-Up*). Envía una señal de arranque al procesador indicado. La información del vector se utiliza como dirección de arranque desde la cual comenzará a ejecutar el procesador receptor (profundizaremos en este tema en secciones posteriores).
Destination Mode	Modo de direccionamiento de APIC: físico (0) o lógico (1).
Delivery Status	Indica la actividad de interrupciones enviadas anteriormente: 0 si no hay actividad, 1 si hay alguna interrupción pendiente de ser entregada (y por lo tanto hay que esperar antes de programar una nueva).
Level	1 para todos los modos (*Delivery Mode*), excepto para *INIT Level De-assert*, que debe ser 0.
Trigger Mode	0 para todos los modos (*Delivery Mode*), excepto para *INIT Level De-assert*, que debe ser 1.
Destination Shorthand	Permite indicar el destino de la interrupción de manera breve para los casos de auto-interrupción y difusión, ignorando el campo *Destination*: 00 (*destination field, no shorthand*). Utilizar el campo *Destination* para elegir el destino de la interrupción. 01 (*self*). Envía la interrupción al mismo procesador que la genera. 10 (*all including self*). Envía la interrupción a todos los procesadores. 11 (*all excluding self*). Envía la interrupción a todos los procesadores excepto al que la genera.
Destination	Indica el destino de la interrupción en el caso de que *Destination Shorthand* sea 00. En modo físico los bits de menor peso (56-59) determinan el número del APIC destino de la interrupción, y en modo lógico la interpretación de los bits 56-63 dependen del contenido de los registros DFR y LDR.

Tabla 9. Campos del registro Interrupt Command (ICR)

La llegada de una IPI al APIC local hace que se eleve en el procesador una interrupción. En el caso de una IPI de tipo *Fixed*, se elevará una interrupción según el vector indicado en el campo *Vector* de *ICR-low* que invocará el manejador de interrupción correspondiente. Dentro del manejador se debe incluir la aceptación explícita de la interrupción, ya que ésta permanecerá marcada como pendiente en la LVT hasta que se envíe el comando EOI correspondiente. Para esta operación debe escribirse un valor cualquiera (0 para futura compatibilidad) en el registro EOI (que no contiene campos).

2.2.4. Control de errores

El registro *Error Status Register* (ESR) contiene información acerca del estado del APIC, especialmente en lo referido al envío y recepción de IPI. La Figura 18 describe los bits que indican distintas situaciones de error. Para que se actualice este registro es necesario realizar una escritura aleatoria (0 para futura compatibilidad) en este registro, antes de consultar su estado.

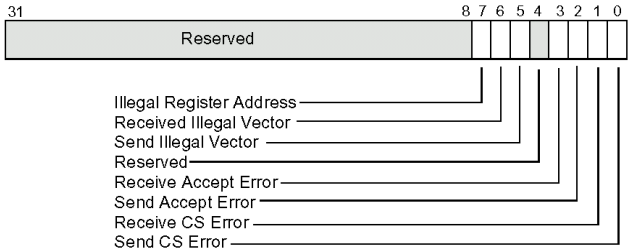


Figura 18. Registro Error Status (ICR)

2.3. I/O APIC

Las especificaciones del I/O APIC han sido tomadas del documento *82374EB/82374SB EISA System Component (ESC) [5]*, el manual de una placa madre fabricada por Intel. Se trata de una breve descripción de los registros que permiten la programación de este dispositivo. El resto de la información acerca de su funcionamiento se encuentra repartida por diversos capítulos de la especificación Intel MP *[8]* y los manuales *Intel Architecture Software Developer's Manual [6]*.

En el desarrollo actual de Minix SMP no se hace ningún uso de los comandos de programación del I/O APIC. Sin embargo sí se incluyen algunas definiciones y funciones para su probable uso futuro. Por esta razón creemos conveniente introducir un apartado de descripción de este dispositivo debido a su fuerte integración en el hardware multiprocesador.

El I/O APIC consta de un conjunto de registros de 32 bits accesibles a través de dos direcciones de memoria de 32 bits, una que selecciona un determinado registro (*I/O REGISTER SELECT REGISTER*, IORSR) y otra que lo lee o escribe (*I/O WINDOW REGISTER*, IOWR). Estas posiciones de memoria se ubican en la dirección que se puede obtener en la Tabla de Configuración MP, en las entradas de la tabla base correspondientes a I/O APIC, aunque habitualmente sólo encontraremos una unidad I/O APIC en el sistema y a partir de la dirección FEC00000H (FEC00000H para el IORSR, FEC00010H para el IOWR). Los bits 0-7 del IORSR determinan el registro del I/O APIC que deseamos acceder para lectura o escritura. Los 32 bits del IOWR contienen la información leída del registro seleccionado o permiten escribirla.

El I/O APIC contiene 3 registros de configuración (Tabla 10) más un número variable de registros de redirección de interrupciones, que determinan la forma en que las señales de interrupción de los dispositivos de E/S pasan al bus de comunicación de los APIC (ICC), y por lo tanto, la forma en que se reparten y entregan a los procesadores.

Registro	Dirección	Bits	Descripción
I/O APIC IDENTIFICATION	00H	31-28	Reservado.
		27-24	Número de identificación del I/O APIC.
		23-0	Reservado.
I/O APIC VERSION	01H	31:24	Reservado.
		23-16	Máxima entrada de redirección. Indica el número de entradas de redirección de interrupciones, es decir, el número de registros de redirección. Indica el número de registros menos 1.
		15-8	Reservado.
		7-0	Versión del APIC. 0xH para APIC externa (procesadores 80486DX), 1xH para APIC integradas (Pentium y posteriores), resto de valores reservado.
I/O APIC ARBITRATION	02H	31-28	Reservado.
		27-24	Número de identificación de arbitraje.
		23-0	Reservado.

Tabla 10. Registros de configuración del I/O APIC

Los registros de redirección de interrupciones tienen un tamaño de 64 bits, y son direccionados de forma consecutiva desde la dirección 10H. Todos tienen la misma estructura (Tabla 11). Existe un registro para cada línea de entrada de interrupciones externas que posea el I/O APIC. Su función es determinar qué tipo de interrupción se debe generar y a qué procesadores se debe entregar cuando se active cada entrada al I/O APIC. Esto permite hacer una distribución *inteligente* de las interrupciones hacia los procesadores más adecuados en cada momento.

Bits	Campo	Descripción
63-56	*Destination field*	Si el modo de destino (bit 11) es físico (0), los bits 56-59 determinan el APIC de destino. Si el modo es lógico (1), los bits 56-63 indican potencialmente un conjunto de APIC de destino.
55-17	Reservado	
16	*Interrupt Mask*	Inhabilita (1) la señal de interrupción.
15	*Trigger Mode*	Modo de disparo: 0 por flancos (*edge*), 1 por nivel (*level*).
14	*Remote IRR*	(Sólo lectura y definido sólo con disparo por nivel). Vale 1 cuando una interrupción ha sido entregada pero no se ha recibido la señal EOI correspondiente desde el APIC local.
13	*Interrupt Input Pin Polarity*	Especifica la polaridad de la interrupción: 0 activo a nivel alto, 1 activo a nivel bajo.
12	*Delivery Status*	(Sólo lectura). Vale 1 cuando una interrupción está siendo enviada y es necesario esperar antes de enviar la siguiente. Vale 0 cuando no hay actividad pendiente.
11	*Destination Mode*	Especifica el modo de destino, 0 para físico y 1 para lógico.
10-8	*Delivery Mode*	Especifica el tipo de interrupción que hay que elevar para cada entrada del I/O APIC. Los valores posibles son: 000 (*Fixed*). Entrega el vector de interrupción indicado a los APIC de destino. 001 (*Lowest Priority*). Entrega el vector de interrupción indicado al APIC que menor nivel de prioridad tenga programado en sus registros *Task Priority Register* y *Arbitration Priority Register*. 010 (*SMI*). Entrega una interrupción de administración del sistema (*System Management Interrupt*) ignorando el vector. 011 (Reservado) 100 (*NMI*). Entrega una interrupción no enmascarable (*Nonmaskable Interrupt*) ignorando el vector. 101 (*INIT*). Entrega una señal de inicio (*INIT*) ignorando el vector. Esto sitúa el APIC en estado de inicio. 110 (Reservado). 111 (*ExtINT*). Entrega el vector de interrupción indicado a los APIC de destino como si procediera de la fuente externa de interrupciones (PIC 8259).
7:0	*Interrupt Vector*	Indica el vector de interrupción que se debe entregar. Sólo son válidos los valores 10H a 0FEH.

Tabla 11. Campos de los registros de redirección de interrupciones

2.4. Instrucciones atómicas

El repertorio de instrucciones de la arquitectura Intel ofrece dos mecanismos de atomicidad en la ejecución de instrucciones máquina: instrucciones implícitamente atómicas y un prefijo *LOCK* válido para un conjunto de instrucciones no atómicas que fuerzan un acceso atómico a la memoria, bloqueando el bus durante la ejecución y forzando a que dos procesadores no puedan acceder a las mismas posiciones de memora de forma simultánea.

Entre las instrucciones implícitamente atómicas destacamos las recogidas en la Tabla 12. Además, el prefijo *LOCK* se puede anteponer a las siguientes instrucciones cuando tienen alguna referencia a memoria: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG.

Instrucción	Significado	Formato	Direccionamiento	Operación
BTC	*Bit test and complement*	BTC a, b	BTC r/m16,r16 BTC r/m32,r32 BTC r/m16,imm8 BTC r/m32,imm8	Establece CF al valor del bit *b*-ésimo de *a* y complementa el bit *b*-ésimo de *a*.
BTR	*Bit test and reset*	BTR a, b	BTR r/m16,r16 BTR r/m32,r32 BTR r/m16,imm8 BTR r/m32,imm8	Establece CF al valor del bit *b*-ésimo de *a* y pone a 0 el bit *b*-ésimo de *a*.
BTS	*Bit test and set*	BTS a, b	BTS r/m16,r16 BTS r/m32,r32 BTS r/m16,imm8 BTS r/m32,imm8	Establece CF al valor del bit *b*-ésimo de *a* y pone a 1 el bit *b*-ésimo de *a*.

Tabla 12. Instrucciones atómicas

2.5. El sistema de cache

Como hemos visto en la introducción del capítulo, la especificación multiprocesador Intel exige que la coherencia de cache se realice de forma transparente al software, por lo que no debemos preocuparnos de ello. Sólo existe un detalle que sí debemos tener en cuenta: cada procesador puede habilitar o inhabilitar su cache interna de forma individual. Si un procesador no habilita su cache, la coherencia del sistema de memoria no está asegurado para ese procesador.

La configuración básica de la cache de un procesador se realiza a través de los campos `NW` y `CD` del registro de configuración `cr0` del procesador (bits 29 y 30 respectivamente). El máximo rendimiento *[6]* se logra cuando ambos bits están a cero (la cache está habilitada y funcionando –con la mejora de rendimiento que ello proporciona– y la coherencia se asegura), y el sistema se inhabilita (cache desactivada y coherencia no asegurada) cuando ambos están a 1.

2.6. Procedimiento de arranque del segundo procesador

El procedimiento a seguir para realizar el arranque de los procesadores de aplicación (AP, *Application Processor*) se describe en la especificación MP *[8]*. Esencialmente consiste en el envío de una secuencia de interrupciones (IPI) desde el BSP (*Bootstrap Processor*) que hacen que el AP comience a ejecutar desde una determinada dirección de memoria.

Como ya hemos introducido en las secciones anteriores, el BIOS debe encargarse de configurar el sistema durante la secuencia de arranque para que se mantenga la compatibilidad con un sistema monoprocesador. Esto incluye que sólo uno de los procesadores (el BSP) debe funcionar inicialmente, estando todos los demás detenidos.

El acceso a los dispositivos relacionados con el multiprocesamiento (p.e. el APIC local, el I/O APIC, etc.) requieren que el BSP esté funcionando en modo protegido, teniendo acceso a la memoria superior. Una vez leída la configuración del hardware desde la Tabla de Configuración MP comenzará el procedimiento de arranque de los distintos AP, que se debe realizar uno por uno.

La documentación de Intel no es muy clara respecto al procedimiento de arranque. Facilita un "algoritmo universal" que funciona, pero no explica por qué debe ser así. Este algoritmo debe ser válido para todo tipo de APIC local (integradas o no):

Algoritmo universal de arranque de AP

```
BSP envía IPI INIT a AP
Esperar 10µs
```

```
Si APIC es integrada {
       BSP envía IPI STARTUP a AP
       Esperar 200µs
       BSP envía IPI STARTUP a AP
       Esperar 200µs
}
Comprobar arranque del AP
```

Puesto que en la versión actual de Minix SMP sólo se tiene en cuenta la arquitectura Pentium y posterior, el algoritmo se simplifica a:

Algoritmo de arranque de AP para Pentium

```
BSP envía IPI INIT a AP
Esperar 10µs
BSP envía IPI STARTUP a AP
Esperar 200µs
BSP envía IPI STARTUP a AP
Esperar 200µs
Comprobar arranque del AP
```

La documentación impone dos restricciones respecto al uso de estas IPI:

1. Una IPI STARUP sólo se puede enviar después de haber enviado una IPI INIT.
2. Después de una IPI INIT debe enviarse una IPI INIT DE-ASSERT.

Según esto, la segunda IPI STARTUP únicamente sirve para asegurar un poco más la entrega de la interrupción (ya que podría fallar). Entonces podemos simplificar el algoritmo a:

Algoritmo modificado de arranque de AP para Pentium

```
BSP envía IPI INIT a AP
Esperar 10µs
BSP envía IPI INIT DE-ASSERT a AP
Esperar 10µs
BSP envía IPI STARTUP a AP
Esperar 200µs
Comprobar arranque del AP
```

En los procesadores Pentium, la recepción de la secuencia de IPI INIT-INIT DE-ASSERT provoca que el procesador receptor se reinicie. Se consulta el código almacenado en la dirección 0Fh de la RAM CMOS. Este código tiene diferentes significados. Un valor 0Ah indica que se realice un arranque caliente (*warm reset*), consistente en un salto a la dirección indicada en el *warm reset vector*, una dirección de memoria almacenada en la dirección 40:67. Por defecto, el *warm reset vector* apunta a la secuencia de arranque caliente del BIOS, pero modificando esta dirección podemos hacer que el procesador salte a cualquier dirección.

Esto debería ser suficiente para iniciar el AP, sin necesidad de la IPI STARTUP. En los procesadores 80486DX con el APIC local no integrado, STARTUP no está definido, por tanto no se puede utilizar, y la secuencia de INIT debería ser suficiente. Este extremo no ha podido ser verificado puesto que no se dispone de dicho hardware para comprobarlo. La realidad es que con procesadores Pentium no es suficiente.

Por lo tanto, después de la secuencia de INIT, se envía una IPI de tipo *STARTUP*. Esta interrupción provoca que el procesador receptor comience a ejecutar código en modo real a partir de la dirección 000VV000H (CS:IP VV00:0000h), donde *VV* es el vector de interrupción enviado en la *IPI STARTUP*. El

sistema operativo debe encargarse de alojar el fragmento de código de arranque del AP en una dirección adecuada y enviar el vector correspondiente, estando los vectores A0h-BFh reservados. Así, la manera correcta y detallada de realizar el arranque consiste en los siguientes pasos:

Algoritmo final de arranque de AP

```
Alojar el código inicial para el AP en una dirección VV00:0000h
Reprogramar el warm reset vector apuntando a la dirección VV00:0000h
Asegurar el código 0Ah en la dirección 0Fh de la RAM CMOS
Enviar IPI INIT a AP
Esperar 10µs
Enviar IPI INIT DE-ASSERT a AP
Esperar 10µs
BSP envía IPI STARTUP a AP con el vector VV
Esperar 200µs
Comprobar arranque del AP
```

Con esta secuencia, el AP comenzará a ejecutar en modo real a partir de la dirección VV00:0000h. En esta dirección previamente habremos colocado el código en modo real que debe ejecutar el AP cuando arranque. La secuencia de pasos que debe realizar cada AP es la siguiente:

Código inicial del AP

```
Pasar a modo protegido
Establecer las estructuras de memoria
Habilitar cache
Establecer una bandera a un valor determinado para informar al BSP del arranque del AP
Configurar APIC local
```

Obsérvese la necesidad de habilitar la cache, ya que por defecto el procesador se inicia con la cache inhabilitada. También es necesario programar el APIC local para habilitar la llegada de interrupciones al AP de manera que el funcionamiento sea simétrico.

2.7. Descripción del hardware utilizado para el desarrollo

El desarrollo y pruebas de Minix SMP se ha realizado utilizando un sistema multiprocesador real compatible con la especificación Intel MP 1.4. Se trata de un PC con dos procesadores *Intel Pentium III MMX* a 500 MHz. (modelo 673) *[7]* montados sobre una placa *Gigabyte GA-6BXD r1.6* que utiliza un chipset *Intel 440BX AGP* (BIOS v.2.5). El BIOS de la placa es de tipo *Award Modular BIOS v4.51P6*.

Los datos que se listan a continuación son los recogidos de la Tabla de Configuración MP por el propio Minix SMP durante el arranque multiprocesador.

La estructura FPS se encuentra en la dirección F5B30H, e indica que se trata de una configuración no predeterminada (*custom*) basada en una especificación Intel MP 1.1[1]. Por último indica que la tabla de configuración MP se encuentra en la dirección F1400H.

La tabla de configuración MP confirma estar basada en la especificación Intel MP 1.1. El registro del fabricante viene firmado como "OEM00000" y el del producto como "PROD00000000". Indica que la dirección de memoria de acceso al APIC local está establecida en FEE000000H. Por último, indica que la

[1] En el *setup* de configuración del BIOS se puede configurar la versión de la especificación MP que se informa al sistema operativo (*MP version to report to OS*) entre 1.1 y 1.4. Por lo tanto consideramos que el computador es compatible con la versión 1.4. Sin embargo, la modificación de este valor, sorprendentemente, no afecta al contenido de la Tabla de Configuración MP.

tabla base contiene 28 entradas, 2 de tipo CPU, 3 de tipo bus, 1 de tipo I/O APIC, 20 de tipo interrupción de E/S, y 2 de tipo interrupción local. La descripción de estas entradas se recoge en la Tabla 13. Hay que destacar, que el número de entradas de tipo interrupción E/S depende de la cantidad y tipo de dispositivos conectados en los buses del sistema (tarjetas). Por ejemplo, si cambiamos una tarjeta de red PCI por una ISA, o simplemente la quitamos, desaparecerá una asignación de interrupción E/S debido al que el BIOS *plug-and-play* liberará el recurso correspondiente.

CPU habilitada tipo 673 (Pentium III) con APIC local versión 1.7 (integrada) identificada con el número 0, actuando de BSP.
CPU habilitada tipo 673 (Pentium III) con APIC local versión 1.7 (integrada) identificada con el número 1, actuando de AP.
Bus tipo *PCI* identificado con el número 0
Bus tipo *PCI* identificado con el número 1
Bus tipo *ISA* identificado con el número 2
I/O APIC versión 1.7 identificada con el número 2 y habilitada en la dirección de memoria FEC00000H.
Interrupción de E/S tipo *ExtINT* procedente del bus 2 IRQ#0 conectada al I/O APIC 2 por la línea INTN#0
Interrupción de E/S tipo *INT* procedente del bus 2 IRQ#1 conectada al I/O APIC 2 por la línea INTN#1
Interrupción de E/S tipo *INT* procedente del bus 2 IRQ#0 conectada al I/O APIC 2 por la línea INTN#2
Interrupción de E/S tipo *INT* procedente del bus 2 IRQ#3 conectada al I/O APIC 2 por la línea INTN#3
Interrupción de E/S tipo *INT* procedente del bus 2 IRQ#4 conectada al I/O APIC 2 por la línea INTN#4
Interrupción de E/S tipo *INT* procedente del bus 2 IRQ#5 conectada al I/O APIC 2 por la línea INTN#5
Interrupción de E/S tipo *INT* procedente del bus 2 IRQ#6 conectada al I/O APIC 2 por la línea INTN#6
Interrupción de E/S tipo *INT* procedente del bus 2 IRQ#7 conectada al I/O APIC 2 por la línea INTN#7
Interrupción de E/S tipo *INT* procedente del bus 2 IRQ#8 conectada al I/O APIC 2 por la línea INTN#8
Interrupción de E/S tipo *INT* procedente del bus 2 IRQ#9 conectada al I/O APIC 2 por la línea INTN#9
Interrupción de E/S tipo *INT* procedente del bus 2 IRQ#10 conectada al I/O APIC 2 por la línea INTN#10
Interrupción de E/S tipo *INT* procedente del bus 2 IRQ#11 conectada al I/O APIC 2 por la línea INTN#11
Interrupción de E/S tipo *INT* procedente del bus 2 IRQ#12 conectada al I/O APIC 2 por la línea INTN#12
Interrupción de E/S tipo *INT* procedente del bus 2 IRQ#13 conectada al I/O APIC 2 por la línea INTN#13
Interrupción de E/S tipo *INT* procedente del bus 2 IRQ#14 conectada al I/O APIC 2 por la línea INTN#14
Interrupción de E/S tipo *INT* procedente del bus 2 IRQ#15 conectada al I/O APIC 2 por la línea INTN#15
Interrupción de E/S tipo *INT* procedente del bus 0 IRQ#28 conectada al I/O APIC 2 por la línea INTN#19
Interrupción de E/S tipo *INT* procedente del bus 0 IRQ#36 conectada al I/O APIC 2 por la línea INTN#17
Interrupción de E/S tipo *INT* procedente del bus 0 IRQ#44 conectada al I/O APIC 2 por la línea INTN#19
Interrupción de E/S tipo *SMI* procedente del bus 2 IRQ#0conectada al I/O APIC 2 por la línea INTN#23
Interrupción local tipo *ExtINT* procedente del bus 0 IRQ#0 conectada a todos los APIC locales por la línea INTIN#0
Interrupción local tipo *NMI* procedente del bus 0 IRQ#0 conectada a todos los APIC locales por la línea INTIN#1

Tabla 13. Descripción de las entradas de la tabla base de configuración MP

En la Figura 19 se muestran algunas fotografía del computador utilizado.

Figura 19. Imágenes del computador utilizado para el desarrollo y prueba de Minix SMP

3 Arquitectura de Minix 2.0

Antes de describir Minix SMP es conveniente que repasemos los aspectos de la estructura original de Minix relacionados con las ampliaciones y modificaciones llevadas a cabo sobre ella. Comenzaremos por los aspectos generales de la arquitectura, para centrarnos después en algunas cuestiones más concretas, como el planificador, el flujo de entradas y salidas al núcleo, y el procedimiento de arranque y detención del sistema, ya que son los puntos estratégicos que afectan a la extensión multiprocesador.

3.1. Organización general

Minix es un sistema operativo microkernel. La gestión de procesos tiene lugar según tres niveles de prioridad: las tareas de E/S (*TASK*), servidores (*SERVER*) y procesos de usuario (*USER*). Dentro del núcleo sólo queda la gestión de la comunicación entre tareas, la planificación, la gestión de interrupciones, y, por supuesto, todo lo referido al arranque y detención del sistema. La Figura 20 muestra esta estructura. Los procesos del sistema son los servidores y las tareas. Los procesos de usuario interaccionan con ellos a través del núcleo mediante paso de mensajes. Sólo las tareas tienen acceso directo al hardware, ya que son las encargadas de la manipulación a bajo nivel de los dispositivos.

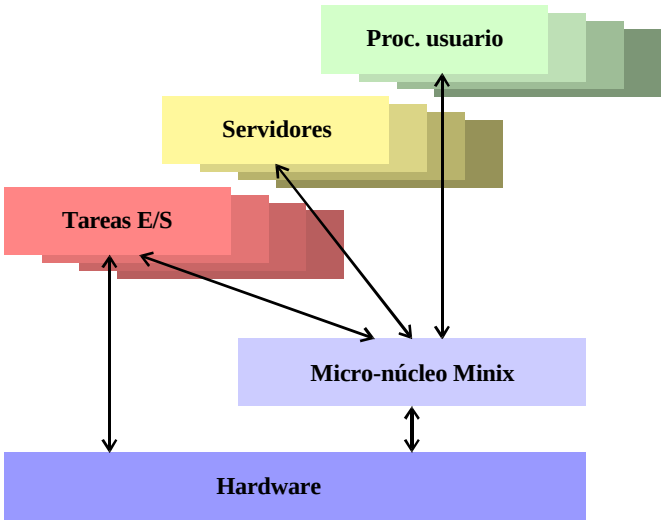


Figura 20. Arquitectura microkernel de Minix

El paso de mensaje entre tareas se hace mediante el paradigma de cita. Si un proceso intenta enviar un mensaje a otro proceso que no lo está esperando, el primero quedará detenido hasta que el segundo alcance la cita. Igualmente, si un proceso solicita la recepción de un mensaje que todavía no ha sido enviado, quedará detenido hasta que el mensaje esté disponible.

El núcleo y las tareas de E/S se compilan en un mismo espacio de direccionamiento. Sin embargo las tareas no acceden a información compartida, sino que se comunican mediante paso de mensajes. Los servidores se compilan cada uno de forma independiente. Finalmente núcleo, tareas de E/S, servidores e INIT se unen para formar la imagen del sistema operativo, que se carga en memoria durante el arranque del sistema.

La versión 2.0.0 *[9]* soporta los procesadores Intel 80386 y superiores para, entre otras cosas, funcionar en modo protegido. Esto es importante, ya que es una condición de partida para poder acceder a todos los dispositivos relacionados con el multiprocesador, como el sistema de APIC.

Los procesadores Intel en modo protegido definen hasta 4 niveles de privilegio para las tareas: nivel de supervisor, nivel de usuario y otros dos niveles intermedios. El núcleo trabaja en modo supervisor y los servidores y procesos de usuario en modo usuario. Las tareas de E/S funcionan en uno de los modos intermedios, con nivel de privilegio suficiente como para acceder al hardware, pero con algunas capacidades restringidas.

El proceso ocioso (IDLE) se organiza como una tarea de E/S, aunque con algunas características especiales, como veremos más adelante.

3.2. Planificación

El planificador de Minix es muy sencillo. Podemos entenderlo como un objeto que se manipula mediante un conjunto de métodos. Su implementación consiste en tres colas de procesos preparados, correspondientes a los tres tipos de procesos antes citados. Las tareas de E/S son las más prioritarias, seguidas de los servidores y finalmente de los procesos de usuario. Cuando no hay procesos dispuestos en ninguna de estas colas, se ejecuta la tarea de E/S especial IDLE. A diferencia de otros sistemas, en Minix IDLE es una tarea de E/S y no parte del núcleo. Esto, que es un inconveniente para la explotación completa

de esta tarea[2] (recordemos que Minix se creó para aprender, no para explotado comercialmente), será una importante ventaja que facilitará el diseño multiprocesador: cuando se ejecuta IDLE el procesador corre en un proceso, por lo que no retiene recursos del núcleo que puedan ser necesarios cuando ocurra alguna interrupción. A pesar de definirse como una tarea de E/S más, tiene un tratamiento especial y diferente en todo momento. Por ejemplo, no se planifica junto con el resto de tareas de E/S. Por esta razón, en la Figura 21, que representa el esquema de colas de tareas, IDLE se ha representado como una tarea especial, que sólo se ejecuta cuando no hay otros procesos listos.

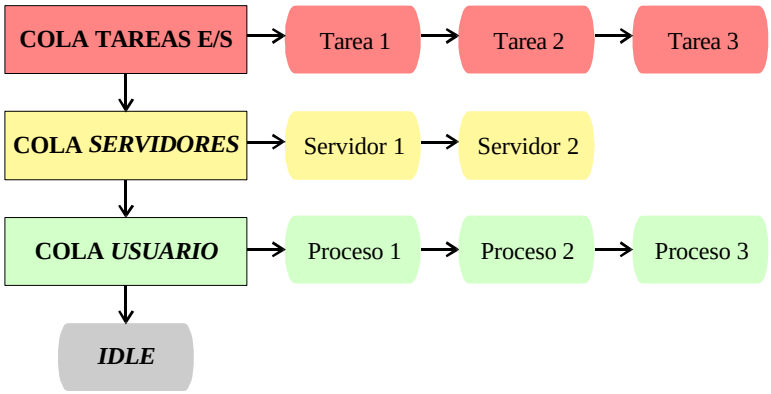


Figura 21. Gestión de colas de procesos en Minix

La gestión de las colas se hace mediante un conjunto de métodos, que podemos encontrar definidos en el fichero `proc.c`:

- **`ready`**: añade un proceso al final a la cola de preparados que le corresponda.
- **`unready`**: elimina un proceso de la cola de preparados en que se encuentra.
- **`pick_proc`**: selecciona el proceso más prioritario para ser ejecutado. Primero busca en la cola de tareas de E/S, si no encuentra procesos, busca en la de servidores, luego en la de proceso de usuario, y si no ha encontrado ningún proceso en las colas, entonces selecciona la tarea IDLE. El proceso seleccionado se almacena en la variable global `proc_ptr`. Además, si es un proceso de usuario, lo apunta en `bill_ptr` para que se le contabilice el tiempo de CPU que consume.
- **`sched`**: realiza una planificación de la cola de procesos de usuario, de manera que el primer proceso de la cola pasa a ser el último (algoritmo *round-robin*).

`ready` y `unready` se emplean cuando algún proceso pasa a estado de preparado o queda bloqueado, respectivamente. Entonces `pick_proc` se invoca si puede haber un nuevo proceso preparado y más prioritario que el actual. `sched` sólo se invoca cuando un proceso de usuario ha agotado su cuantum. Un proceso pasa a estado de preparado cuando completa una cita, y queda bloqueado cuando espera que una cita se complete. Esto sólo ocurre en el envío y recepción de mensajes entre tareas, que se realiza exclusivamente mediante las funciones `mini_send` y `mini_rec`.

- **`mini_send`**: intenta enviar un mensaje. Si el receptor no lo está esperando, elimina de la cola de preparados (`unready`) al remitente y anota en éste el mensaje pendiente de ser enviado. Si el receptor estaba detenido esperando el menaje, se lo entrega y lo pone en la cola de preparados (`ready`).
- **`mini_rec`**: intenta recibir un mensaje. Si el emisor no lo ha enviado aún, se elimina de la cola de preparados (`unready`) al que intenta recibir. Si, por el contrario, el emisor estaba detenido

[2] IDLE, por ser una tarea E/S, ejecuta en un nivel de privilegio inferior al núcleo y no puede invocar algunas instrucciones privilegiadas como HLT, que detiene el procesador y permite, por ejemplo, que se activen los procedimientos de ahorro de energía.

esperando enviar el mensaje, toma el mensaje anotado en éste y lo pone en la cola de preparados (ready).

3.3. Entradas y salidas al núcleo

La invocación de las funciones de planificación se hace en las entradas al núcleo, que son las interrupciones y las llamadas al sistema: las rutinas **interrupt** y **sys_call** respectivamente.

- Los manejadores de interrupción invocan la rutina interrupt, la cual se limita a enviar un mensaje a la tarea de E/S correspondiente al dispositivo que interrumpió, de manera que la tarea de E/S se activa y realiza la atención de la interrupción.
- Las llamadas al sistema ejecutan la rutina sys_call. Sólo existen 3 llamadas al sistema: send, receive y send_rec, que se traducen en un único trap que invoca a sys_call.

El núcleo de Minix es reentrante. Una reentrada en el núcleo interrumpirá la ejecución alguno de los métodos del planificador, los cuales deben ejecutarse de forma atómica. Para asegurar la atomicidad sin necesidad de bloquear las interrupciones, Minix utiliza una *cola de interrupciones retenidas*. En lugar de activar la tarea de E/S correspondiente al dispositivo que interrumpe, interrupt anota la interrupción en una cola, de manera que cuando se finaliza la entrada al núcleo se recorre la cola y entonces se activan las tareas asociadas a los dispositivos.

Además del núcleo, algunas tareas de E/S, como la tarea del reloj, invocan los métodos del planificador. Mientras se ejecutan estos métodos en el contexto de una tarea de E/S, puede ocurrir una interrupción que requiera acceso al planificador. Por esta razón existe una variante de los métodos del planificador, utilizada por las tareas de E/S, con el mismo nombre, pero precedido de "lock_" (lock_ready, lock_unready, etc.). Esta variante invoca a la original, pero previamente establece una bandera llamada **switching** que hace que cualquier entrada al núcleo por interrupción sea considerada como reentrada, retenida y encolada para ser atendida más adelante.

Los manejadores de interrupción de dispositivos realizan el cambio de contexto invocando la rutina save, reconocen la interrupción enviando los comandos adecuados al controlador de interrupciones, y acaban invocando interrupt. De forma simétrica, el único trap utilizado para las llamadas al sistema realiza el cambio de contexto en la rutina s_call (que es una simplificación de save) y acaba invocando sys_call.

El cambio de contexto es ligeramente diferente para el caso de una entrada y una reentrada (por eso s_call, que no puede ejecutarse por reentrada, es una simplificación de save). En el caso de un entrada es necesario, además de la salvaguarda de registros generales, conmutar a la pila del núcleo, mientras que en una reentrada este cambio ya se habrá realizado en la primera entrada.

Para diferenciar entradas y reentradas existe una variable **k_reenter** que se incrementa en cada entrada o reentrada (tanto por save, como por s_call) desde el valor –1, y se decrementa al salir. Si k_reenter tiene un valor mayor de 0 después de incrementarse, entonces se trata de una reentrada y no es necesario conmutar a la pila del núcleo. Además, sólo habrá que volver a conmutar a la pila de alguna tarea en el caso de que estemos en la última salida. La pila del núcleo es un espacio de 1024 bytes en el que se apilan las llamadas a procedimiento que se realizan durante la ejecución del núcleo.

La salida del núcleo se hace en la rutina **restart**. Esta rutina tiene en realidad dos puntos de entrada, restart y restart1, que se invocan en función de si se trataba de una entrada o una reentrada. restart1 es el despachador del sistema: decrementa el valor de k_reenter y restaura los registros de la última entrada o reentrada, retornando al punto de ejecución anterior. Por su parte, restart coloca los registros del proceso indicado por proc_ptr (seleccionado por la rutina pick_proc del planificador) para ser restaurados y continúa por restart1, haciendo que se despache ese proceso. restart, antes que nada, comprueba que la

cola de interrupciones retenidas está vacía. Si no lo está, antes de continuar repite las interrupciones que quedaron anotadas y retenidas en reentradas anteriores, y que no pudieron ser atendidas.

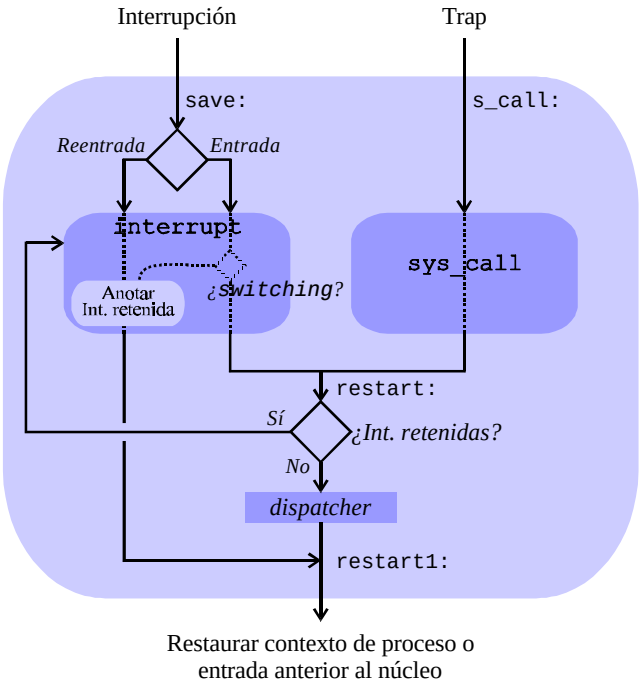


Figura 22. Flujo de entradas y reentradas al núcleo de Minix

La Figura 22 ilustra el esquema de entradas al núcleo, que podemos resumir diciendo que se produce de dos formas:

- Mediante una interrupción hardware: la elevación de la interrupción hace saltar al procesador al manejador correspondiente. Dentro del manejador, se invoca a la rutina `save`, que actualiza y comprueba el contador de reentradas y en caso necesario conmuta a la pila del núcleo. Además, establece si se debe retornar a `restart` o `restart1`. El manejador, a continuación, reconoce la interrupción, e invoca a `interrupt` para que se envíe el mensaje adecuado a la tarea de E/S correspondiente. Finalmente retorna a `restart` o `restart1`, dependiendo de lo establecido en `save`. Dentro de `interrupt`, si se detecta que nos encontramos en una reentrada o que se estaba accediendo a alguna estructura protegida, se anota la interrupción retenida, en lugar en enviar el mensaje adecuado.
- Mediante una llamada al sistema: la ejecución de la instrucción de trap (`int`) produce una salto a la rutina `s_call`. En ella se actualiza el contador de reentradas, se realiza un cambio incondicional a la pila del núcleo (no puede ser una reentrada), se invoca a la rutina `sys_call` (que accede al planificador), y se continúa por `restart`. Dentro de `sys_call` se envían los mensajes a las tareas de E/S y servidores adecuados para que se realice la atención del servicio solicitado.

Las rutinas `save`, `s_call` y `restart` las encontramos en el fichero fuente `mpx386.s` (versión para modo protegido, el único que se tiene en cuenta en este trabajo).

3.4. Arranque y detención del sistema

Del mecanismo utilizado por Minix durante su arranque sólo nos interesa ahora la parte final. Una vez el monitor ha cargado en memoria la imagen de Minix y ha comenzado a ejecutarla, el flujo de ejecución llega hasta la rutina `main` (en `main.c`). Desde esta rutina se invocan los procedimientos que establecen la

estructura de memoria, inician la tabla de procesos, activan las tareas de E/S, servidores y procesos de inicio (`INIT`), y finalmente se invoca al despachador para que comience a ejecutar el primer proceso preparado.

La rutina de detención del sistema se llama `wreboot`, y la encontramos con el código fuente de la tarea del teclado (`keyboard.c`), ya que se invoca cuando se realiza la combinación de teclas `CTRL+ALT+SUPR`. También se invoca desde cualquiera de los puntos del núcleo que implican la detención o reinicio del sistema: llamadas al sistema de apagado o la rutina de detención por error crítico en el núcleo (`panic`).

La función de `wreboot` es invocar las secuencias de detención de las distintas tareas, volver a modo real y retornar al monitor de carga del sistema.

4 Principios de diseño

El principio de la multicomputación simétrica establece que ninguno de los procesadores del sistema debe ser diferente de los demás. El reparto de tareas entre procesadores debe ser distribuido, de manera que ningún procesador se dedique a asignar trabajo a los demás *[8]*.

La idea aplicada en Minix SMP es que cada procesador trabaje en el sistema como si estuviera solo, compitiendo con los demás por realizar su trabajo: ejecutar procesos. Así, únicamente nos preocuparemos, primero, de que dos procesadores no intenten ejecutar el mismo proceso al mismo tiempo (para lo cual modificaremos sensiblemente el planificador), y segundo, de que los procesadores no ejecuten código crítico a la vez.

Analizamos en esta sección las directrices generales seguidas para el diseño de Minix SMP. Para cumplir los objetivos propuestos, la implementación del núcleo multiprocesador se ha realizado procurando siempre la mínima modificación de la estructura y organización del núcleo original de Minix, descrito en el capítulo 3. Tomando como punto de partida la versión 2.0.0 de Minix *[9]*, primera versión que es conforme POSIX 1003.1a, se han realizado pequeñas modificaciones sobre el código fuente, aunque también ha sido necesario escribir código nuevo: la interfaz con el hardware SMP. La Figura 23 representa la ampliación realizada: se ha ampliado el núcleo y se ha modificado una pequeña parte del mismo. Algunas tareas también han sufrido alguna leve modificación debido a su fuerte relación con el hardware.

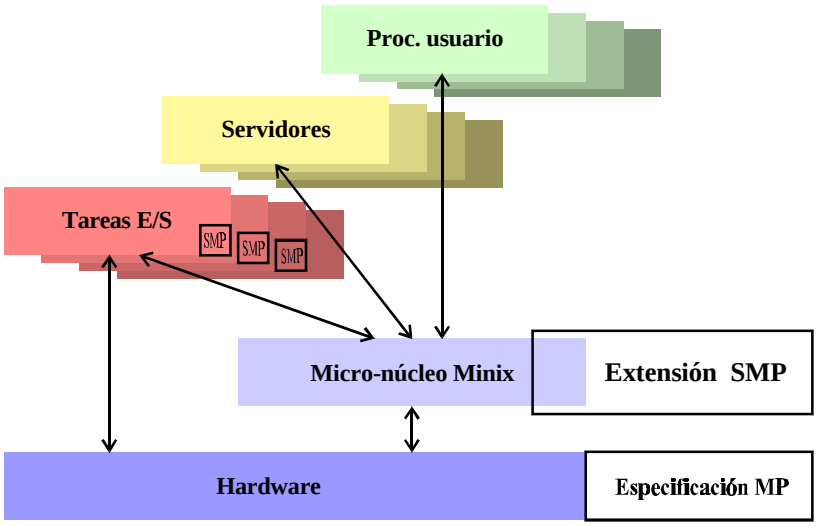


Figura 23. Alcance de la extensión SMP

La especificación 1.4 MP de Intel está diseñada para ser escalable en número de procesadores. Por esta razón, todas las modificaciones sobre el código del núcleo se han realizado de manera condicional sobre dos parámetros de configuración: la habilitación del sistema MP (`ENABLE_MP`), y el número de procesadores (`MP_NR_CPUS`). Así, modificando estos parámetros podemos compilar el núcleo ordinario Minix o un núcleo con soporte para multiproceso simétrico con cualquier número de procesadores.

En cuanto al nuevo código, se trata del necesario para cubrir las siguientes funciones (correspondientes a la interfaz MP):

1. Manipulación de la Tabla de Configuración MP.
2. Manipulación del sistema de APIC.
3. Arranque y detención del modo multiprocesador
4. Primitivas de sincronización global entre procesadores.

En cuanto a las modificaciones realizadas tanto en el núcleo como en las tareas de E/S, persiguen la solución a dos problemas principales:

1. La necesidad de replicar algunos recursos para los distintos procesadores: pila del núcleo, variables globales, etc.
2. La necesidad de sincronización entre procesadores en dos puntos críticos: la entrada al núcleo y el planificador.

A continuación describiremos el tipo de construcciones utilizadas para llevar a cabo la sincronización entre procesadores en las secciones críticas identificadas: el núcleo y el planificador, dedicándose los dos siguientes apartados a analizar los principios de diseño aplicados en estas secciones. Terminaremos el capítulo introduciendo la replicación de estructuras, tema que se estudiará en profundidad en el próximo capítulo, dedicado a los detalles de la implementación.

4.1. Primitivas de sincronización

La protección de secciones críticas frente al acceso multiprocesador se va a realizar mediante el uso de las primitivas de sincronización basadas en cerrojos (*spinlock*) *[1,18]* que se han construido: una primitiva LOCK al inicio de una sección crítica (ver Figura 24) cierra (bloquea) un cerrojo e impide que otro flujo de ejecución entre en la sección (quedaría detenido en una espera activa, sin realizar ningún trabajo útil). Otra primitiva UNLOCK al final de la sección libera el cerrojo, dando paso a otro flujo que espere en LOCK. Además existirá otra primitiva IS_LOCKED que permite conocer el estado del cerrojo.

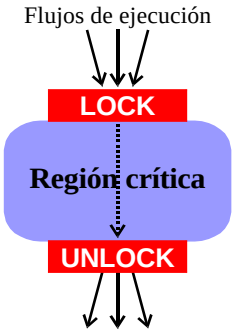


Figura 24. Esquema de protección de regiones críticas mediante cerrojos

Estas primitivas son globales al conjunto de procesadores puesto que trabajan en memoria compartida, y no ofrecen ninguna protección frente a interbloqueo. A continuación se muestra el pseudocódigo que expresa la semántica de estas primitivas (obviando la necesaria atomicidad de las operaciones):

```
LOCK (Cerrojo) {
   MIENTRAS (Cerrojo==Cerrado) Esperar;
   Cerrojo=Cerrado;
}

UNLOCK (Cerrojo) {
   Cerrojo=Abierto;
}

IS_LOCKED (Cerrojo) {
   RETORNAR (Cerrojo==Cerrado);
}
```

4.2. Protección del núcleo

Se han identificado dos secciones críticas en el núcleo de Minix que deben ser protegidas en un entorno multiprocesador. La primera de ellas es el núcleo completo. Dentro del código del núcleo se realizan tres operaciones que necesitan completarse de forma atómica:

1. La programación del controlador de interrupciones.
2. La manipulación de la cola de interrupciones retenidas.
3. La ejecución de los métodos del planificador.

Como hemos visto, Minix soporta reentrancia en el núcleo original, que puede producirse por la llegada de una interrupción. El núcleo original se encuentra, por lo tanto, protegido contra los efectos de las reentradas en un entorno monoprocesador. Esta protección, consistente en asegurar la atomicidad de las operaciones, se logra mediante la inhabilitación de interrupciones para el acceso al controlador de interrupciones y la manipulación de la cola de interrupciones retenidas, y mediante el uso de `switching` y la retención de interrupciones para el caso de los métodos del planificador.

En un entorno multiprocesador, la inhabilitación de interrupciones se produce a nivel de procesador, es decir, que un procesador sólo puede habilitar o inhabilitar sus propias interrupciones, y no las de los demás. Por lo tanto, la atomicidad protegida por la inhabilitación de interrupciones ya no es segura. La solución a este problema es utilizar las primitivas de sincronización globales descritas en el apartado anterior: debemos evitar que dos procesadores ejecuten una sección crítica al mismo tiempo. Para ello podemos utilizar dos aproximaciones:

- Solución A: Proteger regiones críticas grandes con cerrojos de granularidad gruesa.
- Solución B: : Proteger regiones críticas pequeñas con cerrojos de granularidad fina.

Como primera aproximación (A, granularidad gruesa), podemos considerar el núcleo completo como una región crítica, protegiéndolo con un único cerrojo: una primitiva `LOCK` en la entrada del núcleo (tanto por interrupciones como por traps) y una `UNLOCK` a la salida. Pero si colocamos un cerrojo a la entrada del núcleo, un intento de reentrada encontraría el cerrojo cerrado, lo que acabaría en un interbloqueo. El núcleo Minix se encuentra correctamente protegido de las reentradas que pueden producirse por parte del mismo procesador que está ejecutando código dentro de él. Así, únicamente debemos protegernos de las entradas que puedan ocurrir por parte de otros procesadores. Por lo tanto colocaremos el cerrojo del núcleo de manera que únicamente se intente el bloqueo (`LOCK`) cuando se produce la primera entrada (y no una reentrada). El primer procesador que intente entrar en el núcleo cerrará el paso a cualquier otro procesador que lo intente después, pero no a las posibles reentradas por interrupciones que se eleven en el mismo procesador.

De la misma forma, la salida de la última entrada anidada por parte de un procesador, provoca la liberación del cerrojo, dando paso a otro de los procesadores que esperan activamente. La Figura 25 ilustra este esquema.

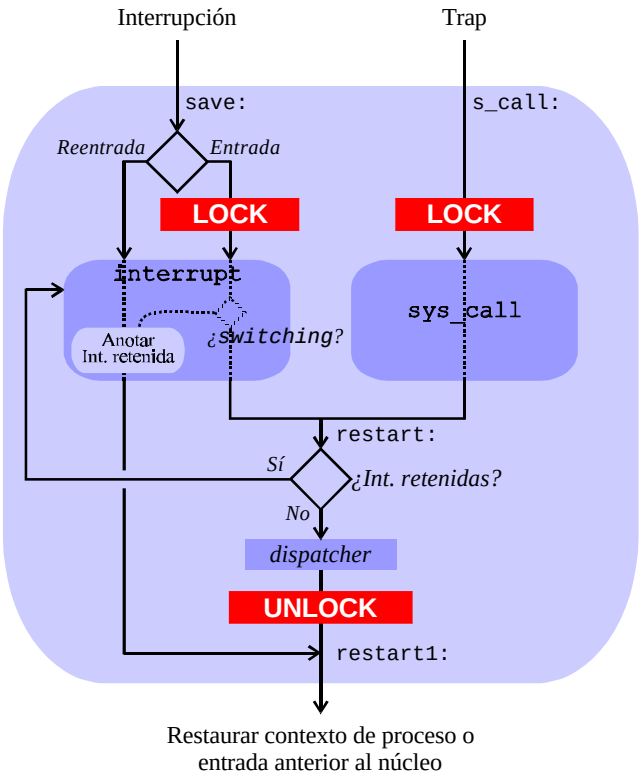


Figura 25. Flujo de entradas y reentradas respecto al cerrojo del núcleo

Puesto que hablamos de un sistema microkernel, el tiempo de contención producido por el uso de este cerrojo de granularidad gruesa se supone pequeño, y por lo tanto permisible. Aquí encontramos una diferencia con respecto a la estrategia de sincronización de los sistemas monolíticos como Linux, donde desde un primer instante se observa gran inquietud por lograr regiones críticas más pequeñas (granularidad más fina). Podríamos plantear esta misma estrategia (granularidad fina, solución B) en Minix: proteger con cerrojos independientes las distintas secciones críticas internas al núcleo: programación del controlador de interrupciones, manipulación de la cola de interrupciones retenidas y métodos del planificador.

- El controlador de interrupciones se programa en cada manejador asociado a un vector de interrupción, lo que obliga a modificar el código de todos ellos. Puesto que en realidad estos manejadores se definen (en `mpx386.s`) como un par de macros, los cambios necesarios son pocos.
- La cola de interrupciones retenidas se manipula (ver Figura 25) en `interrupt` y en `restart` (a través de la función `unhold`, en `proc.c`). Además, desde `unhold`, mientras se recorre la lista de interrupciones retenidas, se invoca a `interrupt`. Esto obliga a abrir y cerrar el cerrojo de esta cola para cada interrupción retenida.
- Todos las invocaciones a los métodos del planificador deben ser protegidas por cerrojos.

Esto (solución B) supone un mayor número de modificaciones en comparación con la protección del núcleo completo (solución A). Pero lo realmente interesante es que este esfuerzo se vea compensado con una disminución del tiempo de contención perdido en los cerrojos. Veamos si es así.

Supongamos la siguiente situación: en un instante dado, un procesador está ejecutando código del núcleo (cualquiera de las secciones críticas antes identificadas), y en ese mismo instante, se eleva una interrupción. Pueden ocurrir dos cosas:

1) La interrupción se entrega al mismo procesador que ejecuta el núcleo.
2) La interrupción se entrega a otro procesador que no está ejecutando el núcleo.

Veamos qué ocurre en estas situaciones suponiendo ambas soluciones propuestas: un núcleo protegido por un único cerrojo (A), y un núcleo protegido por cerrojos en las secciones críticas internas (B).

A) Con la protección del núcleo completo, la situación *1)* produciría una reentrada, que funcionaría de forma semejante a como ocurre en el núcleo original de Minix: la interrupción será retenida y atendida tan pronto finalice la entrada anterior. En el caso *2)* el procesador interrumpido esperará en la entrada del núcleo hasta que el primer procesador termine y entonces atenderá la interrupción como una entrada normal.

B) Con la protección de las distintas secciones críticas, la situación *1)* produciría una reentrada, que funcionaría de forma semejante a como ocurre en el núcleo original de Minix: la interrupción será retenida y atendida tan pronto finalice la entrada anterior[3]. En el caso *2)* el procesador interrumpido deberá entrar en el núcleo como una reentrada (ya que ya hay una entrada anterior, aunque sea por parte de otro procesador), y retener la interrupción, que no podrá ser atendida hasta que el primer procesador termine su entrada. Mientras, el segundo procesador podrá seguir su ejecución anterior tras superar posibles contenciones al acceder al controlador de interrupciones y a la cola de interrupciones retenidas. Pero, a cambio, el primer procesador se verá sobrecargado por la liberación y bloqueo repetido de los cerrojos de la cola de interrupciones retenidas, del acceso al controlador de interrupciones y acceso a los métodos del planificador tantas veces como interrupciones retenidas estén pendientes.

Así, observamos que la ganancia en rendimiento no es mucho mayor y en cambio sí se complica el diseño. Puesto que el primer criterio de diseño aplicado en Minix SMP es mantener en lo posible la estructura original de Minix, y que el objetivo de Minix no es el rendimiento sino la claridad, se la optado por la primera solución: proteger el núcleo completo con un cerrojo de granularidad gruesa.

[3] Recordemos que un procesador no puede producir interbloqueo consigo mismo por reentrada en ninguna de las secciones críticas porque, bien las interrupciones se encuentran inhabilitadas en esas secciones (controlador de interrupciones, interrupciones retenidas) o bien no se accede a esas secciones por reentrada (planificador) debido a que se retiene la interrupción.

4.3. Planificador SMP

4.3.1. Asignación de procesos a procesadores

La correcta asignación de procesos a los distintos procesadores del sistema requiere que se añada al descriptor de proceso información acerca del procesador que lo está ejecutando o que tiene asignado (Figura 26), de manera que para despachar un proceso en un procesador no sólo es necesario que esté preparado, sino que, además, no *pertenezca* a otro procesador. Esta relación de pertenencia se puede interpretar de muchas formas según diferentes políticas de planificación. Muchos sistemas operativos asignan un proceso a un procesador, de manera que siempre se ejecuta en ese procesador hasta que el proceso acaba *[19]*. Esto permite aprovechar mejor los sistemas de cache, pero complica el diseño del planificador ya que hay que prever mecanismos de migración a otros procesadores en caso de fallo o desactivación de algún procesador, sobrecarga, etc.

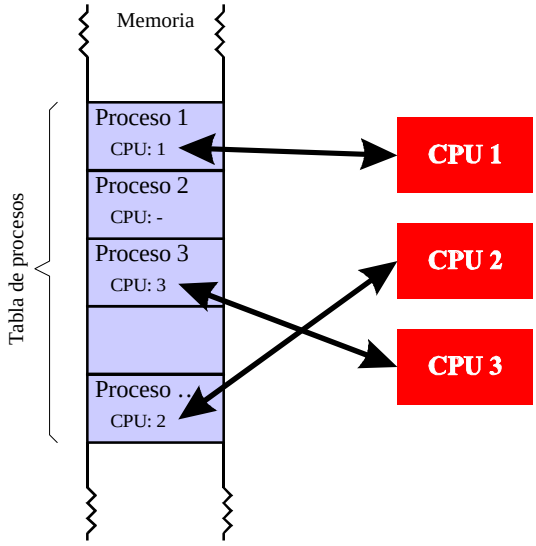


Figura 26. Asignación de procesos a procesadores

Como hemos visto en el capítulo 2, arquitectura multiprocesador Intel asegura de forma transparente la coherencia entre las distintas caches del sistema y la memoria principal. Por lo tanto el sistema operativo se descarga de toda la labor relacionada con la escritura e invalidación de las caches de los procesadores durante el cambio de contexto y la migración de procesos.

Aprovechando esta característica, el mecanismo de asignación de procesos a procesadores adoptado en Minix SMP es muy sencillo. Consiste en entender que un proceso pertenece a un procesador sólo si éste lo está ejecutando en este momento. Es decir, que cualquier procesador puede ejecutar cualquier proceso preparado con la única condición de que no lo esté ejecutando ningún otro procesador. Una vez más optamos por un diseño sencillo y respetuoso con la estructura original de Minix, antes que por otro que puede llegar a aportar mejores prestaciones, siempre teniendo en cuenta las características particulares del hardware utilizado en este caso.

4.3.2. Protección del planificador

Algunas tareas invocan directamente los métodos del planificador. Como ya hemos visto en la sección 3.3, en el núcleo original de Minix el acceso al planificador desde las tareas de E/S se hace mediante una variante de los métodos del planificador con un prefijo "`lock_`", que *"envuelve"* la ejecución de los métodos con una bandera `switching` que, mientras está establecida, fuerza a cualquier entrada al núcleo por

`interrupt` a retener la interrupción en una cola. Así, la ejecución de los métodos del planificador mediante la variante `lock_*` es atómica. Sin embargo, la bandera `switching` sólo protege los métodos del planificador del acceso simultáneo en una situación muy concreta, que es la única que puede dase en un entorno monoprocesador: cuando se produce una entrada al núcleo por interrupción de una tarea de E/S que accedía al planificador desde un método `lock_*`. La situación contraria, que una tarea de E/S intente acceder mientras una interrupción manipula el planificador, que no puede darse en el núcleo original porque las tareas no pueden interrumpir al núcleo, sí puede sin embargo darse en el núcleo multiprocesador. Así, dos nuevas situaciones pueden darse:

- Mientras un procesador ejecuta el planificador en el contexto del núcleo (sirviendo una interrupción o una llamada al sistema), otro procesador accede al planificador desde una tarea de E/S.
- Dos o más tareas de E/S acceden a los métodos del planificador al mismo tiempo (desde distintos procesadores).

Para solucionar este problema, la bandera `switching` se ha reemplazado por un cerrojo (independiente del cerrojo principal del núcleo) que impide el acceso simultáneo en cualquier combinación de intentos de acceso. Así, no se pueden ejecutar dos métodos del planificador simultáneamente, tal y como representa la Figura 27:

- Los métodos del planificador y las funciones de tratamiento de llamadas al sistema e interrupciones invocan internamente a las funciones sin "`lock_`". Por ejemplo, `mini_send` invoca a `ready` y a `unready`, pero no a `lock_ready` ni a `lock_unready`, y, a su vez, `ready` invoca a `pick_proc`, pero no a `lock_pick_proc`.
- Las tareas de E/S invocan a las funciones con "`lock_`". (Observamos que `lock_mini_rec` no existe. La razón es que no se invoca nunca.)
- `interrupt` y `sys_call` están también protegidas por el cerrojo. `sys_call` se ha renombrado a `sys_call_SMP` y se ha creado una nueva `sys_call` similar a los "`lock_`", que invoca a `sys_call_SMP` bajo la protección del cerrojo.

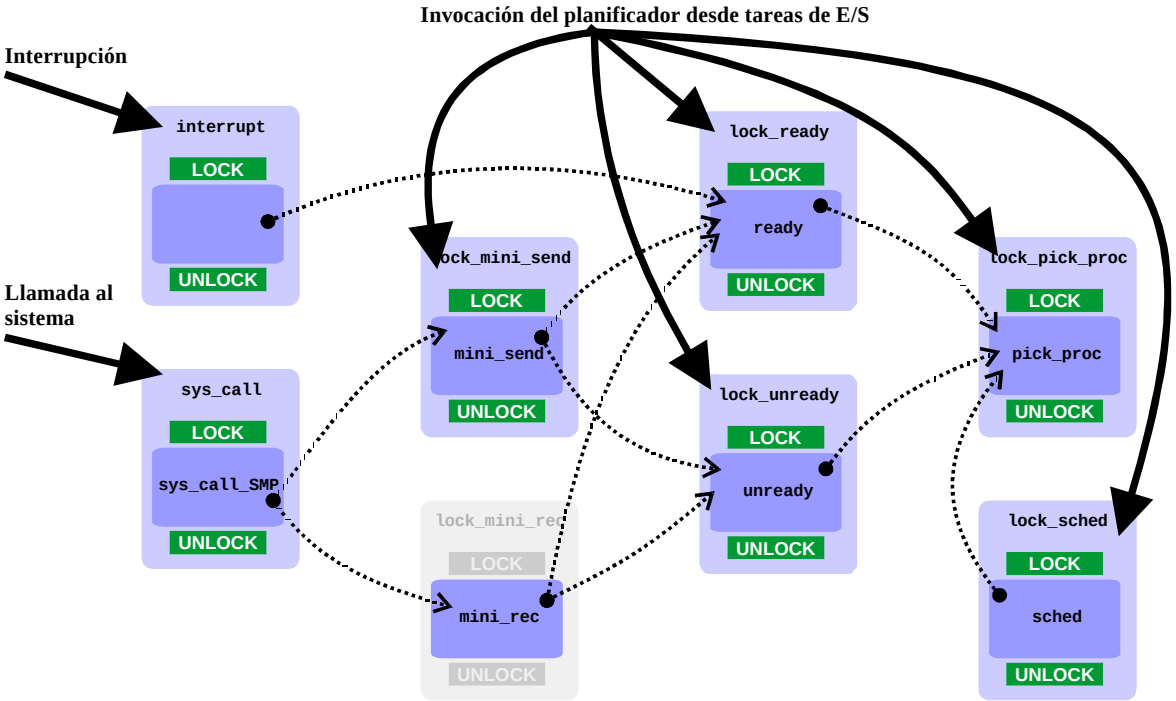


Figura 27. Esquema de dependencias en los métodos del planificador

La contención debida a estos cerrojos, tanto dentro del núcleo como dentro de las tareas de E/S, es muy breve, ya que se trata de fragmentos muy breves de código.

4.4. Replicación de estructuras

En el interior del núcleo se utilizan algunas estructuras de datos que necesitan ser replicadas para los distintos procesadores del sistema:

- Algunas variables globales del núcleo `proc_ptr`, `bill_ptr`, `k_reenter` y otras relacionadas con ellas en las tareas E/S, como `prev_ptr` en la tarea del reloj.
- La estructura TSS, relacionada con la arquitectura de procesos Intel.
- La tarea IDLE, para evitar que varios procesadores ejecuten el mismo proceso.

Estas estructuras se convierten en vectores con tantos elementos como procesadores existan en el sistema, y cada procesador debe acceder únicamente a su índice correspondiente. Para ello, cada procesador puede conocer su propia identidad a través de uno de los registros de su APIC local. Normalmente el BSP se identifica con el número 0, y los AP con valores sucesivos. Obsérvese que lo que se identifica no es el procesador, sino el APIC. Así, el I/O APIC también se identifica con un número, normalmente el siguiente al asignado al último APIC local (ver Figura 5).

En el siguiente capítulo analizaremos en detalle las causas que fuerzan la necesidad de replicar estas estructuras.

5 Detalles de implementación

Hemos visto que la arquitectura multiprocesador de Intel consiste esencialmente en un conjunto de dispositivos externos al procesador que deben invocarse sólo para tareas de comunicación entre procesadores y gestión avanzada de interrupciones. Así, la arquitectura MP no presenta ninguna novedad respecto a una arquitectura de un único procesador ya que es una extensión a su interfaz de programación original

Los cambios realizados sobre el núcleo original se han centrado, como hemos descrito en los principios de diseño, en la protección de algunas regiones críticas, la replicación de algunas estructuras de datos, y la implementación del nuevo interfaz con el hardware MP.

Veamos ahora en detalle cómo se han realizado estas modificaciones.

5.1. Ficheros fuente implicados

En el desarrollo de Minix SMP se han creado varios archivos de código fuente y se han modificado algunos de los que forman parte del núcleo (en `/usr/src/kernel`) y de la configuración general de minix (en `/usr/include/minix`)

Los ficheros nuevos contienen la interfaz multiprocesador:

- `mp.c`: contiene el código de las nuevas funciones de soporte multiprocesador escritas en C.
- `mp.h`: contiene la declaración de la interfaz pública del soporte multiprocesador.
- `xmp.s`: contiene el código de las nuevas funciones de soporte multiprocesador escritas en ensamblador.
- `xmp.h`: contiene algunas definiciones y macros de ensamblador públicas y empleadas en `xmp.s`.
- `mptable.h`: es una extensión de `mp.c`, separada de `mp.c` para evitar que éste tenga un tamaño excesivo, donde se definen las estructuras y constantes necesarias para la manipulación de la Tabla de Configuración MP.
- `mpinfo.c`: es otra extensión de `mp.c`, separada de `mp.c` para evitar que éste tenga un tamaño excesivo, donde se definen las funciones que se utilizan para mostrar información sobre la

configuración multiprocesador, principalmente en lo referido a la detección del hardware (Tabla de Configuración MP) y proceso de arranque multiprocesador.

Además, se han registrado cambios en los siguientes ficheros del núcleo de Minix (en `/usr/src/kernel`): Makefile, `clock.c`, `dmp.c`, `dp8390.c`, `driver.c`, `drvlib.c`, `exception.c`, `glo.h`, `keyboard.c`, `main.c`, `main.c`, `mcd.c`, `mpx386.s`, `proc.c`, `proc.h`, `protect.c`, `protect.h`, `system.c`, `table.c`, `tty.c`; y de la configuración general de minix (en `/usr/include/minix`): `com.h`, `config.h`, `const.h`.

5.2. Algunas rutinas y definiciones de base de apoyo

A continuación se introducen algunas rutinas y definiciones que, si bien no forman parte directa del código multiprocesador, sí que sirven de apoyo para su desarrollo.

Además, al final del capítulo de detalles de implementación se listan otras funciones para la impresión de información sobre el arranque multiprocesador y definiciones utilizadas para la manipulación de la Tabla de Configuración MP.

5.2.1. Manipulación del APIC local

Para la lectura y escritura del APIC local se han creado dos funciones `LOCAL_APIC_READ()` y `LOCAL_APIC_WRITE()` que aceptan como parámetro el registro del APIC local a leer o escribir, y el valor de 32 bits a escribir en el caso de `LOCAL_APIC_WRITE()`. Además se incluyen un conjunto de definiciones para los registros el APIC utilizados, así como para la interpretación de sus campos. La lectura y escritura de los registros del APIC local se hace en base a una dirección de memoria donde se encuentra el APIC. Por defecto esta dirección es FEE00000H, aunque puede variar en función de la información contenida en la Tabla de Configuración MP. La variable global `local_apic_base` mantiene esta dirección, que se actualiza durante la interpretación de la Tabla de Configuración MP. Todo ello se encuentra en el fichero `mp.c`.

mp.c

```
u32_t local_apic_base=0x0FEE00000;

#define LOCAL_APIC_ICR_LOW  0x0300 /* offset for IRC low regiter */
#define LOCAL_APIC_ICR_HIGH 0x0310 /* offset for IRC high register */
#define LOCAL_APIC_ESR             0x0280  /* offset for ESR register */
#define LOCAL_APIC_SPIV            0x00F0  /* offset for SPIV register */
#define LOCAL_APIC_TASKP    0x0080 /* offset for TASK PRIORITY register */
#define LOCAL_APIC_LDR             0x00D0  /* offset for LD register */
#define LOCAL_APIC_DFR             0x00E0  /* offset for DF register */
#define LOCAL_APIC_IDR             0x0020  /* offset for ID register */
#define LOCAL_APIC_LVTL0    0x0350 /* offset for LVT LINTIN0 */
#define LOCAL_APIC_LVTL1    0x0360 /* offset for LVT LINTIN1 */
#define LOCAL_APIC_EOI             0x00B0  /* offset for EOI register */

#define LVT_DM_FIXED        0      /* delivery mode in L.APIC LVT */
#define LVT_DM_NMI          4      /* delivery mode in L.APIC LVT */
#define LVT_DM_EXTINT               7       /* delivery mode in L.APIC LVT */

#define LVT_MASKED_SHIFT    16     /* shift for masked LINTINx bit */
#define LVT_DM_SHIFT        8      /* shift for delivery mode field */

#define DELIVERY_FIXED      0      /* 000 = FIXED */
#define DELIVERY_INIT       5      /* 101 = INIT IPI */
#define DELIVERY_STARTUP 6  /* 110 = STARTUP IPI */
#define DELIVERY_SHIFT      8      /* delivery bit position in dword */
#define LEVEL_SHIFT  14     /* level bit position in dword */
#define TRIGGER_SHIFT       15     /* trigger mode bit position in dword */
```

```
#define PHYSICAL_DEST       0      /* physical destination addr mode */
#define LOGICAL_DEST 1      /* logical destination addr mode */
#define DEST_MODE_SHIFT     11     /* destination mode bit position in dword */

#define DELIVERY_STATUS_SHIFT      12      /* status of ipi */
#define DELIVERY_PENDING    1      /* ipi is being sent */
#define DELIVERY_IDLE              0       /* sent: no work to be done */
#define DELIVERY_STATUS     (LOCAL_APIC_READ(LOCAL_APIC_ICR_LOW) \
                                      & (1<<DELIVERY_STATUS_SHIFT))

#define DEST_FIELD   0      /* destination by field */
#define DEST_SELF    1      /* destination is self cpu */
#define DEST_ALL     2      /* destination is all cpus */
#define DEST_OTHERS  3      /* destination is all cpus but self */
#define DEST_SHORT_SHIFT 18 /* destination shorthand bit position */

#define DEST_FIELD_SHIFT (56-32)   /* destination field bits position */

#define ENABLE_APIC_SHIFT   8      /* enable APIC bit position in SPIV reg */

void LOCAL_APIC_WRITE(u32_t reg, u32_t val) {
   phys_copy_dword( vir2phys(&val), local_apic_base+reg );
}

u32_t LOCAL_APIC_READ(u32_t reg) {
   u32_t val;
   phys_copy_dword( local_apic_base+reg, vir2phys(&val) );
   return val;
}
```

5.2.2. Manipulación del I/O APIC

Aunque en la versión actual de Minix SMP no se utiliza el I/O APIC como controlador de interrupciones E/S, se han incluido en su código las definiciones básicas para su manipulación, semejantes a las proporcionadas para el APIC local: funciones `IO_APIC_READ()` e `IO_APIC_WRITE()` que aceptan como parámetro el registro del APIC a leer o escribir, y el valor de 32 bits a escribir en el caso de `IO_APIC_WRITE()`. Además se incluyen un conjunto de definiciones para los registros el APIC, así como para la interpretación de sus campos. La lectura y escritura de los registros del I/O APIC se hace en base a una dirección de memoria donde se encuentra el APIC, suponiendo como condición de partida que sólo existe un I/O APIC en el sistema. Por defecto esta dirección es FEC00000H, aunque puede variar en función de las entradas de tipo I/O APIC que pueden aparecer en la Tabla de Configuración MP. La variable `io_apic_base` mantiene esta dirección, que se actualiza durante la interpretación de la Tabla de Configuración MP. Todo ello se encuentra en el fichero `mp.c`.

mp.c

```
u32_t io_apic_base=0xFEC00000;

#define IO_APIC_ACC_REG_SEL 0x00
#define IO_APIC_ACC_WIN_REG 0x10

#define IO_APIC_ID_REG                      0x00
#define IO_APIC_VERSION_REG        0x01
#define IO_APIC_ARBITR_REG         0x02
#define IO_APIC_REDTBL_REG_LOW(n)  (0x10 + (2 * n))
#define IO_APIC_REDTBL_REG_HIGH(n) (0x11 + (2 * n))

#define IOA_ID_SHIFT 24     /* id field in io-apic id reg. */
#define IOA_VER_SHIFT       0      /* version field in io-apic version reg. */
#define IOA_MXENT_SHIFT     16     /* max entries field in io-apic version reg. */
```

```
#define IOA_ARB_SHIFT       24     /* arb. field in io-apic arbitration reg. */
#define IOA_ID_MASK  (0x0F<<IOA_ID_SHIFT)
#define IOA_VER_MASK (0xFF<<IOA_VER_SHIFT)
#define IOA_MXENT_MASK      (0xFF<<IOA_MXENT_SHIFT)
#define IOA_ARB_MASK (0x0F<<IOA_ARB_SHIFT)

void IO_APIC_WRITE(u32_t reg, u32_t val) {
   reg &= 0xFF;
   phys_copy_dword( vir2phys(&reg), io_apic_base+IO_APIC_ACC_REG_SEL);
   phys_copy_dword( vir2phys(&val), io_apic_base+IO_APIC_ACC_WIN_REG);
}

u32_t IO_APIC_READ(u32_t reg) {
   u32_t val;
   phys_copy_dword( vir2phys(&reg), io_apic_base+IO_APIC_ACC_REG_SEL);
   phys_copy_dword( io_apic_base+IO_APIC_ACC_WIN_REG, vir2phys(&val));
   return val;
}
```

5.2.3. Manipulación del CMOS

Para el acceso a la memoria CMOS requerido para la reprogramación del código y dirección del arranque caliente (*warm-reset*) utilizado en la secuencia de IPI INIT durante el arranque multiprocesador se han definido dos funciones `CMOS_READ()` y `CMOS_WRITE()` en `mp.c` que aceptan como parámetro la dirección CMOS a leer o escribir, y el valor de 8 bits a escribir en el caso de `CMOS_WRITE()`.

mp.c

```
void cmos_write(u8_t addr, u8_t value) {
   out_byte(0x70, addr);
   out_byte(0x71, value);
}

u8_t cmos_read(u8_t addr) {
   out_byte(0x70, addr);
   return in_byte(0x71);
}
```

5.2.4. Macros y funciones de propósito general

5.2.4.1. Función phys_copy_dword()

Las funciones de acceso al APIC local y al I/O APIC requieren el acceso a posiciones de memoria física fuera del espacio de direccionamiento del núcleo. Además, estos accesos deben hacerse de forma única a palabras de 32 bits, por lo que no puede utilizarse la función existente en Minix `phys_copy()`, que transfiere una determinada cantidad de bytes entre dos direcciones de memoria física. Por esta razón se ha creado la función `phys_copy_dword()`, similar a `phys_copy()` con la diferencia de que no se indica la longitud de los datos a transferir (ya que se supone que son 32 bits) y que la lectura y escritura no se hace byte a byte, sino en un único acceso de 32 bits. Encontramos esta función en `xmp.s`.

xmp.s

```
! PUBLIC void phys_copy_dword(phys_bytes source, phys_bytes destination);
! Copy a block of physical memory.
PC_ARGS       =      4 + 4 + 4 + 4 ! 4 + 4
!             es edi esi eip        src dst
```

```
        .align 16
_phys_copy_dword:
        cld
        push   esi
        push   edi
        push   ds

        mov    eax, FLAT_DS_SELECTOR
        mov    ds, ax

        mov    eax, PC_ARGS(esp)    ! src
        mov    ebx, PC_ARGS+4(esp)  ! dst

        mov    edx, (eax)
        mov    (ebx), edx

        pop    ds
        pop    edi
        pop    esi
        ret
```

5.2.4.2. Macros `FOR_EACH`

A lo largo de todo el código C del soporte multiprocesador se repiten un conjunto de bucles de tipo `for` que se utilizan para la repetición de la misma operación para cada procesador. Por esta razón se han definido en `mp.h` tres macros que sustituyen a tres variantes del bucle. Todas ellas aceptan como parámetro la variable de tipo entero que mantiene el contador y que numera cada procesador.

- `FOR_EACH_CPU`: recorre todos los procesadores del sistema.
- `FOR_EACH_AP`: recorre todos los AP del sistema (todos los procesadores excepto el BSP).
- `FOR_EACH_ENABLED_CPU`: recorre todos los procesadores del sistema marcados como activos (en el vector `cpu_available`)

mp.h

```
#define FOR_EACH_CPU(cpu)          for(cpu=0; cpu<MP_NR_CPUS; ++cpu)

#define FOR_EACH_AP(cpu)           for(cpu=1; cpu<MP_NR_CPUS; ++cpu)

#define FOR_EACH_ENABLED_CPU(cpu)  FOR_EACH_CPU(cpu) \
                                   if (cpu_available[cpu]==CPU_ENABLED)
```

5.3. Configuración del núcleo multiprocesador

Con el objetivo de mantener la estructura original del núcleo lo más intacta posible y realizar un trabajo adaptable a nuevas configuraciones de hardware, se han definido dos constantes que aparecerán a lo largo de todo el desarrollo. Una es **`ENABLE_MP`** permite habilitar el soporte para multiprocesador (valor 1) o recuperar el núcleo casi original (valor 0). La otra es **`MP_NR_CPUS`** y permite reservar espacio en el núcleo para un número máximo de procesadores. El valor de `MP_NR_CPUS` es ignorado en el caso de que `ENABLE_MP` sea 0.

Ambas constantes, junto con dos cadenas que registran la versión del desarrollo para mostrarlo durante el arranque, se localizan en el fichero `<minix/config.h>`.

<minix/config.h>

```
/* Enable or disable multiprocessor suport (only for intel architecture),
   and configure number of processor (1 cpu implicit if MP not enabled) */
#define ENABLE_MP    1      /* 1 enable, 0 disable */
#define MP_NR_CPUS   2      /* ignored if ENABLE_MP is 0 */
/* MP version, only for printing */
#define MP_RELEASE   "1"
#define MP_VERSION   "0"

...
...

#if (ENABLE_MP != 1) /* only 1 cpu if multiprocessor not enabled */
#undef  MP_NR_CPUS
#define MP_NR_CPUS          1
#endif
```

5.4. Iniciación de los procesadores de aplicación

El primero de los primeros pasos llevados a cabo en el desarrollo de Minix MP ha sido la implementación de todo lo referente al arranque de los procesadores de aplicación (AP). Puesto que en el momento de arrancar Minix, el BIOS ha configurado el sistema para que funcione como si de un monoprocesador se tratara, todo el proceso de inicialización de Minix es ajeno al número de procesadores que existan. Justo en el momento previo a la primera invocación del *dispatcher*, en `main()` dentro de `main.c`, cuando todo el sistema se encuentra en disposición de funcionar, se invoca a la rutina de inicialización del sistema multiprocesador.

La iniciación se hace invocando a la función `mp_start()`, localizada en el fichero `mp.c`. Durante el proceso de iniciación es conveniente emitir mensajes referidos a cómo se va desarrollando el proceso, mensajes de error, etc. En este momento del arranque no es posible utilizar la rutina `printk()` para escribir en la consola, por lo que se ha desarrollado un método alternativo: rellenar una cadena de caracteres que se muestra cuando se inicia el terminal (*tty*) con ayuda de 2 funciones que permiten escribir cadenas (`ADDS_MP_STATUS`) y números (`ADDN_MP_STATUS`, con indicación de la base y el número mínimo de dígitos).

mp.c

```
void ADDS_MP_STATUS(char *message);
void ADDN_MP_STATUS(u32_t number, u32_t basis, u32_t size);
```

El primer paso de la inicialización es detectar la configuración del hardware, para lo cual se localiza la Tabla de Configuración MP que el BIOS ha dispuesto en algún punto de la memoria. En ella encontraremos información relativa a la versión de la especificación, número de procesadores, identificación de APIC, localización de APIC en memoria, buses, organización inicial de interrupciones, etc. Una vez localizada la *Floating Pointer Structure (FPS)* accedemos a la cabecera de la Tabla de Configuración MP, que almacenamos en la estructura `mph`. De aquí obtenemos la dirección de acceso al APIC local, que almacenamos en la variable `local_apic_base`.

mp.c

```
void mp_start() {
/* Load MP configuraton and wake up each CPU in the system */

   u32_t trampoline_addr;
```

```
    int cpu;

    /* Start messaging to console */
    MP_STATUS[0]=0;
    ADDS_MP_STATUS("IntelMP multiprocessor kernel support v" \
                   MP_RELEASE "." MP_VERSION "\n");

    /* Try to load MP structures from memory */
    if (load_fps()) {
       /*print_fps_info();*/
       if (load_mph()) {
          /*print_mp_configuration();*/
          local_apic_base=mph.mpch_apic_addr;
          process_mp_configuration();
          /* Try to allocate the trampoline for APs to start */
          if ((trampoline_addr=find_trampoline())) {
             /* Start each processor */
             FOR_EACH_AP(cpu) {
                if (cpu!=1) milli_delay(100);
                ap_running_flag=0;
                send_init_ipi(trampoline_addr, cpu);
                send_startup_ipi(trampoline_addr, cpu);
                if (! AP_running()) {
                   ADDS_MP_STATUS("\n\n*** WARNING! AP#");
                   ADDN_MP_STATUS(cpu,10,1);
                   ADDS_MP_STATUS(" is not running ***\n\n");
                }
             }
             free_trampoline(trampoline_addr);
          }
          else ADDS_MP_STATUS("MP ERROR!: Unable to allocate trampoline\n");
       }
    }
}
```

Una vez recabada esta información sabremos cuántos procesadores tenemos realmente (en la configuración MP de Minix previamente tendremos que haber establecido un número máximo de procesadores para que se reserve la memoria necesaria para los vectores de estructuras replicadas), cómo se identifican (número asignado a sus APIC locales), etc.[4]

A continuación comenzamos el procedimiento de arranque de cada AP. Según la especificación MP de Intel, éste se realiza enviando una secuencia especial de interrupciones (IPI) al AP que se desea arrancar, mediante el uso del APIC local del BSP. Como consecuencia de ello, el AP comenzará a ejecutar en modo real a partir de una dirección configurada en esas interrupciones. En dicha dirección previamente nos hemos de asegurar la presencia del primer fragmento de código que ejecutará el AP, una especie de pista de aterrizaje (*trampoline*, según la terminología de Linux).

La función `find_trampoline()` intenta localizar una dirección válida de memoria donde alojar el código de arranque del AP, y si lo consigue, lo copia en esa dirección y la devuelve como valor de retorno. Ahora es el momento de enviar a cada procesador la secuencia de IPI que lo hagan saltar a la dirección habilitada. Antes, iniciamos a cero una bandera que cada AP que arranque con éxito deberá establecer para indicar al BSP que está funcionando correctamente. La comprobación del arranque se hace en la rutina `AP_running` (localizada en `mp.c`) que espera un tiempo prudencial para que el AP establezca el valor mágico `AP_LIFE_FLAG_MARK` en la variable `ap_running_flag`.

[4] En la versión actual se da por supuesto que los procesadores se numeran como 0 para el BSP, 1, 2, 3, etc. para los AP, y un valor por encima del último AP para el I/O APIC.

mp.c

```
int AP_running(void) {
/* Wait for an AP to write an value in the flag that means it is running */
   int max_loops=100;

   while ((ap_running_flag!=AP_LIFE_FLAG_MARK) && (--max_loops))
       milli_delay(1);
   return ap_running_flag==AP_LIFE_FLAG_MARK;
}
```

Después de iniciar el último procesador, liberamos el espacio ocupado por el código de arranque del AP y retornamos al proceso de arranque normal de Minix, que invocará el *dispatcher* para comenzar a ejecutar los procesos. Para entonces, todos los AP deben estar funcionando y esperando bloqueados por el cerrojo principal del núcleo. Cuando el BSP abandone el *dispatcher*, abrirá el cerrojo y uno de lo AP podrá continuar, invocando el *dispatcher* para comenzar a ejecutar y liberando al siguiente AP, etc. La Figura 28 ilustra el procedimiento de arranque multiprocesador.

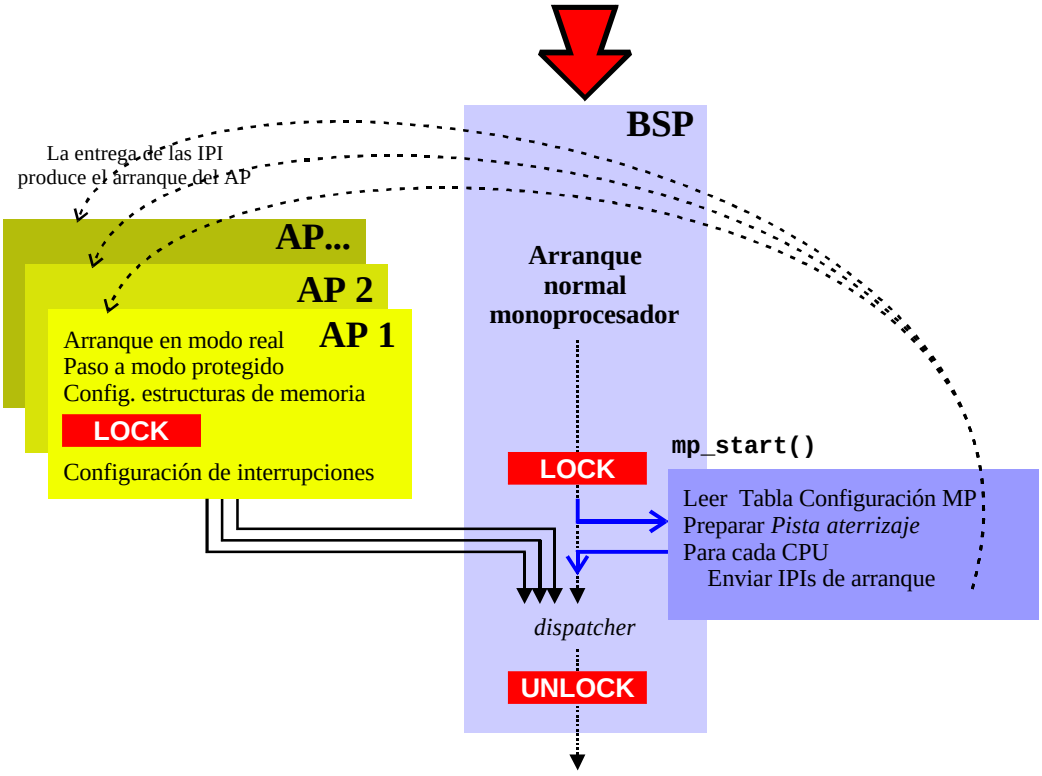


Figura 28. Procedimiento de arranque de los procesadores de aplicación (AP)

Veremos todos estos pasos de forma detallada en las siguientes secciones.

5.4.1. Identificación de la FPS

Necesitamos localizar la *Floating Pointer Structure* porque, entre otra información, contiene la dirección de la Tabla de Configuración MP. Como ya hemos visto, la FPS no se encuentra en una dirección de memoria fija, sino dentro de un rango de direcciones conocidas alineada en párrafos de 16 bytes. Para encontrarla en memoria dispone de un campo con una firma (*SIGNATURE*) y otro de verificación (*CHECKSUM*). El algoritmo recorre secuencialmente las direcciones válidas de memoria en busca de la

firma hasta que encuentra una cuyo checksum sea correcto, quedando la información almacenada en una variable global `fps` de tipo `struct floating_pointer`.

mp.c

```
int test_fps_checksum(struct floating_pointer *fps) {
   unsigned char *data, checksum;
   int i;

/* Check in a FPS structure for a valid signature and checksum
   Return noncero on success */

   /* fisrt look for signature */
   for (i=0; i<4; i++)
      if (fps->fp_signature[i] != SIGN_FPS[i]) return 0;

   /* then calculate checksum*/
   data=(unsigned char *)fps;
   checksum=0;
   for (i=0; i<sizeof (struct floating_pointer); i++)
      checksum+= data[i];
   return (!checksum);  /* checksum ok if sums 0 */
}



int load_fps(void) {
/* Find and load a valid FPS structure. Retun nonzero on success */

   unsigned addr;

   /* for the moment, scan the whole base memory, in
      16-byte steps */
   for (addr=0xF5B30; addr<0xFFFF0; addr+=0x10) {
      phys_copy ( addr , vir2phys(&fps) , sizeof(fps) );
      if (test_fps_checksum(&fps)) {
         /*ADDS_MP_STATUS("MP floating pointer struct found at 0x");
         ADDN_MP_STATUS(addr,16,0);
         ADDS_MP_STATUS("\n");*/
         return 1;
      }
   }
   ADDS_MP_STATUS("MP scan FAILLED! MP config table not found in base memory\n");
   return 0;
}
```

5.4.2. Identificación de la Tabla de Configuración MP

Con la dirección proporcionada por la FPS localizamos la Tabla de Configuración MP[5]. Se guarda el contenido de la cabecera en una variable global `mph` de tipo `struct mp_config_header` y se comprueba que la tabla es correcta mediante el campo de checksum que incluye.

mp.c

```
int test_mptable_checksum(void) {
/* Cheks a MP table header structure for a valid signature and checksum.
   Return noncero on success */
```

[5] En la versión actual de Minix SMP se da por supuesto que la configuración multiprocesador no es estándar y existe una Tabla de Configuración MP.

```
   unsigned char data, checksum;
   int i;

   /* first match the signature */
   for (i=0; i<4; i++)
      if (mph.mpch_signature[i] != SIGN_MP_C_HEADER[i] )
         return 0;

   /* then calculate checksum */
   checksum=0;
   for (i=0; i<mph.mpch_length; i++) {
      /* the complete table is in another segment, so we need copy
         each byte into kernel address space (mph is only the header) */
      phys_copy(fps.fp_mp_table+i, vir2phys(&data), 1);
      checksum +=data;
   }
   return (!checksum);  /* checksum ok if sums 0 */
}



int load_mph(void) {
/* Find and load a valid MP config header struct. Return noncero on success */
   unsigned addr;

   phys_copy ( fps.fp_mp_table , vir2phys(&mph) , sizeof(mph) );
   if (test_mptable_checksum()) return 1;
   ADDS_MP_STATUS("MP config table checksum FAILLED!\n");
   return 0;
}
```

5.4.3. Interpretación de la Tabla de Configuración MP

La interpretación de la Tabla de Configuración MP pretende recoger diversa información acerca de la configuración del hardware. Resulta especialmente útil para detectar situaciones no soportadas por la versión actual. El algoritmo recorre cada entrada de la tabla base, almacenando el contenido en un registro de tipo adecuado. Existen funciones (en el fichero `mpinfo.c`, cuya descripción se encuentra al final de este capítulo) que imprimen en pantalla la información recogida, pero las llamadas a estas funciones están omitidas (puestas como comentarios) para no saturar la pantalla de arranque de información inútil para el usuario normal. En caso de encontrar alguna situación anómala, se muestra un mensaje de advertencia, pero no se toma ninguna acción.

mp.c

```
void process_mp_configuration(void) {
/* Scan MP table and find some information about MP configuration */

u32_t next;
   int cpu_count, bus_count, apic_count, ioint_count, lint_count;
   int i=mph.mpch_entry_count;
   u8_t this_entry;
   struct io_apic_entry     ae;
   struct cpu_entry  ce;
   struct io_int_entry      ioie;
   struct l_int_entry       lie;

   cpu_count= bus_count= apic_count= ioint_count= lint_count= 0;

   next=fps.fp_mp_table+sizeof (struct mp_config_header);
   while (i-- > 0) {
      phys_copy(next, vir2phys(&this_entry), 1);
      switch (this_entry) {
```

```
            case CPU_ENTRY_ID :
               if (++cpu_count > 2)
                  ADDS_MP_STATUS("MP PANIC: only 2 cpus currently supported!\n");
               phys_copy(next,vir2phys(&ce),sizeof(struct cpu_entry));
               next+=sizeof(struct cpu_entry);
               PRINT_CPU_ENTRY(cpu);
               if ((ce.ce_flags & CE_EN_FLAG_MASK)==0)
                  ADDS_MP_STATUS("MP PANIC: disabled apics not currently supported\n");
               break;
            case BUS_ENTRY_ID :
               bus_count++;
               next+=sizeof(struct bus_entry);
               break;
            case IO_APIC_ENTRY_ID :
               if (++apic_count > 1)
                  ADDS_MP_STATUS("MP PANIC: only 1 I/O APIC currently supported!\n");
               phys_copy(next,vir2phys(&ae),sizeof(struct io_apic_entry));
               next+=sizeof(struct io_apic_entry);
               /*PRINT_IO_APIC_ENTRY(ae);*/
               if ((ae.ae_flags & AE_EN_FLAG_MASK)==0)
                  ADDS_MP_STATUS("MP PANIC: disabled apic not currently supported!");
               io_apic_base=ae.ae_address;
               break;
            case IO_INT_ENTRY_ID :
               ioint_count++;
               phys_copy(next,vir2phys(&ioie),sizeof(struct io_int_entry));
               next+=sizeof(struct io_int_entry);
               /*PRINT_IO_INT_ENTRY(ioie);*/
               break;
            case LOCAL_INT_ENTRY_ID :
               lint_count++;
               phys_copy(next,vir2phys(&lie),sizeof(struct l_int_entry));
               next+=sizeof(struct l_int_entry);
               /*PRINT_LOCAL_ENTRY(lie);*/
               break;
            default :
               ADDS_MP_STATUS("MP config table corrupt!\n");
               i=0; /* break loop */
               break;
         }
      }
      /*PRINT_MP_SUMMARY(cpu_count,bus_count,apic_count,
                       ioint_count,lint_count);*/
      if ((apic_count!=1))
         ADDS_MP_STATUS("MP PANIC: only 1 I/O APIC currently supported!\n");
   }
```

5.4.4. Preparación del área de arranque del AP

Cuando el AP comience a ejecutar por la dirección del código de arranque, lo hará en modo real. Sin embargo el código de arranque se compila junto con el resto del núcleo como código en modo protegido de 32 bits. Es posible forzar al compilador a que genere instrucciones de 16 bits utilizando los prefijos de ensamblador `o16` (*operand-size override*) y `a16` (*address-size override*):

- `o16` fuerza a que la siguiente instrucción de ensamblador se interprete con operandos de 16 bits.
- `a16` fuerza a que la siguiente instrucción de ensamblador se interprete con direcciones de 16 bits.

Cuando el ensamblador encuentra estos prefijos, los introduce en el código máquina (`o16` como 66h y `a16` como 67h) y genera una instrucción con operandos y/o direcciones de 16 bits. Pero estos operandos también funcionan de forma inversa: cuando la CPU los encuentra ejecutando en modo real de 16 bits los interpreta como que la siguiente instrucción utiliza operandos/direcciones de 32 bits.

Así, si no los utilizamos, el compilador generará instrucciones de 32 bits, pero la CPU ejecutando en modo real espera encontrar instrucciones de 16 bits. Y si los utilizamos, el compilador generará instrucciones de 16 bits y antepondrá el prefijo, pero la CPU ejecutando en modo real encontrará el prefijo esperará encontrar instrucciones de 32 bits.

La solución adoptada para este problema consiste en utilizar los prefijos cuando sea necesario y luego eliminarlos del código máquina sustituyéndolos por códigos `NOP` (*No operation*, 90h). Esto es posible fácilmente debido a que el código de arranque debe ser trasladado a la dirección de arranque desde la dirección en que el compilador lo aloje; y permisible, puesto que se trata de un breve fragmento de código, el mínimo para pasar el procesador a modo protegido de 32 bits.

La estrategia seguida es incluir en el código ensamblador los dos prefijos seguidos y siempre en el mismo orden (`o16`+`a16`), de manera que se puedan encontrar fácil e inequívocamente en el código máquina (como 6766h), y reemplazarlos por dos códigos `NOP` (9090h).

La función `disable_operand_size()`, localizada en el fichero `mp.c`, realiza esta operación sobre la zona de memora alojada para el arranque de los AP.

mp.c

```
void disable_operand_size(u32_t trampoline_addr, u32_t trampoline_sz) {
/* Change operand-size modifier codes (66h & 67h) on trampoline's machine
   code introduced by assembler and replace they with <nop> code (90h) */

   u16_t code;
   u32_t code_addr=vir2phys(&code);

   while (trampoline_sz>1) {       /* last byte not necessary to scan */
      phys_copy(trampoline_addr, code_addr, 2);
      if ((code==0x6766)) {        /* o16 a16 */
         code=0x9090;              /* nop nop */
         phys_copy(code_addr, trampoline_addr, 2);
         trampoline_addr+=2;
         trampoline_sz-=2;
      }
      else {
         trampoline_addr++;
         trampoline_sz--;
      }
   }
}
```

5.4.5. Localización del área de arranque de los AP

La recepción de las interrupciones procedentes del BSP hacen que el AP comience a ejecutar en modo real a partir de una dirección de memoria especificada en una de las interrupciones. Esta dirección debe cumplir dos condiciones: estar dentro del área de memoria baja (accesible en modo real) dentro del rango de direcciones 1000H hasta FF000H (0100:0000H hasta FF00:0000H), exceptuando desde A000H hasta C0000H (A000:0000H hasta C000:0000H), y estar alienada en páginas de 4 Kb. No es posible forzar al compilador a que aloje un fragmento de código en estas condiciones, por lo que será necesario localizar una zona válida y trasladar el código desde la zona en la que lo sitúe el compilador copiándolo byte a byte en tiempo de ejecución.

El código inicial para el arranque de un AP está escrito en ensamblador entre las direcciones `_init_ap` y `_end_init_ap` (fichero `xmp.s`). Ese es el fragmento de memoria que hay que copiar en la dirección que se localice para el arranque.

Para localizar dicha zona (función `find_trampoline()` en `mp.c`) debemos encontrar un área de memoria con el mismo tamaño que el código de arranque (entre `_init_ap` y `_end_init_ap`) que no sea utilizado por ningún otro programa. Hemos tomado el criterio de tomar un área cuyo contenido esté lleno de ceros, dando por supuesto que no contiene nada útil.

mp.c

```
u32_t find_trampoline(void) {
/* Find a memory zone suitable for allocating trampoline code.
   It will be a 0's zone, suposing this is free. */
   u32_t addr8;
   u32_t addr;
   u32_t i;
   u8_t c;

   /* size of tampoline code */
   u32_t tramp_len=(u32_t)end_init_ap-(u32_t)init_ap;

   /* Look for a hole of free memory in each valid position of base memory */
   /* For some reason 0x0000F000 is not valid!! so start form 0x10 */
   for (addr8=0x11; addr8<0x100; addr8++) {
      if (addr8==0xA0) addr8=0xC0;  /* vectors A0..BF are reserved */
      addr=addr8<<12;               /* aligned in 4kb boundaries */
      for (i=0; i<tramp_len; i++) {
         phys_copy(addr+i, vir2phys(&c), 1);
         if (c) break;
      }
      if (i==tramp_len) {
         /* Prepare and copy the tampoline */
         copy_registers_to_trampoline();
         phys_copy(vir2phys(init_ap), addr, tramp_len);
         disable_operand_size(addr,tramp_len);
         /*dump_trampoline(addr,tramp_len);*/
         return addr;                                  /* return it */
      }
   }
   return 0;
}
```

Por si la zona elegida para alojar el código de arranque no estuviera realmente libre, sino que los ceros fueran parte de alguna estructura de datos o programa, una vez arrancados todos los AP, el área se vuelve a llenar con los ceros originales (función `free_trampoline()` en `mp.c`).

mp.c

```
void free_trampoline(u32_t addr) {
/* Restore with 0's the memory zone used for trampoline */

   char dummy=0;
   u32_t tramp_len=(u32_t)end_init_ap-(u32_t)init_ap;
   while (tramp_len--) phys_copy((u32_t)&dummy,addr++,1);
}
```

La función `find_trampoline()`, además de localizar el área de memoria adecuada, se encarga de copiar y preparar los datos del programa de arranque. La función `copy_registers_to_trampoline()` (localizada en `xmp.s`) coloca en el código de arranque información necesaria para que el AP pueda configurar las estructuras de memoria. Como veremos a continuación, el código de arranque del AP contiene un conjunto de variables reservadas a almacenar el valor de los registros de memora (`ldtr`, `idtr`, `esi`, `edi` y

ebp) que debe establecer para pasar a modo protegido y configurarlo. El valor de estos registros se conoce en tiempo de ejecución, por lo que es entonces cuando deben ser establecidos.

xmp.s

```
_copy_registers_to_trampoline:
       ! Copy gdt and idt
              sgdt   (_gdtr_data)
              sidt   (_idtr_data)
       ! Copy 32-bit regsiters
              mov    eax,   esi
              mov    (_esi_data),  eax
              mov    eax,   edi
              mov    (_edi_data),  eax
              mov    eax,   ebp
              mov    (_ebp_data),  eax
              ret
```

5.4.6. Código de arranque de los AP

El código de arranque de los AP está dividido en tres partes, correspondientes a tres fases diferenciadas de arranque. En primer lugar se produce el arranque en modo real en la dirección que consiga localizarse. Luego, se pasa a modo protegido a partir de una dirección fija, donde deben establecerse (en ensamblador) las estructuras de memoria y otras configuraciones de bajo nivel. Finalmente se continúa con la última fase de arranque, ya escrita en C con los últimos pasos de preparación.

5.4.6.1. Arranque en modo real

La primera fase corresponde al código de arranque (*trampoline*) que debe ser realojado y modificado en tiempo de ejecución. Puesto que es necesario manipular este código como si fuera una estructura de datos, se ha alojado en el segmento de datos.

El código de arranque comienza en la dirección `_init_ap` y termina en `_end_init_ap`. Es necesario conocer el punto final del código para calcular su longitud. Al principio del código se introduce un espacio de variables con un desplazamiento conocido respecto al principio del código (`_init_ap`). En estas variables, el BSP, mediante la función `copy_registers_to_trampoline()`, escribirá valor de los registros de memora (`ldtr`, `idtr`, `esi`, `edi` y `ebp`) que debe establecer cada AP una vez arrancado para pasar a modo protegido.

Hay que tener en cuenta que en el momento del arranque no contamos con las mismas estructuras de memoria que el compilador ha dado por supuestas: el uso del segmento de datos que ha hecho el compilador no tiene sentido (¡ni siquiera estamos en modo protegido!), por lo que las direcciones de memoria (desplazamientos respecto al segmento de datos) que puede asignar no tienen ninguna utilidad. La estrategia a seguir será establecer el segmento de datos a una dirección fija y conocida (`_init_ap`, el inicio del fragmento de código) al comienzo de la ejecución y acceder a posiciones de memoria relativas a esa dirección. Esta es la razón de poner las variables al principio del código: sólo hay que calcular el número de bytes que ocupa la instrucción de salto y conoceremos la dirección de cada variable respecto a `_init_ap`. Emplearemos algunas definiciones de `xmp.h` para facilitar el trabajo.

xmp.s

```
.sect .da .sect .data
.align 16

#define C16          o16 a16

_init_ap:
```

```
        C16    jmp    _skip_data
_data_area:                         ! Space for variables
_gdtr_data:    .data2 0x0000        !   gdt limit
               .data4 0x00000000    !       addr
_idtr_data:    .data2 0x0000        !   idt limit
               .data4  0x00000000   !       addr
_esi_data:     .data4 0x00000000    !   esi
_edi_data:     .data4 0x00000000    !   edi
_ebp_data:     .data4 0x00000000    !   ebp
_skip_data:
        ! Mark life_flag to tell BSP we are running
        C16    cli                  ! safe from interrupts
        C16    mov    ax,    cs
        C16    mov    ds,    ax     ! ds= cs (_init_ap)

        ! Prepare environment to jump into protected mode
        C16    lgdt   ( [ TR_GDTR_OFFSET ] )
        C16    lidt   ( [ TR_IDTR_OFFSET ] )

        ! Into protected mode
               mov    eax,   cr0
               orb    al,    1
               mov    cr0,   eax
        ! Far jmp to start with 32 bit execution.
        ! Jump to a .text CS-addressed point
        C16    jmpf   CS_SELECTOR:_trampoline_pm
_end_init_ap:
        hlt
```

El código establece los registros `ldtr` e `idtr` a los valores adecuados y realiza el paso a modo protegido. Este paso no es efectivo hasta que no se produce un salto largo, que se realiza a una dirección correctamente conocida por el compilador, correspondiente al código de la segunda fase de arranque: `_trampoline_pm`..

xmp.h

```
/* Offsets from trampoline start to data areas */
#define TR_GDTR_OFFSET            _gdtr_data - _init_ap
#define TR_IDTR_OFFSET            _idtr_data - _init_ap
```

5.4.6.2. Arranque en modo protegido

En la segunda fase del arranque, ya en modo protegido, se establecen las estructuras de memoria y se prepara el entorno para comenzar a ejecutar normalmente el código C. La estrategia a seguir es utilizar en todos los procesadores las mismas estructuras de memoria que las establecidas en el arranque para el BSP. Cada AP podría repetir la secuencia de inicialización que ya realizó el BSP, pero eso obligaría a separar, de ese código original de inicialización, las partes cuya ejecución se debe repetir de las que no se debe repetir (como la construcción de las estructuras de memoria). Siguiendo la estrategia de mantener la estructura original de Minix, se ha optado por clonar el contenido de los registros del BSP en cada AP (segmentos, descriptores, etc.), excepto aquéllos que deben ser independientes (TSS, pila), que deben establecerse según el número de procesador.

La rutina `trampoline_pm()`, localizada en `xmp.s`, es la encargada de realizar todas estas operaciones. Se llega a esta rutina a través del salto largo que efectúa el AP al final del código de arranque en modo real para activar el modo protegido. Así, en este punto (`trampoline_pm`) ya nos encontramos ejecutando en modo protegido. La primera acción es establecer los segmentos de datos al mismo valor que el BSP (ya que van a acceder a las mismas variables). Lo mismo con el resto de registros importantes, cuyo valor se almacenó en la zona de variables del código de arranque del AP.

Para el caso del registro de tareas, el segmento de estado de tarea (TSS, *Task State Segment*) debe ser independiente para cada procesador. Así, cada AP debe establecer su propio registro a través de un descriptor. La explicación de este descriptor se estudiará en la sección 5.8.4.

Otra tarea a realizar por el AP es la de activar su sistema de cache, inhabilitado por defecto en el momento de arrancar, para lo cual se invoca la función `enable_cache`, que se describe en el apartado 5.13.

Previamente a la invocación de esta función hay que establecer la pila para el AP. Cada procesador debe contar con su propia pila para ejecutar dentro del núcleo. La pila del núcleo es un espacio de memoria de 1024 bytes por cada procesador. Una macro `SET_CPU_STK` establece el puntero de pila a la posición adecuada de este espacio en función del número de procesador que se obtiene mediante otra macro `THIS_CPU`. Ambas macros se estudiarán detalladamente en la sección 5.5.

Una vez listas todas estas estructuras, pasamos a la tercera fase del arranque, escrita ya en C, mediante la invocación de la función `ap_main()`, localizada en el fichero `mp.c`. La última instrucción (`hlt`) nunca debería alcanzarse.

xmp.s

```
.sect .text
.align 16

_trampoline_pm:
       ! We are in protected mode. Load AP registers as in BSP
              mov    ax,    DS_SELECTOR
              mov    ds,    ax              ! load DS
              mov    ss,    ax              ! load SS
              mov    es,    ax              ! load ES
              mov    fs,    ax              ! load FS
              mov    gs,    ax              ! load GS
              mov    eax,   ( _esi_data )
              mov    esi,   eax             ! load ESI
              mov    eax,   ( _edi_data )
              mov    edi,   eax             ! load EDI
              mov    eax,   ( _ebp_data )
              mov    ebp,   eax             ! load EBP
       ! Load TSS of this ap
              THIS_CPU(eax)
              dec    eax                    ! AP# less 1
              shl    eax,   3               ! mult DESC_SIZE
              add    eax,   TSS_FIRST_AP_SELECTOR
              ltr    ax                     ! load TSS
       ! Now we are ready to address as usual and execute normal
       ! kernel code, so, lets go
       ! Each CPU needs its own stack space inside kernel stack
       ! Make esp point to cpus stack top
              THIS_CPU(eax)
              SET_CPU_STK(eax)
       ! Enable AP cache
              call   _enable_cache
       ! Continue in C code
              call   _ap_main
hlt
```

5.4.6.3. Arranque en C

La tercera fase de arranque tiene lugar en la función `ap_main()`, localizada en el fichero `mp.c`, donde se efectúan los últimos pasos de configuración: hacer saber al BSP que el arranque ha sido correcto, habilitar interrupciones, tomar trabajo y comenzar a ejecutarlo. Tras establecer la variable `ap_running_flag` al valor `AP_LIFE_FLAG_MARK` (el esperado por el BSP), sincronizamos el AP con el resto de procesadores, bloqueando

el cerrojo principal del núcleo (`lock_mp_kernel()`). El BSP cerró este cerrojo antes de comenzar el arranque multiprocesador, por lo que todos los AP quedarán detenidos en este punto hasta que el BSP libere el cerrojo, lo que ocurrirá tan pronto finalice el arranque multiprocesador, invoque el *dispatcher* (`restart`) y abandone el núcleo. Entonces sólo uno de los AP adquirirá el cerrojo y podrá continuar.

El siguiente paso de cada AP es habilitar las interrupciones en su APIC local y marcarse a sí misma como disponible en el sistema, lo que se realiza mediante la función `enable_cpu()` que estudiaremos en el apartado 5.12. `this_cpu` identifica al procesador que ejecuta el código, como veremos en el siguiente apartado, por lo que la llamada a `enable_cpu(this_cpu,...)` hace que el procesador se habilite a sí mismo.

A continuación se invoca la función `lock_pick_proc()` para que el planificador asigne una tarea preparada al puntero de proceso (`proc_ptr`) correspondiente al AP, de manera que la invocación al *dispatcher* (`restart()`) restablezca los registros de ese proceso, el AP abandone el núcleo, comience a ejecutar el proceso y libere el cerrojo del núcleo, dado paso al siguiente AP. Obsérvese la suposición de que la llamada a `restart()` nunca retorna.

mp.c

```
void ap_main(void) {
/* This is the main routine for AP's. This is the entry point before
   the CPU has been started and jumped to protected mode */

   /* Tell BSP we are running */
   ap_running_flag=AP_LIFE_FLAG_MARK;

   /* Now we're ready to take some work. We find any task and call
      restart() to execute it (or to idle), but we must synchonize
      other cpus before enter kernel code */
   lock_mp_kernel(); /* restart() will unlock later */

   /* Enable APIC interrupt aceptance */
   enable_cpu(this_cpu, WITHOUT_ECHO);

   /* Now, kernel is our! */
   lock_pick_proc(); /* Is there any work? */
   restart();        /* Let's go... */
}
```

5.5. Identificación de procesadores

La replicación de estructuras anunciada en los principios de diseño tiene como objetivo que cada procesador acceda a su propia información sin interferir en la del resto de procesadores. Cada estructura replicada se convierte en un vector (*array*) con tantos elementos como procesadores, donde cada procesador accede a su índice. Así cada procesador tiene asignado un número único que debe poder conocer en cualquier momento. La forma de identificar cada procesador es a través del registro *Local APIC ID Register* (IDR) de su APIC local.

La necesidad de identificación ocurre en algunos puntos críticos de código, como el cambio de contexto, donde la eficiencia del código es crucial. Por esta razón, la implementación de la identificación de procesadores se ha cuidado especialmente y se ha realizado en ensamblador procurando una optimización máxima.

Partimos de una macro de ensamblador que proporciona acceso al IDR y devuelve el valor de identificación del APIC (o sea, del procesador) en un registro que se pasa por parámetro. Esta macro se llama `THIS_CPU_SHORT` y se encuentra definida en el fichero `xmp.h`.

xmp.h

```
#define THIS_CPU_SHORT(reg)                         ;\
              mov    edx,   (_local_apic_base)     ;\
              add    edx,   0x20                   ;\
              mov    reg,   FLAT_DS_SELECTOR       ;\
              mov    ds,    reg                    ;\
              mov    reg,   (edx)                  ;\
              and    reg,   0x0F000000             ;\
              shr    reg,   6*4
```

Este código necesita modificar el segmento de datos para poder acceder a direcciones fuera del espacio del núcleo. En la mayoría de ocasiones es necesario restaurar el segmento previamente definido para continuar ejecutando. Por esta razón se define otra macro `THIS_CPU`, basada en la anterior, que restaura el segmento de datos al que utiliza normalmente el núcleo una vez obtenida la identificación.

xmp.h

```
#define THIS_CPU(reg)                               ;\
              THIS_CPU_SHORT(reg)                   ;\
              mov    edx,   DS_SELECTOR             ;\
              mov    ds,    dx
```

`THIS_CPU` proporciona la identificación del procesador en un registro de propósito general (`eax`, `ebx` o `ecx`; `edx` no se puede utilizar ya que se emplea dentro del desarrollo de la macro).

Para utilizar `THIS_CPU` desde código C se define la macro `this_cpu` en `xmp.h`, con dos variantes, una para un núcleo multiprocesador, que invoca una función, y otra para monoprocesador, que siempre vale 0.

xmp.h

```
#if (ENABLE_MP == 1)
#define this_cpu      f_this_cpu()
#else
#define this_cpu      0
#endif
```

La versión multiprocesador invoca a la función `f_this_cpu()`, definida en `xmp.s`, que incluye desde ensamblador la macro THIS_CPU_SHORT haciendo una salvaguarda del registro `edx` y del segmento de datos.

zmp.s

```
! Returns APIC id of curret cpu (0..n-1)
_f_this_cpu:
       o16    push   ds
              push   edx
              THIS_CPU_SHORT(eax)
              pop    edx
       o16    pop    ds
              ret
```

5.6. Pila del núcleo

La pila del núcleo no debe ser compartida por los distintos procesadores. La pila original es una zona de memoria de tamaño K_STACK_BYTES (1024 bytes) que comienza en k_stack y termina en k_stktop, definida en mpx386.s[6]. Para el uso multiprocesador se multiplica este tamaño por el número de procesadores.

mpx386.s

```
k_stack:
       .space K_STACK_BYTES * MP_NR_CPUS
                           ! kernel stack for all cpus
k_stktop:                  ! top of kernel stack
```

En el núcleo original, conmutar a la pila del núcleo consistía en hacer apuntar el puntero de pila (esp) a la cima de la pila (k_stktop). Con el núcleo multiprocesador, el espacio de pila está dividido en múltiples partes de K_STACK_BYTES bytes cada una. La estrategia para conmutar a la pila del núcleo para cada procesador es hacer que el puntero de pila apunte a k_stktop menos K_STACK_BYTES × número de procesador, para lo que se define una macro de ensamblador SET_CPU_STK en xmp.h, que cambia a la pila correspondiente al procesador que indique el registro pasado como parámetro (no puede ser edx).

xmp.h

```
#define       K_STACK_BYTES_SH     10       /* log2 K_STACK_BYTES */
                                   /* K_STACK_BYTES MUST BE POWER OF 2 */

#if (K_STACK_BYTES != (1<<K_STACK_BYTES_SH))
#error "ERROR: K_STACK_BYTES_SH is not log2(K_STACK_BYTES)"
#endif

/* This secuence sets kernel stack to the top of kernel stack
   depending of current cpu number stored in <reg>
   Registers <reg>, EDX, ESP are afected
   <reg> can be EAX, EBX, ECX */

#define SET_CPU_STK(reg)                          ;\
              shl    reg,   K_STACK_BYTES_SH     ;\
              mov    edx,   k_stktop             ;\
              sub    edx,   reg                  ;\
              mov    esp,   edx
```

Obsérvese que la multiplicación por K_STACK_BYTES se hace mediante un desplazamiento a la izquierda de K_STACK_BYTES_SH bits. Si se modificara el tamaño para la pila del núcleo habría que cambiar también el valor de K_STACK_BYTES_SH. Además es necesario que el nuevo tamaño sea potencia perfecta de 2, para poder seguir utilizando el desplazamiento como producto, condición que se comprueba tras la definición de K_STACK_BYTES_SH, generándose un error de compilación si no se cumple.

5.7. Implementación de cerrojos

Como ya hemos visto en secciones previas, la protección de las regiones críticas se ha realizado mediante el uso de cerrojos (*spinlock*) globales (en la memoria compartida) *[18]*. Conceptualmente hemos

[6] En la versión 32 bits de mpx.s, la única tenida en cuenta en esta versión de Minix SMP.

hablado de unas primitivas LOCK, UNLOCK que manipulan semáforos binarios que impiden la entrada de un procesador a una región crítica en cuyo interior se encuentre otro procesador, de manera que el procesador bloqueado queda detenido en una espera activa hasta que el propietario libere el cerrojo.

Para la implementación de los cerrojos en la memoria compartida se han aprovechado las instrucciones atómicas (`bts`, `btr`), así como el prefijo de bloqueo del bus (`lock`) que proporciona la arquitectura Intel.

Los cerrojos, al igual que la identificación de procesadores, se utilizan en situaciones donde la eficiencia es crucial, como el cambio de contexto para la entrada y salida del núcleo, por lo que también se ha cuidado especialmente su diseño. Se han implementado utilizando macros de ensamblador.

Las primitiva de bloqueo y desbloqueo son macros definidas en el fichero `xmp.h` de forma condicional al tipo de núcleo (monoprocesador o multiprocesador). En el núcleo monoprocesador las primitivas de cerrojo se emplean únicamente a modo de banderas (sustituyendo a la asignación de valores a la variable `switching`), por lo que su definición se simplifica notablemente.

xmp.h

```
#if ( ENABLE_MP == 1 )

#define MP_LOCK(semaphore_var)                          ;\
0:            clc                                       ;\
       lock   bts    (semaphore_var),      0            ;\
              jnc    2f                                  ;\
1:            cmp    (semaphore_var),MP_LOCK_FREEVAL     ;\
              jne    1b                                  ;\
              jmp    0b                                  ;\
2:

#define MP_UNLOCK(semaphore_var)                        ;\
       lock   btr    (semaphore_var),      0


#else


#define MP_LOCK(semaphore_var)                          ;\
              mov    (semaphore_var),      1

#define MP_UNLOCK(semaphore_var)                        ;\
              mov    (semaphore_var),      0

#endif
```

Ambas macros aceptan como parámetro una variable, la que debe almacenar el estado del cerrojo, 0 si está libre, y 1 si está cerrado. Esta variable debe estar alineada en párrafos de 4 bytes para que se asegure la atomicidad. El funcionamiento de la macro `MP_LOCK` es el siguiente:

1. Establecer el cerrojo a 1
2. Si antes estaba a 0, terminar (cerrojo adquirido)
3. Si no, esperar hasta que esté a 0 y volver a empezar.

Obsérvese que la espera activa se hace consultado de forma normal el valor de la variable, y no con la instrucción atómica y el bloqueo de bus. Puesto que esta comparación se hace de forma constante y muy rápida, si se bloqueara el bus en cada consulta, se retrasaría la actividad en los demás procesadores. Por ello, la espera activa se hace con instrucciones no atómicas, pero cuando se detecta la liberación del cerrojo es necesario empezar de nuevo, forzando otra vez el uso de la instrucción atómica.

La liberación mediante la macro `MP_UNLOCK` se hace usando la instrucción atómica `btr` y el bloqueo del bus.

Estos cerrojos se utilizan en el código ensamblador del cambio de contexto (fichero `mpx386.s`), como veremos en el apartado 5.10. Además, es necesario utilizar los cerrojos desde código C. Para ello se definen varias funciones (en el fichero `xmp.s`) que utilizan las macros de ensamblador:

- `lock_mp_kernel()` bloquea el cerrojo del núcleo.
- `unlock_mp_kernel()` libera el cerrojo del núcleo.
- `mp_switching_lock()` bloquea el cerrojo del planificador.
- `mp_switching_unlock()`libera el cerrojo del planificador.
- `mp_switching_islocked()`consulta el estado del cerrojo del planificador.

xmp.s

```
! This is the same that use the assembler macro MP_LOCK(k_main_lock)
!    but callable from C code
_lock_mp_kernel:
       MP_LOCK(_k_main_lock)
       ret

! This is the same that use the assembler macro MP_UNLOCK(k_main_lock)
!    but callable from C code
_unlock_mp_kernel:
       MP_UNLOCK(_k_main_lock)
       ret

! locks a multiprocessor-safe semaphore for switching jobs
.align 4
switching_data:      .data4 MP_LOCK_FREEVAL
_mp_switching_lock:
       MP_LOCK(switching_data)
       ret


! unlocks a multiprocessor-safe semaphore for switching jobs
_mp_switching_unlock:
       MP_UNLOCK(switching_data)
       ret

! Tell is switching lock is locked or not
_mp_switching_islocked:
       mov    eax,   (switching_data)
       xor    eax,   MP_LOCK_FREEVAL
       ret
```

5.8. Replicación de estructuras

Algunas de las estructuras que ha sido necesario replicar para cada procesador son las siguientes: el puntero de proceso (`proc_ptr`), que indica el proceso en ejecución; el puntero de facturación (`bill_ptr`), que indica el proceso al que hay que computar el tiempo de ejecución; el contador de reentradas al núcleo (`k_reenter`), que diferencia entre una entrada y una reentrada. Estas modificaciones implican toda una serie de cambios en todos los puntos donde se referencian estas variables, incluidas algunas tareas, por ejemplo, *clock*, el manejador del reloj, donde se manipula información relativa a la tarificación. En la tarea del reloj también ha sido necesario replicar la estructura `prev_ptr`, que almacena el puntero al proceso anteriormente planificado.

Otras estructuras que ha habido que replicar vienen impuestas por la propia arquitectura de Intel. Por ejemplo, es necesario que cada procesador disponga de su propio espacio de pila para ejecutar código del núcleo, incluido el código previo a la llegada al cerrojo principal. También es necesario reproducir para cada procesador la estructura TSS (*Task State Segment*) imprescindible para el cambio de contexto en modo protegido. Por último, también se necesita que cada procesador disponga de su propia tarea IDLE, de manera que cuando dos o más procesadores estén ociosos, no ocupen el mismo descriptor de proceso .

5.8.1. Variable `proc_ptr`

`proc_ptr` apunta a la entrada de la tabla de procesos que corresponde al proceso que está actualmente en ejecución. Es una variable global que se modifica en el planificador (en `pick_proc()`) y que se consulta en numerosos puntos del núcleo. Al existir múltiples procesadores, existen múltiples procesos ejecutándose al mismo tiempo, por lo que `proc_ptr` se convierte en un vector de punteros, uno para cada procesador.

En el planificador, cada procesador accederá a su entrada del vector correspondiente, ya que cada procesador toma su propio trabajo. Otros accesos a esta variable se han encontrado en algunas tareas de E/S y en el *dispatcher*.

En el cambio de contexto, `proc_ptr` se utiliza para conocer el contexto del proceso que hay que restaurar para continuar ejecutándolo tras la salida del núcleo. Aquí de nuevo cada procesador tomará el contexto de su propio proceso, accediendo a su índice del vector.

En las tareas de E/S, una consulta a `proc_ptr` no proporciona otra cosa que la dirección de la entrada de la tabla de procesos correspondiente a esa tarea, puesto que en el momento de consultarse, apunta al proceso en ejecución que no es otro que la propia tarea. Por lo tanto, en el entorno multiprocesador debe consultarse la tarea del mismo procesador que la ejecuta.

En conclusión, `proc_ptr` pasa de ser un puntero a un vector de punteros, y cada acceso a `proc_ptr` en el núcleo se convierte en un acceso a **`proc_ptr[this_cpu]`**. Para mejorar el rendimiento, en las funciones en que se consulta más de una vez el valor de `this_cpu`, se almacena el valor de `this_cpu` en una variable y luego se indexa mediante la variable.

`proc_ptr` está definido en el fichero `glo.h`.

glo.h

```
EXTERN struct proc *proc_ptr[MP_NR_CPUS];
                            /* pointer to currently running process */
```

Se han encontrado referencias a `proc_prt` en los siguientes archivos del núcleo y tareas de E/S: `dp8390.c`, `driver.c`, `drvlib.c`, `exception.c`, `mcd.c`, `sb16_dsp.c` y `mpx386.s`.

5.8.2. Variables `bill_ptr` y `prev_ptr`

`bill_ptr` apunta a la entrada de la tabla de procesos que corresponde al proceso al que actualmente se le carga la tarificación de tiempo de CPU. Coincide con el valor de `proc_ptr` cuando éste se refiere a un proceso de usuario, pero cuando se está ejecutando un servidor o tarea de E/S (excepto `IDLE`), `bill_ptr` apunta al último proceso de usuario que se ha ejecutado, ya que es a éste al que se le deben cargar el tiempo de sistema que se está consumiendo, probablemente en una llamada al sistema que él mismo invocó.

`bill_ptr` se manipula en dos puntos: el planificador y la tarea del reloj:

- En el planificador, se establece su valor cuando se planifica una nuevo proceso (en `pick_proc()`). Cada procesador debe establecer su propio proceso, por tanto también el proceso tarificado.

- En la tarea del reloj se realiza la tarificación, para lo que se consulta el valor de `bill_ptr`. Otra variable llamada `prev_ptr` mantiene el valor anterior de `bill_ptr`, por lo que su uso va a ser paralelo. Las modificaciones necesarias para estas variables en la tarea del reloj serán estudiadas en profundidad al revisar la tarea del reloj en el apartado 5.14.

Además de estos dos puntos, `bill_ptr` se inicializa apuntando a `IDLE` en `main.c`, durante el arranque del sistema, donde habrá que ampliar la iniciación a todos los índices del array. Originalmente esta inicialización se produce casi al final de `main()`, justo antes de invocar `pick_proc()`. En el nuevo núcleo se ha adelantado esta inicialización y se ha unificado con el de otras variables (como `proc_addr_IDLE`, que se describe en el punto 5.15).

5.8.3. Variable k_reenter

La variable `k_reenter` se define en el fichero `glo.h` y su cometido es mantener el contador de reentradas al núcleo. Su valor fuera del núcleo es –1 (255 si se interpreta como byte). Durante una entrada al núcleo valdrá 0, y durante una reentrada, algún valor mayor de 0 (¡ojo con valor 255!).

glo.h

```
EXTERN unsigned char k_reenter[MP_NR_CPUS];
                            /* kernel reentry count (entry count less 1)*/
```

Puesto que tenemos varios procesadores, cada uno puede estar dentro o fuera del núcleo, así que es necesario replicar la información. Normalmente cada procesador va a acceder exclusivamente a su índice (`k_reenter[this_cpu]`). Dentro del núcleo no tiene sentido que un procesador acceda al contador de los demás procesadores, puesto que hemos puesto un cerrojo a la entrada y sólo un procesador puede estar dentro del núcleo.

Sí debemos reseñar el uso de `k_reenter` en la tarea del reloj, descrita en la sección 5.14.

5.8.4. Estructura TSS

El cambio de contexto de la arquitectura Intel guarda algunos registros en una estructura llamada TSS (*Task State Segment*). Para evitar la interferencia entre los cambios de contexto de los diferentes procesadores, cada uno debe contar con su propia estructura.

La estructura TSS se configura en el procesador mediante la carga de un registro de la CPU, lo que obliga a que la dirección del TSS esté incluida entre los descriptores de la GDT a la hora de estructurar la memoria en modo protegido. Es decir, hay que incluir en la GDT un descriptor por cada TSS que se vaya a utilizar, junto con los descriptores del segmento de datos, de código, de pila, etc.

Estas definiciones se incluyen en el fichero `protect.h`. En primer lugar hay que definir el número de entrada de la GDT correspondiente a los TSS de los AP. En la GDT aparecen los descriptores de diversos segmentos del núcleo, seguidos de los segmentos correspondientes a las LDT de los procesos. Se ha optado por añadir las entradas de las TSS al final de los descriptores del núcleo y antes de las LDT, ya que implica pocas modificaciones. La macro `TSS_AP_INDEX(n)` identifica la entrada para el AP número *n*, numerados como 1, 2, 3, etc. (0 correspondería a un BSP y no está permitido). Al añadirse entradas intermedias, hay que redefinir la macro `FIRST_LDT_INDEX` que identifica la primera entrada correspondiente a una LDT.

Finalmente habría que hacer la definición `TSS_AP_SELECTOR(n)`, que identificaría el selector que hay que copiar al registro `TR` (*task register*) para establecer la TSS. La forma correcta de hacerlo sería, siguiendo la estrategia del código original de Minix, definirlo como `(TSS_AP_INDEX(n) * DESC_SIZE)`, pero esta definición se utiliza en código ensamblador (concretamente para los AP en la rutina `_trampoline_pm` en

xmp.s que hemos visto anteriormente) y el ensamblador no puede preprocesar la expresión correctamente, por lo que se le debe dar precalculada. Así, para añadir un número arbitrario de procesadores habría que hacer un conjunto amplio de definiciones precalculadas (una por procesador) que luego no podrían utilizarse en bucles ni de forma genérica (basada en índices o valores de variables). Por esta razón se ha optado por especificar únicamente el valor del primer descriptor (TSS_FIRST_AP_SELECTOR), dejando para el código ensamblador el cálculo para el enésimo procesador (ver la rutina _trampoline_pm en xmp.s).

protect.h

```
/* Fixed global descriptors.  1 to 7 are prescribed by the BIOS. */
#define GDT_INDEX        1  /* GDT descriptor */
#define IDT_INDEX        2  /* IDT descriptor */
#define DS_INDEX         3  /* kernel DS */
#define ES_INDEX         4  /* kernel ES (386: flag 4 Gb at startup) */
#define SS_INDEX         5  /* kernel SS (386: monitor SS at startup) */
#define CS_INDEX         6  /* kernel CS */
#define MON_CS_INDEX     7  /* temp for BIOS (386: monitor CS at startup) */
#define TSS_INDEX        8  /* kernel TSS */
#define DS_286_INDEX     9  /* scratch 16-bit source segment */
#define ES_286_INDEX    10  /* scratch 16-bit destination segment */
#define VIDEO_INDEX     11  /* video memory segment */
#define DP_ETH0_INDEX       12     /* Western Digital Etherplus buffer */
#define DP_ETH1_INDEX       13     /* Western Digital Etherplus buffer */
#define TSS_AP_INDEX(n)     (13+n) /* TSS for AP 1,2,3... */

#define FIRST_LDT_INDEX (13+MP_NR_CPUS)
                            /* rest of descriptors are LDT's */

#define GDT_SELECTOR      0x08     /* (GDT_INDEX * DESC_SIZE) bad for asld */
#define IDT_SELECTOR      0x10     /* (IDT_INDEX * DESC_SIZE) */
#define DS_SELECTOR       0x18     /* (DS_INDEX * DESC_SIZE) */
#define ES_SELECTOR       0x20     /* (ES_INDEX * DESC_SIZE) */
#define FLAT_DS_SELECTOR  0x21     /* less privileged ES */
#define SS_SELECTOR       0x28     /* (SS_INDEX * DESC_SIZE) */
#define CS_SELECTOR       0x30     /* (CS_INDEX * DESC_SIZE) */
#define MON_CS_SELECTOR   0x38     /* (MON_CS_INDEX * DESC_SIZE) */
#define TSS_SELECTOR      0x40     /* (TSS_INDEX * DESC_SIZE) */
#define DS_286_SELECTOR   0x49     /* (DS_286_INDEX * DESC_SIZE + 1) */
#define ES_286_SELECTOR   0x51     /* (ES_286_INDEX * DESC_SIZE + 1) */
#define VIDEO_SELECTOR    0x59     /* (VIDEO_INDEX * DESC_SIZE + 1) */
#define DP_ETH0_SELECTOR  0x61     /* (DP_ETH0_INDEX * DESC_SIZE) */
#define DP_ETH1_SELECTOR  0x69     /* (DP_ETH1_INDEX * DESC_SIZE) */
#define TSS_FIRST_AP_SELECTOR  0x70 /* TSS_AP_INDEX(1) * DESC_SIZE) */
```

La definición, asignación e iniciación de los descriptores para la TSS se encuentran en protect.c. En primer lugar hay que replicar la estructura que almacena en memoria la TSS, llamada tss.

protect.c

```
PUBLIC struct tss_s tss[MP_NR_CPUS];       /* zero init */
```

Finalmente es necesario indexar todas las referencias a tss que se encuentran en el código, todas ellas en la rutina prot_init() en protect.c. En algunos puntos hay que repetir la misma operación de iniciación para todas las TSS (independientemente del número de procesador), mientras que en otros casos, hay que hacer referencia al número de entrada de la GDT, por lo que hay que distinguir entre el BSP y los AP (ya que se ha decidido poner las entradas de la TSS de los AP con una macro diferente de la original).

prot_init() en protect.c

```
...
...

/* Build main TSS.
   * This is used only to record the stack pointer to be used after an
   * interrupt.
   * The pointer is set up so that an interrupt automatically saves the
   * current process's registers ip:cs:f:sp:ss in the correct slots in the
   * process table.
   */
  FOR_EACH_CPU(cpu)
     tss[cpu].ss0 = DS_SELECTOR;


  init_dataseg(&gdt[TSS_INDEX], vir2phys(&(tss[0])),
                    (phys_bytes) sizeof (struct tss_s), INTR_PRIVILEGE);

  FOR_EACH_AP(cpu)
     init_dataseg(&gdt[TSS_AP_INDEX(cpu)], vir2phys(&(tss[cpu])),
                    (phys_bytes) sizeof (struct tss_s), INTR_PRIVILEGE);

  gdt[TSS_INDEX].access = PRESENT | (INTR_PRIVILEGE << DPL_SHIFT) | TSS_TYPE;
  FOR_EACH_AP(cpu)
     gdt[TSS_AP_INDEX(cpu)].access = PRESENT | (INTR_PRIVILEGE << DPL_SHIFT) | TSS_TYPE;

  /* Build descriptors for interrupt gates in IDT. */
  for (gtp = &gate_table[0];
       gtp < &gate_table[sizeof gate_table / sizeof gate_table[0]]; ++gtp) {
       int_gate(gtp->vec_nr, (phys_bytes) (vir_bytes) gtp->gate,
               PRESENT | INT_GATE_TYPE | (gtp->privilege << DPL_SHIFT));
  }
  int_gate(SYS_VECTOR, (phys_bytes) (vir_bytes) p_s_call,
          PRESENT | (USER_PRIVILEGE << DPL_SHIFT) | INT_GATE_TYPE);
  int_gate(LEVEL0_VECTOR, (phys_bytes) (vir_bytes) level0_call,
          PRESENT | (TASK_PRIVILEGE << DPL_SHIFT) | INT_GATE_TYPE);

#if _WORD_SIZE == 4
  /* Complete building of main TSS. */
  FOR_EACH_CPU(cpu)
       tss[cpu].iobase = sizeof(struct tss_s);   /* empty i/o permissions map */

...
...
```

5.8.5. Tarea IDLE

La tarea IDLE necesita ser replicada para evitar interferencias entre los procesadores cuando ambos están ociosos, puesto que la información del cambio de contexto se guarda en el descriptor de proceso. La replicación de IDLE implica un amplio conjunto de cambios en múltiples puntos del código.

Puesto que se trata de una tarea de E/S, el primer paso es añadir una nueva tarea por cada AP, para lo cual hay que incrementar el contador de tareas en el fichero `<minix/const.h>`.

<minix/const.h>

```
/* Number of tasks. */
#define NR_TASKS     (9 + ENABLE_WINI + ENABLE_SCSI + ENABLE_CDROM \
                     + ENABLE_NETWORKING + 2 * ENABLE_AUDIO + \
                     + MP_NR_CPUS -1 )
```

A continuación hay que añadir las tareas en la tabla de tareas `tasktab`, definida en `table.c`.

table.c

```
PUBLIC struct tasktab tasktab[] = {
       { tty_task,          TTY_STACK,      "TTY"        },
#if ENABLE_NETWORKING
       { dp8390_task,              DP8390_STACK, "DP8390"      },
#endif
#if ENABLE_CDROM
       { cdrom_task,        CDROM_STACK,    "CDROM"            },
#endif
#if ENABLE_AUDIO
       { audio_task,        AUDIO_STACK,    "AUDIO"            },
       { mixer_task,        MIXER_STACK,    "MIXER"            },
#endif
#if ENABLE_SCSI
       { scsi_task,         SCSI_STACK,     "SCSI"             },
#endif
#if ENABLE_WINI
       { winchester_task,   WINCH_STACK,    "WINCH"            },
#endif
       { syn_alrm_task,     SYN_ALRM_STACK, "SYN_AL"    },
#if (MP_NR_CPUS > 4)
#error Please, add more IDLE tasks here :-)
#endif
#if (MP_NR_CPUS > 3)
       { idle_task,         IDLE_STACK,     "IDLE3"            },
#endif
#if (MP_NR_CPUS > 2)
       { idle_task,         IDLE_STACK,     "IDLE2"            },
#endif
#if (MP_NR_CPUS > 1)
       { idle_task,         IDLE_STACK,     "IDLE1"            },
#endif
       { idle_task,         IDLE_STACK,     "IDLE"             },
       { printer_task,             PRINTER_STACK,       "PRINTER"     },
       { floppy_task,              FLOP_STACK,   "FLOPPY"      },
       { mem_task,          MEM_STACK,      "MEMORY"     },
       { clock_task,        CLOCK_STACK,    "CLOCK"            },
       { sys_task,          SYS_STACK,      "SYS"        },
       { 0,                 HARDWARE_STACK, "HARDWAR"    },
       { 0,                 0,              "MM"         },
       { 0,                 0,              "FS"         },
#if ENABLE_NETWORKING
       { 0,                 0,              "INET"             },
#endif
       { 0,                 0,              "INIT"             },
};
```

Obsérvese que cada nueva entrada debe establecerse manualmente. Se han introducido entradas para hasta 4 procesadores, generándose un error para un número mayor, error que se soluciona reproduciendo (poco más que *copiar y pegar*) tantas entradas como sea necesario.

Como tenemos más tareas necesitamos que el espacio de pila se incremente. Esta orden la introducimos al declarar el tamaño total de la pila del núcleo, en `table.c`.

table.c

```
#define       TOT_STACK_SPACE              (TTY_STACK + DP8390_STACK + SCSI_STACK + \
       SYN_ALRM_STACK + ((IDLE_STACK)*MP_NR_CPUS) + HARDWARE_STACK + PRINTER_STACK + \
       WINCH_STACK + FLOP_STACK + MEM_STACK + CLOCK_STACK + SYS_STACK + \
       CDROM_STACK + AUDIO_STACK + MIXER_STACK)
```

Al introducir tareas en mitad de la tabla cambian los números asignados a determinadas tareas. Esta numeración se establece en `<minix/com.h>`, y ahora va a depender del número de procesadores del sistema. Además, la numeración de la tarea IDLE para a ser dependiente de un parámetro que identifica el número de procesador (ya que hay una IDLE para cada procesador).

<minix/com.h>

```
...
...

#define WINCHESTER   (SYN_ALRM_TASK - ENABLE_WINI)
                        /* winchester (hard) disk class */

#define SYN_ALRM_TASK     (-7 - MP_NR_CPUS)/* task to send CLOCK_INT messages */

#define IDLE(CPU)      (-7 - (CPU))        /* task to run when there's nothing to run*/

#define PRINTER           -6       /* printer I/O class */

...
...
```

Al existir más de una tarea IDLE es necesario actualizar un conjunto de definiciones que encontramos en `proc.h` que se refieren a las excepciones que caracterizan a las tareas especiales `HARDWARE` e `IDLE`.

proc.h

```
#define NIL_PROC          ((struct proc *) 0)
#define isidlehardware(n) (( ((n)<=IDLE(0)) && ((n)>=IDLE(MP_NR_CPUS-1)) ) || (n) ==
HARDWARE)
#define isoksrc_dest(n)   (isokprocn(n) || (n) == ANY)
#define isoksusern(n)     ((unsigned) (n) < NR_PROCS)
#define isokusern(n)      ((unsigned) ((n) - LOW_USER) < NR_PROCS - LOW_USER)
#define isrxhardware(n)   ((n) == ANY || (n) == HARDWARE)
#define issysentn(n)      ((n) == FS_PROC_NR || (n) == MM_PROC_NR)
#define istaskp(p)        ((p) < END_TASK_ADDR && ( (p) > proc_addr(IDLE(0)) || (p) <
proc_addr(IDLE(MP_NR_CPUS-1)) ) )
```

Además, para tener un acceso más rápido a la dirección del descriptor de proceso del conjunto de tareas IDLE (que se utiliza en múltiples puntos del código del núcleo) se ha creado un vector de apuntadores precalculados que se inicializa en `main.c` como se indica en el apartado 5.15. Este vector se define en `proc.h`.

proc.h

```
EXTERN struct proc *proc_addr_IDLE[MP_NR_CPUS];  /*precalculated pointers to idle task*/
```

Ahora que tenemos varias tareas IDLE, habrá que hacer que cada procesador ejecute la suya. Basta un pequeño cambio en la función `pick_proc()` del planificador (`proc.c`), que veremos en el apartado 5.11.4.

También hay que hacer este cambio cuando se inician por primera vez (en `main.c`) los punteros `proc_ptr` y `bill_ptr`, de manera que cada procesador tome su propia tarea IDLE (ver apartado 5.15).

5.9. Comunicación entre procesadores

Existen diversas situaciones en las que un procesador debe comunicarse con otro. Para estas situaciones se ha creado un mecanismo de intercomunicación a través de una interrupción entre procesadores (IPI). La invocación del servicio se hace a través de la rutina `interrupt_cpu()`, definida en `mp.c`, que acepta 3 parámetros:

- El número de procesador al que se quiere enviar un mensaje.
- El tipo de mensaje: un parámetro numérico de entre un conjunto de posibles mensajes declarados en `mp.h`.
- Un argumento opcional dependiendo del tipo de mensaje, existiendo el valor especial `MP_CM_ARG_NONE` para cuando no es necesario especificar ninguno.

mp.c

```
void interrupt_cpu(int cpu, int command, int param) {
/* Send a message to a CPU consisting in a command and an optional parameter */

   mp_command[cpu]=command;
   mp_param[cpu]=param;
   forward_vector_to_cpu(VECTOR(16),cpu);
}
```

Los mensajes posibles en la versión actual son 5:

- `MP_CM_DISABLE`: Ordena a un procesador que se inhabilite y deje de ejecutar procesos. Se puede especificar una opción de eco (`WITH_ECHO`/`WITHOUT_ECHO`) para que se muestre o no un mensaje en la consola cuando se produzca la inhabilitación.
- `MP_CM_ENABLE`: Ordena a un procesador que se rehabilite y vuelva a ejecutar procesos. Se puede especificar una opción de eco (`WITH_ECHO`/`WITHOUT_ECHO`) para que se muestre o no un mensaje en la consola cuando se produzca la rehabilitación.
- `MP_CM_SCHED`: Ordena a un procesador que tome un nuevo proceso después de una reordenación de la cola de usuario (invocación a `lock_sched()`).
- `MP_CM_PICKPROC`: Ordena a un procesador que tome un nuevo proceso si se encuentra ocioso.
- `MP_CM_HALT`: Ordena a un procesador que se detenga indefinidamente (como parte del proceso de detención del sistema).

mp.h

```
/* MP interprocessor communication */
#define MP_CM_NONE          0      /* nothig to do */
#define MP_CM_DISABLE       1      /* disable cpu */
#define MP_CM_ENABLE        2      /* enable cpu */
#define MP_CM_SCHED         3      /* schedule and pick a new process to run */
#define MP_CM_PICKPROC      4      /* pick a new process to run */
#define MP_CM_REBOOT        5      /* invoke wreboot(mp_param) */
#define MP_CM_HALT          6      /* stop cpu */

#define MP_CM_ARG_NONE      0      /* no parameter */

void interrupt_cpu(int cpu, int command, int param);
```

Los parámetros del mensaje se guardan en dos vectores de enteros globales declarados en `mp.c` de manera que cada procesador recoge sus comandos y argumentos en su índice correspondiente.

mp.c

```
int mp_command[MP_NR_CPUS]; /* command for cpu to execute in IPI */
int mp_param[MP_NR_CPUS];   /* paremeter of IPI command */
```

La interrupción entre procesadores (IPI) se envía utilizando la rutina `forward_vector_to_cpu()`, definida en `mp.c`. Acepta como parámetro el vector que hay que enviar y el número de procesador de destino. El código de esta función es semejante al de las que enviaban las interrupciones de arranque del segundo procesador: preparar los registros ICR (*Interrupt Command Register*) del APIC local para que éste envíe la interrupción. Como prácticamente todos los campos se deben rellenar con ceros, se ha optimizado el código para ahorrar algunas asignaciones inútiles.

mp.c

```
void forward_vector_to_cpu(u8_t vector, int cpu) {
/* Send a interrupt vector to a CPU */
   u32_t icr_h,icr_l;

   while (DELIVERY_STATUS);
   /* Wait for local APIC to complete previous IPI */

   /* prepare to send IPI */
   icr_h = LOCAL_APIC_READ(LOCAL_APIC_ICR_HIGH);
   icr_l = LOCAL_APIC_READ(LOCAL_APIC_ICR_LOW);

   icr_l &=~0x000CDFFF;  /* clear non-reserved fields */
   icr_h &= 0x00FFFFFF;  /* clear non-reserved fields */

   /* Next inetructions does nothig because the values are all 0's,
      so, skip it for speed */
 /*icr_l |= (DELIVERY_FIXED<<DELIVERY_SHIFT);*/  /* stablish FIXED (000) */
 /*icr_l &= ~(1 << TRIGGER_SHIFT);*/             /* trigger = edge */
 /*icr_l |= (PHYSICAL_DEST << DEST_MODE_SHIFT);*/       /* destination = physical */
 /*icr_l |= (DEST_FIELD << DEST_SHORT_SHIFT);*/  /* destination by field */

   icr_l |= vector;                          /* vector field */
   icr_h |= (cpu << DEST_FIELD_SHIFT);           /* cpu to interrupt */

   /* Clear previous error???? We dont hand it, so skip for speed */
   /*apic_error_status();*/

   /* send the IPI */
   LOCAL_APIC_WRITE(LOCAL_APIC_ICR_HIGH, icr_h);
   LOCAL_APIC_WRITE(LOCAL_APIC_ICR_LOW,  icr_l);
}
```

Para el envío de la IPI se ha decidido utilizar el vector correspondiente a un hipotético IRQ 16, (el siguiente al último utilizado por el PIC 8259). Es, por tanto, necesario, definir un nuevo manejador de interrupción para este nuevo vector.

Los vectores de interrupciones hardware se declaran en el fichero `protect.c`, donde aparece una tabla con todos los vectores de interrupción manejados. Aquí se añade una nueva entrada para el vector correspondiente al IRQ 16 con 3 campos: dirección del manejador (la rutina `mp_IPI()`), vector asociado y nivel de privilegio del manejador (`INTR_PRIVILEGE`, para tener todos los privilegios).

protect.c

```
gate_table[] = {
       divide_error, DIVIDE_VECTOR, INTR_PRIVILEGE,
       single_step_exception, DEBUG_VECTOR, INTR_PRIVILEGE,
       nmi, NMI_VECTOR, INTR_PRIVILEGE,
       breakpoint_exception, BREAKPOINT_VECTOR, USER_PRIVILEGE,
       overflow, OVERFLOW_VECTOR, USER_PRIVILEGE,
       bounds_check, BOUNDS_VECTOR, INTR_PRIVILEGE,
       inval_opcode, INVAL_OP_VECTOR, INTR_PRIVILEGE,
       copr_not_available, COPROC_NOT_VECTOR, INTR_PRIVILEGE,
       double_fault, DOUBLE_FAULT_VECTOR, INTR_PRIVILEGE,
       copr_seg_overrun, COPROC_SEG_VECTOR, INTR_PRIVILEGE,
       inval_tss, INVAL_TSS_VECTOR, INTR_PRIVILEGE,
       segment_not_present, SEG_NOT_VECTOR, INTR_PRIVILEGE,
       stack_exception, STACK_FAULT_VECTOR, INTR_PRIVILEGE,
       general_protection, PROTECTION_VECTOR, INTR_PRIVILEGE,
#if _WORD_SIZE == 4
       page_fault, PAGE_FAULT_VECTOR, INTR_PRIVILEGE,
       copr_error, COPROC_ERR_VECTOR, INTR_PRIVILEGE,
#endif
       { hwint00, VECTOR( 0), INTR_PRIVILEGE },
       { hwint01, VECTOR( 1), INTR_PRIVILEGE },
       { hwint02, VECTOR( 2), INTR_PRIVILEGE },
       { hwint03, VECTOR( 3), INTR_PRIVILEGE },
       { hwint04, VECTOR( 4), INTR_PRIVILEGE },
       { hwint05, VECTOR( 5), INTR_PRIVILEGE },
       { hwint06, VECTOR( 6), INTR_PRIVILEGE },
       { hwint07, VECTOR( 7), INTR_PRIVILEGE },
       { hwint08, VECTOR( 8), INTR_PRIVILEGE },
       { hwint09, VECTOR( 9), INTR_PRIVILEGE },
       { hwint10, VECTOR(10), INTR_PRIVILEGE },
       { hwint11, VECTOR(11), INTR_PRIVILEGE },
       { hwint12, VECTOR(12), INTR_PRIVILEGE },
       { hwint13, VECTOR(13), INTR_PRIVILEGE },
       { hwint14, VECTOR(14), INTR_PRIVILEGE },
       { hwint15, VECTOR(15), INTR_PRIVILEGE },
       { mp_IPI,  VECTOR(16), INTR_PRIVILEGE },
  };
```

`mp_IPI()` es una función definida en ensamblador en el fichero `xmp.s`, semejante a las empleadas por los manejadores de las interrupciones E/S procedentes del PIC 8259, excepto por la ausencia del código que reconoce la interrupción escribiendo el comando correspondiente en los registros del PIC. Invoca a la rutina save para que guarde los registros, realice la entrada sincronizada al núcleo y prepare el retorno de la interrupción por el *dispatcher* (`restart`/`restart1`), restablece las interrupciones, invoca el manejador escrito en C `mp_IPI_c()` y retorna.

xmp.s

```
! Entry point of Interprocessor Interrupts. This is an interrupt handler
! that call the C version of the handler

.align 16
_mp_IPI:
       call   save                /* save interrupted process state */
       sti                        /* enable interrupts          */
       call   _mp_IPI_c           /* run C code of handler      */
       cli                        /* disable interrupts                */
       ret                        /* restart (another) process      */
```

`mp_IPI_c()` se encuentra en `mp.c`. Es la función que contiene el verdadero manejador de la interrupción, cuyo cometido consiste en analizar el código y argumento del mensaje solicitado. Antes de nada reconoce la interrupción procedente del APIC local escribiendo un valor arbitrario en el registro EOI (*End Of Interrupt*) del APIC local. Luego accede al vector del comando de comunicación y dependiendo del código ejecuta diversas acciones, cuyo uso está destinado a solucionar aquellas situaciones en las que se debe invocar alguna función del núcleo que no puede ser ejecutada más que por el propio procesador. Estas situaciones son:

- Habilitación e inhabilitación del procesador. Estas operaciones se realizan, como veremos, escribiendo determinados valores en los registros del APIC local del procesador, al que sólo tiene acceso el propio procesador. Así, un procesador no puede habilitar ni inhabilitar a otros procesadores, sólo puede ordenarles que lo hagan.
- Invocación del planificador para tomar un proceso. Cada procesador toma sus propios procesos, luego para hacer que un procesador tome un nuevo proceso hay que forzarlo a ejecutar por sí mismo la función `lock_pick_proc()`.
- Detención del sistema. La detención del sistema implica la detención del multiprocesador para que se restaure la configuración monoprocesador original, en la que el BSP es el único procesador activo. Independientemente del procesador que inicie la secuencia de apagado, debe ser el BSP el que la ejecute y quede al final como único procesador activo, ordenando a todos los demás que se inhabiliten y detengan.

Las acciones a realizar para los distintos mensajes disponibles son las siguientes:

- `MP_CM_DISABLE`: Invoca la función de inhabilitación del propio procesador con la opción de eco pasada como argumento.
- `MP_CM_ENABLE`: Invoca la función de rehabilitación del propio procesador con la opción de eco pasada como argumento.
- `MP_CM_PICKPROC`: Invoca a `lock_pick_prock()` para que se tome un proceso para ejecutar. Esta invocación puede producir un interbloqueo si en el momento de elevarse la interrupción en el procesador, éste se encontraba ejecutando código del planificador protegido por el cerrojo `switching`. Este mensaje, que no se utiliza en el código actual de Minix SMP, se ha previsto para forzar a un procesador a salir del estado ocioso (IDLE), por lo que sólo va a tener efecto si ese es precisamente el estado del procesador.
- `MP_CM_SCHED`: Este mensaje se emplea para hacer que un procesador, cuyo proceso asignado ha agotado su cuantum, tome un nuevo proceso que ya ha sido planificado (los veremos en la sección 5.11.9). Se trata, por tanto, de invocar `lock_pick_proc()`, con las precauciones necesarias para no producir un interbloqueo, como hemos visto para el mensaje anterior. En este caso, el mensaje sólo tiene sentido si el procesador sigue ejecutando un proceso de usuario (posiblemente el que ha agotado el cuantum), y en esta situación no es posible el interbloqueo puesto que los procesos de usuario no acceden al planificador.
- `MP_CM_REBOOT`: Invoca la función `wreboot()` con la opción de reinicio especificada en el argumento.
- `MP_CM_HALT`: Inhabilita el procesador y sus interrupciones y lo detiene en una espera indefinida.

Al final de `mp_IPI_c` se limpia el código del mensaje, puesto que ya está atendido. Obsérvese que los mensajes entre procesadores no se encolan ni son acumulativos.

mp.c

```
void mp_IPI_c(void) {
/* This is the C section of mp_IPI, the interrupt handler of IPIs used for
   inter-CPU messaging */
```

```
    int cpu;

    LOCAL_APIC_WRITE(LOCAL_APIC_EOI,0); /* AKC interrupt to local APIC */

    cpu=this_cpu;
    switch(mp_command[cpu]) {
       case MP_CM_NONE:
        break;
       case MP_CM_DISABLE:
        disable_cpu(cpu,mp_param[cpu]);
        break;
       case MP_CM_ENABLE:
        enable_cpu(cpu,mp_param[cpu]);
        break;
       case MP_CM_PICKPROC:
         if (proc_ptr[cpu]==proc_addr_IDLE[cpu])
               lock_pick_proc();
        break;
       case MP_CM_SCHED:
        if (isuserp(proc_ptr[cpu])) lock_pick_proc();
        break;
       case MP_CM_REBOOT:
        wreboot(mp_param[cpu]);
        break;
       case MP_CM_HALT:
        halt_cpu(cpu);
        break;
    }
    mp_command[cpu]=MP_CM_NONE;     /* done */
}
```

5.10. Entrada y salida del núcleo. Cambio de contexto

Dedicamos esta sección a analizar los cambios necesarios en los puntos de entrada de interrupciones y llamadas al sistema y en el *dispatcher*. Se trata de las rutinas save, `s_call`, `restart` y `restart1` de las que se ha hablado en numerosas ocasiones a lo largo del trabajo. Todas ellas aparecen en el fichero `mpx386.s`.

Puesto que se trata de puntos críticos de código, de cuyo rendimiento depende en gran medida el rendimiento del sistema, se ha decidido separar dos versiones de estas funciones, una para un núcleo monoprocesador y otra para el núcleo multiprocesador. En los siguientes fragmentos de código se muestran ambas versiones para ilustrar las diferencias (la versión coloreada es la multiprocesador). Se observa en todos los casos que la rutina multiprocesador es notablemente más compleja (hay que tener en cuenta el uso masivo de macros de ensamblador).

5.10.1. Rutina save

save es el punto de entrada para las interrupciones hardware. Cada manejador de interrupción invoca a save entre sus primeras instrucciones. Después reconoce la interrupción al PIC y posteriormente invoca el resto del código del manejador.

La entrada por save tiene tres grandes cambios necesarios:

- En primer lugar es necesario actualizar el contador de reentradas correspondiente al procesador que realiza la nueva entrada. Se trata de un vector de valores de 1 byte. Cargamos en `ebx` el número de procesador y en `esi` la dirección inicial del vector (`_k_reenter`). Realizamos el incremento y aprovechamos el resultado del incremento para la siguiente operación.

- La segunda operación es conmutar a la pila del núcleo, si y sólo si se trata de la primera entrada al núcleo, condición que se da cuando el contador de reentradas que se acaba de incrementar queda en 0. En caso necesario disponemos de la macro `SET_CPU_STK` que realiza la conmutación en función del número de procesador especificado en el valor de un registro, `ebx` en este caso, donde todavía conservamos el número de procesador cargado anteriormente.
- El tercer paso es bloquear la entrada al núcleo con el cerrojo principal con `MP_LOCK(_k_main_lock)`, operación que sólo debe hacerse en la primera entrada. Aprovechamos aquí el salto utilizado para omitir la conmutación a la pila del núcleo.

El resto del código permanece tal cual.

mpx386.s

```
! Save for protected mode.
! This is much simpler than for 8086 mode, because the stack already points
! into the process table, or has already been switched to the kernel stack.

       .align 16

#if (ENABLE_MP == 1)
save:
       cld                  ! set direction flag to a known value
       pushad               ! save "general" registers
    o16       push   ds            ! save ds
    o16       push   es            ! save es
    o16       push   fs            ! save fs
    o16       push   gs            ! save gs
       mov    dx, ss        ! ss is kernel data segment
       mov    ds, dx        ! load rest of kernel segments
       mov    es, dx        ! kernel does not use fs, gs
       mov    eax, esp      ! prepare to return

       THIS_CPU(ebx)
       mov    esi,   _k_reenter
       incb   (ebx)(esi)    ! from -1 if not reentering
       jnz    set_restart1  ! stack is already kernel stack

       SET_CPU_STK(ebx)
       MP_LOCK(_k_main_lock)
                            ! not enter while another cpu inside

       push   _restart      ! build return address for int handler
       xor    ebp, ebp      ! for stacktrace
       jmp    RETADR-P_STACKBASE(eax)

       .align 4
set_restart1:
       push   restart1
       jmp    RETADR-P_STACKBASE(eax)

#else

save:
       cld                  ! set direction flag to a known value
       pushad               ! save "general" registers
    o16       push   ds            ! save ds
    o16       push   es            ! save es
    o16       push   fs            ! save fs
    o16       push   gs            ! save gs
       mov    dx, ss        ! ss is kernel data segment
       mov    ds, dx        ! load rest of kernel segments
       mov    es, dx        ! kernel does not use fs, gs
       mov    eax, esp      ! prepare to return
       incb   (_k_reenter)  ! from -1 if not reentering %%%
       jnz    set_restart1  ! stack is already kernel stack
```

```
        mov    esp, k_stktop
        push   _restart      ! build return address for int handler
        xor    ebp, ebp      ! for stacktrace
        jmp    RETADR-P_STACKBASE(eax)

        .align 4
set_restart1:
        push   restart1
        jmp    RETADR-P_STACKBASE(eax)
#endif
```

5.10.2. Rutina `s_call`

Los problemas que presenta `s_call` son muy parecidos a los de save. Se simplifica levemente puesto que el bloqueo y la conmutación a la pila del núcleo es incondicional, pero se complica puesto que en `s_call` se debe salvaguardar el valor de los registros eax, ebx y ecx ya que en ellos se guarda la información de la llamada al sistema.

Necesitamos utilizar alguno de estos registros para actualizar el contador de reentradas y conmutar a la pila del núcleo ya que se hace en función del número de procesador y la macro `THIS_CPU` no puede invocarse con edx. Y no se puede utilizar la pila (con push/pop) para guardar temporalmente una copia de los demás registros ya que parte de las operaciones a realizar incluyen el cambio de pila.

La solución es guardar uno de los registros (ebx) temporalmente en una variable de memoria (save_ebx). El problema es que la memoria es compartida por todos los procesadores, por lo que antes de acceder a esta variable es necesario cerrar el cerrojo del núcleo, de manera que sólo un procesador la pueda utilizar en un momento dado.

Así, una de las primeras operaciones es el bloqueo del núcleo con `MP_LOCK(_k_main_lock)`. Luego hacemos una copia de ebx y cargamos en este registro el número de CPU. Incrementamos el contador de reentradas y conmutamos a la pila del núcleo de la misma manera que en save (excepto porque no se hace la comprobación de reentrada). Finalmente recuperamos el valor de ebx y continuamos con el código normal.

mpx386.s

```
        .align 16
#if (ENABLE_MP == 1)
_s_call:
_p_s_call:
        cld                 ! set direction flag to a known value
        sub    esp, 6*4     ! skip RETADR, eax, ecx, edx, ebx, est
        push   ebp          ! stack already points into proc table
        push   esi
        push   edi
    o16 push   ds
    o16 push   es
    o16 push   fs
    o16 push   gs
        mov    dx, ss
        mov    ds, dx
        mov    es, dx

        MP_LOCK(_k_main_lock);
        mov    (save_ebx),   ebx
        THIS_CPU(ebx)

        mov    esi,   _k_reenter
        incb   (ebx)(esi)
```

```
        mov    esi, esp      ! assumes P_STACKBASE == 0
        SET_CPU_STK(ebx)

        mov    ebx,   (save_ebx)

        xor    ebp, ebp      ! for stacktrace
                             ! end of inline save
        sti                  ! allow SWITCHER to be interrupted
                             ! now set up parameters for sys_call()
        push   ebx           ! pointer to user message
        push   eax           ! src/dest
        push   ecx           ! SEND/RECEIVE/BOTH
        call   _sys_call     ! sys_call(function, src_dest, m_ptr)
                             ! caller is now explicitly in proc_ptr
        mov    AXREG(esi), eax      ! sys_call MUST PRESERVE si
        cli                  ! disable interrupts
```

```
#else

_s_call:
_p_s_call:
        cld                  ! set direction flag to a known value
        sub    esp, 6*4      ! skip RETADR, eax, ecx, edx, ebx, est
        push   ebp           ! stack already points into proc table
        push   esi
        push   edi
    o16         push   ds
    o16         push   es
    o16         push   fs
    o16         push   gs
        mov    dx, ss
        mov    ds, dx
        mov    es, dx
        incb   (_k_reenter)
        mov    esi, esp      ! assumes P_STACKBASE == 0
        mov    esp, k_stktop
        xor    ebp, ebp      ! for stacktrace
                             ! end of inline save
        sti                  ! allow SWITCHER to be interrupted
                             ! now set up parameters for sys_call()
        push   ebx           ! pointer to user message
        push   eax           ! src/dest
        push   ecx           ! SEND/RECEIVE/BOTH
        call   _sys_call     ! sys_call(function, src_dest, m_ptr)
                             ! caller is now explicitly in proc_ptr
        mov    AXREG(esi), eax      ! sys_call MUST PRESERVE si
        cli                  ! disable interrupts

#endif
! Fall into code to restart proc/task running.
```

5.10.3. Rutina `restart/restart1`

`restart` y `restart1` son dos partes de la misma rutina, estando la segunda incluida dentro de la primara: `restart1` decrementa el contador de reentradas, restaura los registros y retorna de la última interrupción o trap. `restart`, además, comprueba la cola de interrupciones retenidas (`held`), restaura la pila de las tareas y establece los registros de memoria (LDT y TSS) de la tarea apuntada por `proc_ptr` para que restart1 los restaure. El núcleo puede realizar su salida por `restart1` (en el caso de una reentrada) o por `restart` (en el caso de una primera entrada).

En `restart`, el primer problema a resolver es el acceso a `proc_ptr`, ya que ahora es un vector que hay que indexar en función del número de procesador. En el código original el acceso a esta variable se hace mediante la dirección de `_proc_ptr`. A esta dirección hay ahora que sumarle 4 bytes (32 bits) por cada número de procesador para indexar el vector. Para ello, almacenamos en `ebx` el número de procesador, lo multiplicamos por 4 haciendo un doble desplazamiento a la izquierda, y le sumamos la dirección de `_proc_ptr`. Accediendo a la dirección almacenada en `ebx` tenemos el índice adecuado de `proc_ptr`.

A continuación tenemos que acceder al vector de estructuras TSS (`tss`). La estrategia será la misma: cargar el número de procesador, multiplicar por el tamaño de la estructura y sumar la dirección base de la estructura. Para obtener el número de procesador, en lugar de invocar de nuevo la macro `THIS_CPU`, que tiene un coste computacional relativamente alto, recuperamos de `ecx` el valor obtenido en la operación anterior, ya que guardamos antes una copia en previsión de esta necesidad. Para multiplicar por el tamaño de la escritura TSS, multiplicamos 104 (su tamaño en bytes) por el valor de `ecx` (`cx`). No queda más remedio que recurrir a la instrucción `mul`, ya que 104 no es potencia de 2 y no podemos hacer un desplazamiento como en el caso de `proc_ptr`. El resultado del producto queda en `edx:eax` (obviamente no se sobrepasa el límite de `eax`), registro al que sumamos la dirección del campo `sp0` en el primer índice de `tss`.

A continuación el código continúa por `restart1`. La primera operación es decrementar el contador de reentradas correspondiente al procesador, para lo cual es necesario conocer el número de procesador mediante `THIS_CPU`. Cuando la salida se produce por `restart1` no queda más remedio que invocar la macro `THIS_CPU`, pero si procedemos de `restart`, tenemos todavía el número de procesador almacenado en `ecx`. Así, si entramos directamente a `restart1`, invocamos `THIS_CPU(ecx)`, pero si procedemos de `restart`, omitimos la invocación a `THIS_CPU`, y nos ahorramos el costoso cálculo. El decremento del contador de reentradas es semejante al incremento hecho en `save` y `s_call`, no merece mayor comentario.

Antes de restaurar los registros y retornar de la interrupción o trap, queda por desbloquear el cerrojo principal del núcleo, operación que sólo debe hacerse en el caso de que no sea una reentrada. Como acabamos de decrementar el contador de reentradas, tenemos su valor a mano, por lo que lo comparamos con el valor que debe tener fuera de toda entrada al núcleo (-1, que en el formato de la variable –*unsigned* de 8 bit– es 255).

En función de esta comparación, ejecutamos o saltamos la invocación a la macro `MP_UNLOCK(_k_main_lock)`, que libera el cerrojo principal del núcleo. Obsérvese que hay un breve instante de tiempo en el que el contador de reentradas indica que no hay ninguna entrada, pero el núcleo se encuentra bloqueado. Si en este momento se encontraran las interrupciones habilitadas (hecho que puede ocurrir sólo cuando se invoca `_unhold`, es decir cuando se accede por `restart`) se podría producir una reentrada al núcleo que no se contabilizaría como reentrada sino como entrada, por lo que trataría de cerrar el cerrojo principal y quedaría interbloqueado consigo mismo. De ahí que se inhabiliten las interrupciones en `restart` antes de tocar el contador. Las entradas de otros procesadores no afecta, ya que acceden a sus propios contadores de reentradas.

Otra situación posible sería haber invertido el orden del desbloqueo y el decremento, incluyendo la una invocación incondicional a `MP_UNLOK` al final de `restart` y antes de `restart1`. Esto elimina el problema del interbloqueo, pero podría ocurrir otro problema grave: durante un breve instante de tiempo el cerrojo está abierto y el contador de reentradas indica que ya hay una entrada. Una interrupción al mismo procesador lo haría reentrar, contabilizando como reentrada, y por tanto no bloquearía el cerrojo, que está abierto. En esta situación, otro procesador podría entrar y reentrar en el núcleo mientras el primero permanece también dentro (¡incluso podría volver a reentrar!).

mpx386.s

```
_restart:
```

```
! Flush any held-up interrupts.
! This reenables interrupts, so the current interrupt handler may reenter.
! This does not matter, because the current handler is about to exit and no
! other handlers can reenter since flushing is only done when k_reenter == 0.

#if (ENABLE_MP == 1)
       cmp    (_held_head), 0      ! do fast test to usually avoid function call
       jz     over_call_unhold
       call   _unhold              ! this is rare so overhead acceptable
over_call_unhold:
       THIS_CPU(ebx)
       mov    ecx,   ebx           ! save a copy of this_cpu
       shl    ebx,   2             ! ebx=ebx * sizeof(_proc_ptr)
       add    ebx,   _proc_ptr     ! ebx=&(_proc_ptr[this_cpu])
       mov    esp,   (ebx)         ! will assume P_STACKBASE == 0
       lldt   P_LDT_SEL(esp)               ! enable segment descriptors for task

       mov    eax,   104           ! sizeof struct tss_s for pentium
       mulb   cl                   ! eax = this_cpu * sizeof tss_s
       add    eax,   _tss+TSS3_S_SP0 ! eax = &(tss[this_cpu].sp0)

       lea    ebx,   P_STACKTOP(esp)       ! arrange for next interrupt
       mov    (eax), ebx           ! to save state in process table

       mov    ebx,   ecx           ! faster than THIS_CPU(ebx)
       cli
       jmp    restart1b
restart1:
       THIS_CPU(ebx)
restart1b:
       mov    esi,   _k_reenter
       decb   (ebx)(esi)

       cmpb   (ebx)(esi), 255              ! 255 = -1
       jne    dont_unlock
       MP_UNLOCK(_k_main_lock)
       sti
dont_unlock:
    o16       pop    gs
    o16       pop    fs
    o16       pop    es
    o16       pop    ds
       popad
       add    esp, 4        ! skip return adr
       iretd                ! continue process

#else

       cmp    (_held_head), 0      ! do fast test to usually avoid function call
       jz     over_call_unhold
       call   _unhold              ! this is rare so overhead acceptable
over_call_unhold:
       mov    esp, (_proc_ptr)     ! will assume P_STACKBASE == 0
       lldt   P_LDT_SEL(esp)               ! enable segment descriptors for task
       lea    eax, P_STACKTOP(esp) ! arrange for next interrupt
       mov    (_tss+TSS3_S_SP0), eax       ! to save state in process table
restart1:
       decb   (_k_reenter)
    o16       pop    gs
    o16       pop    fs
    o16       pop    es
    o16       pop    ds
       popad
       add    esp, 4        ! skip return adr
       iretd                ! continue process
#endif
```

5.11. Planificador multiprocesador

Veamos, función por función, los cambios que han afectado al planificador, todas ellas en `proc.c`. Además de estos cambios en los métodos del planificador, se ha añadido un nuevo campo `p_currentcpu` a la estructura del descriptor de proceso (estructura de tipo `struct proc`, definida en `proc.h`), que identifica el procesador que está ejecutando el proceso en un momento dado. Se ha reservado y definido un valor especial (`NONE_CPU`) para el caso de que un proceso no esté en ejecución en ningún procesador.

proc.h

```
struct proc {
  struct stackframe_s p_reg;       /* process' registers saved in stack frame */

  ...
  ...

  unsigned p_pendcount;            /* count of pending and unfinished signals */
  char p_currentcpu;               /* CPU that is running this process now */
  char p_name[16];                 /* name of the process */
};

#define NONE_CPU     -1           /* no cpu assigned to a process */
```

Cada vez que se crea un nuevo proceso es necesario iniciar este valor a `NONE_CPU`. Esta operación se realiza en la función `do_fork()` localizada en el fichero `system.c` (tarea del sistema).

system.c

```
PRIVATE int do_fork(m_ptr)
register message *m_ptr;    /* pointer to request message */
{
/* Handle sys_fork().  m_ptr->PROC1 has forked.  The child is m_ptr->PROC2. */

  ...
  ...

  rpc->p_currentcpu = NONE_CPU;

  ...
  ...

  return(OK);
}
```

5.11.1. Sincronización de las funciones del planificador

Ya se ha tratado en profundidad este punto en apartado correspondiente en los Principios de Diseño de Minix SMP. La bandera `switching` se ha cambiado por unas primitivas `LOCK/UNLOCK/IS_LOCKED` que se invocan mediante las funciones `mp_switching_lock()`, `mp_switching_unlock()` y `mp_switching_islocked()`. Estas funciones se utilizan para proteger el uso del planificador desde las tareas en las funciones `lock_mini_send()`, `lock_pick_proc()`, `lock_ready()`, `lock_unready()` y `lock_sched()`.

proc.c

```
PUBLIC int lock_mini_send(caller_ptr, dest, m_ptr)
struct proc *caller_ptr;        /* who is trying to send a message? */
```

```
int dest;                       /* to whom is message being sent? */
message *m_ptr;                 /* pointer to message buffer */
{
/* Safe gateway to mini_send() for tasks. */
  int result;
  mp_switching_lock();
  result = mini_send(caller_ptr, dest, m_ptr);
  mp_switching_unlock();
  return(result);
}

PUBLIC void lock_pick_proc()
{
/* Safe gateway to pick_proc() for tasks. */
  mp_switching_lock();
  pick_proc();
  mp_switching_unlock();
}

PUBLIC void lock_ready(rp)
struct proc *rp;                /* this process is now runnable */
{
/* Safe gateway to ready() for tasks. */
  mp_switching_lock();
  ready(rp);
  mp_switching_unlock();
}

PUBLIC void lock_unready(rp)
struct proc *rp;                /* this process is no longer runnable */
{
/* Safe gateway to unready() for tasks. */
  mp_switching_lock();
  unready(rp);
  mp_switching_unlock();
}

PUBLIC void lock_sched(cpu)
int cpu;
{
/* Safe gateway to sched() for tasks. */
  mp_switching_lock();
  sched(cpu);
  mp_switching_unlock();
}
```

Además, para proteger el planificador del acceso multiprocesador desde el núcleo, es necesario que los métodos normales (`mini_send()`, `mini_rec()`, `pick_proc()`, `ready()`, `unready()` y `sched()`) se invoquen bajo la protección del mismo cerrojo (`switching`). Esto implica modificar las funciones `sys_call()` e `interrupt()` para que invoquen las primitivas de sincronización.

5.11.2. Función `interrupt()`

En el caso de `interrpt()`, el bloqueo del planificador no se hace hasta después de habernos asegurado de que es necesario: si se detecta una reentrada (el contador de reentradas de este procesador no es 0 y por tanto no es una primera entrada; el del resto de procesadores no importa porque el cerrojo del núcleo no les ha permitido entrar) o que el planificador está ocupado (`mp_switching_islocked()`, es decir, alguien ha bloqueado el cerrojo del planificador), se anota la interrupción como retenida y se termina..

Si no hay que encolar la interrupción como retenida, es que hay que procesar la interrupción, lo que consiste en enviar un mensaje a una tarea. Lo primero será bloquear el planificador pare enviar el mensaje, ya que es un mensaje especial (procede de la tarea especial `HARDWARE`) y no se envía utilizando `mini_send()`. Si

la tarea de destino no está esperando el mensaje, se le anota como pendiente y se termina, no sin antes liberar el planificador.

Si, por el contrario, sí está esperando el mensaje, se le anotan los datos del mensaje, se activa la tarea receptora, se desbloquea el planificador y se termina. En el código original la activación de la tarea se hace sin invocar `ready()` para mayor velocidad. Se ha preferido sustituir el código de activación por una llamada a `ready()` para no tener que repetir aquí los cambios que para la construcción del núcleo multiprocesador afecten a `ready()`.

Obsérvese también la introducción de un vector `interrupt_count` (definido en `mp.c` como un vector de enteros) que cuenta el número de accesos a esta función desde cada procesador, exclusivamente para uso informativo.

proc.c

```
PUBLIC void interrupt(task)
int task;                   /* number of task to be started */
{
/* An interrupt has occurred.  Schedule the task that handles it. */

  register struct proc *rp; /* pointer to task's proc entry */
  int cpu;

  rp = proc_addr(task);

  /* If this call would compete with other process-switching functions, put
   * it on the 'held' queue to be flushed at the next non-competing restart().
   * The competing conditions are:
   * (1) k_reenter == (typeof k_reenter) -1:
   *     Call from the task level, typically from an output interrupt
   *     routine.  An interrupt handler might reenter interrupt().  Rare,
   *     so not worth special treatment.
   * (2) k_reenter > 0:
   *     Call from a nested interrupt handler.  A previous interrupt handler
   *     might be inside interrupt() or sys_call().
   * (3) switching != 0:
   *     Some process-switching function other than interrupt() is being
   *     called from the task level, typically sched() from CLOCK.  An
   *     interrupt handler might call interrupt and pass the k_reenter test.
   */
  interrupt_count[cpu=this_cpu]++;

  if ((k_reenter[cpu] != 0) || (mp_switching_islocked())) {
       lock();
       if (!rp->p_int_held) {
              rp->p_int_held = TRUE;
              if (held_head != NIL_PROC)
                     held_tail->p_nextheld = rp;
              else
                     held_head = rp;
              held_tail = rp;
              rp->p_nextheld = NIL_PROC;
       }
       unlock();
       return;
  }

  mp_switching_lock();
  /* If task is not waiting for an interrupt, record the blockage. */
  if ( (rp->p_flags & (RECEIVING | SENDING)) != RECEIVING ||
      !isrxhardware(rp->p_getfrom)) {
       rp->p_int_blocked = TRUE;
       mp_switching_unlock();
       return;
```

```
  }

  /* Destination is waiting for an interrupt.
   * Send it a message with source HARDWARE and type HARD_INT.
   * No more information can be reliably provided since interrupt messages
   * are not queued.
   */
  rp->p_messbuf->m_source = HARDWARE;
  rp->p_messbuf->m_type = HARD_INT;
  rp->p_flags &= ~RECEIVING;
  rp->p_int_blocked = FALSE;

   /* Make rp ready and run it unless a task is already running.  This is
    * ready(rp) in-line for speed.
    */
   ready(rp);
   mp_switching_unlock();

/*  proc_ptr[cpu=this_cpu]->p_currentcpu=NONE_CPU;
  if (rdy_head[TASK_Q] != NIL_PROC)
       rdy_tail[TASK_Q]->p_nextready = rp;
  else
       proc_ptr[cpu] = rdy_head[TASK_Q] = rp;
  rdy_tail[TASK_Q] = rp;
  rp->p_nextready = NIL_PROC;
  proc_ptr[cpu]->p_currentcpu=cpu;*/
}
```

5.11.3. Función sys_call()

En el caso de `sys_call`, los cambios se reducen a indexar por el número de procesador el acceso a `proc_ptr` para obtener el remitente de la llamada al sistema y a proteger la función completa con el cerrojo del planificador. Para evitar oscurecer el código, que tiene varios puntos de salida con `return`, se ha optado por encapsular la función completa dentro de otra que la invoca con el cerrojo cerrado. Además, la función original se renombra a `sys_call_SMP()` y la nueva se llama como la original (`sys_call()`) para evitar cambios en el resto del código del núcleo. La nueva `sys_call()` sólo cierra el cerrojo, invoca a `sys_call_SMP()`, recoge su salida, libera el cerrojo y retorna la salida de `sys_call_SMP()`.

Obsérvese también la introducción de un vector `scall_count` (definido en `mp.c` como un vector de enteros) que cuenta el número de accesos a este función desde cada procesador, exclusivamente para uso informativo.

proc.c

```
PUBLIC int sys_call(function, src_dest, m_ptr)
int function;               /* SEND, RECEIVE, or BOTH */
int src_dest;               /* source to receive from or dest to send to */
message *m_ptr;                   /* pointer to message */
{
/* The only system calls that exist in MINIX are sending and receiving
 * messages.  These are done by trapping to the kernel with an INT instruction.
 * The trap is caught and sys_call() is called to send or receive a message
 * (or both). The caller is always given by proc_ptr.
 */

  int res;
  mp_switching_lock();
  res=sys_call_SMP(function,src_dest,m_ptr);
  mp_switching_unlock();
  return res;
}
```

```
PRIVATE int sys_call_SMP(function, src_dest, m_ptr)
int function;               /* SEND, RECEIVE, or BOTH */
int src_dest;               /* source to receive from or dest to send to */
message *m_ptr;                   /* pointer to message */
{
/* The only system calls that exist in MINIX are sending and receiving
 * messages.  These are done by trapping to the kernel with an INT instruction.
 * The trap is caught and sys_call() is called to send or receive a message
 * (or both). The caller is always given by proc_ptr.
 */

  register struct proc *rp;
  int n, cpu;

  scall_count[cpu=this_cpu]++;

  /* Check for bad system call parameters. */
  if (!isoksrc_dest(src_dest)) return(E_BAD_SRC);
  rp = proc_ptr[cpu];

  if (isuserp(rp) && function != BOTH) return(E_NO_PERM);

  /* The parameters are ok. Do the call. */
  if (function & SEND) {
       /* Function = SEND or BOTH. */
       n = mini_send(rp, src_dest, m_ptr);
       if (function == SEND || n != OK)
              return(n);    /* done, or SEND failed */
  }

  /* Function = RECEIVE or BOTH.
   * We have checked user calls are BOTH, and trust 'function' otherwise.
   */
  return (mini_rec(rp, src_dest, m_ptr));
}
```

5.11.4. Función `pick_proc()`

Esta función selecciona en el puntero `proc_ptr` un proceso de las colas de procesos preparados. También actualiza si es necesario el valor de `bill_ptr` al nuevo proceso tarificado. Se ha optado por mantener el código original para el núcleo monoprocesador mediante compilación condicional, ya que los cambios que requiere, si bien no son complejos, sí que introducen una carga computacional notable en un punto estratégico del núcleo.

El primer cambio que se observa es el acceso al puntero de proceso `proc_ptr` mediante un índice a la posición correspondiente al procesador que invoca la función, almacenado en la variable `cpu` al principio del código. Lo mismo ocurre con los accesos a `bill_ptr`.

Recordamos que la nueva estrategia de selección de procesos es similar a la original, con la excepción de que para poder elegir un proceso, además de estar preparado, no debe estar siendo ejecutado por ningún otro procesador, lo cual se indica en el campo `p_currentcpu` del descriptor. Así, antes de seleccionar un nuevo proceso hay que marcar el actual como disponible para otros procesadores, y a la hora de recorrer las distintas colas de procesos hay que encontrar un proceso que no esté en ejecución por otro procesador (`p_currentcpu` sea `NONE_CPU`). Además, cuando se encuentre un candidato adecuado, hay que marcarlo como asignado para que ningún otro procesador lo intente ejecutar.

Por último, cuando no hay ningún proceso preparado, cada procesador ejecuta su propia tarea IDLE. Obsérvese que también se marca la tarea IDLE como en ejecución por el procesador correspondiente. Aunque en este caso no es necesario, se hace por regularidad.

Se han señalado en el siguiente fragmento de código los cambios más importantes (no se han marcado por ejemplo, los accesos `proc_ptr` y `bill_ptr` como vector). Obsérvese la introducción de un vector `pickproc_count` (definido en `mp.c` como un vector de enteros) que cuenta el número de accesos a este función desde cada procesador, exclusivamente para uso informativo.

Es importante remarcar que un procesador selecciona su propio proceso, y debe invocar `pick_proc` por sí mismo, no siendo posible que un procesador seleccione procesos para otro.

proc.c

```
PRIVATE void pick_proc()
{
/* Decide who to run now.  A new process is selected by setting 'proc_ptr'.
 * When a fresh user (or idle) process is selected, record it in 'bill_ptr',
 * so the clock task can tell who to bill for system time.
 */

  register struct proc *rp; /* process to run */

#if (ENABLE_MP == 1)
  int cpu=this_cpu;

  proc_ptr[cpu]->p_currentcpu=NONE_CPU;
  proc_ptr[cpu]=NIL_PROC;

  pickproc_count[cpu]++;

  if ( (rp = rdy_head[TASK_Q]) != NIL_PROC) {
       while ((rp) && (rp->p_currentcpu!=NONE_CPU)) rp=rp->p_nextready;
       if (rp) {
              proc_ptr[cpu] = rp;
              rp->p_currentcpu=cpu;
              return;
       }
  }

  if ( (rp = rdy_head[SERVER_Q]) != NIL_PROC) {
       while ((rp) && (rp->p_currentcpu!=NONE_CPU)) rp=rp->p_nextready;
       if (rp) {
              proc_ptr[cpu] = rp;
              rp->p_currentcpu=cpu;
              return;
       }
  }

  if ( (rp = rdy_head[USER_Q]) != NIL_PROC) {
       while ((rp) && (rp->p_currentcpu!=NONE_CPU)) rp=rp->p_nextready;
       if (rp) {
              proc_ptr[cpu] = bill_ptr[cpu] = rp;
              rp->p_currentcpu=cpu;
              return;
       }
  }

  /* No one is ready.  Run the idle task.  The idle task might be made an
   * always-ready user task to avoid this special case.
   */
  bill_ptr[cpu] = proc_ptr[cpu] = proc_addr_IDLE[cpu];
  proc_ptr[cpu]->p_currentcpu=cpu;

/* Check if there is a pending halting */
```

```
   if (cpu_available[cpu]==CPU_HALTED) {
      mp_switching_unlock();
      halt_cpu(cpu);
   }

  #else

   if ( (rp = rdy_head[TASK_Q]) != NIL_PROC) {
        proc_ptr[0] = rp;
        return;
   }
   if ( (rp = rdy_head[SERVER_Q]) != NIL_PROC) {
        proc_ptr[0] = rp;
        return;
   }
   if ( (rp = rdy_head[USER_Q]) != NIL_PROC) {
        proc_ptr[0] = rp;
        bill_ptr[0] = rp;
        return;
   }
   /* No one is ready.  Run the idle task.  The idle task might be made an
    * always-ready user task to avoid this special case.
    */

   bill_ptr[0] = proc_ptr[0] = proc_addr_IDLE[0];
 #endif
 }
```

La comprobación sobre el estado de detención del procesador al final de la función se explica en el apartado 5.16.

5.11.5. Función `mini_send()`

Esta función envía un mensaje a una tarea, activándola si estaba esperándolo, o bloqueando al remitente si destinatario no lo esperaba. Esta función no ha requerido ningún cambio, ya que se limita a comprobar la corrección de la estructura del mensaje, intercambiar los datos en memoria e invocar las funciones `ready()` y `unready()`.

5.11.6. Función `mini_rec()`

Esta función, que en principio es simétrica a `mini_sed()` sí que ha requerido un pequeño cambio. Hacia el final de la misma se invoca a la función `inform()`. Esta función, definida en `system.c` también se utiliza desde la tarea `SYSTEM`, de cuyo código forma parte. `inform()` invoca algunas rutinas del planificador, y como tarea de E/S que es, lo debe hacer a través de las funciones `lock_*`, las cuales bloquean el planificador para asegurar el acceso exclusivo. Por tanto, antes de invocar `inform()`, `mini_rec()` debe liberar el planificador.

proc.c

```
PRIVATE int mini_rec(caller_ptr, src, m_ptr)
register struct proc *caller_ptr;  /* process trying to get message */
int src;                    /* which message source is wanted (or ANY) */
message *m_ptr;                     /* pointer to message buffer */
{
  ...
  ...

  /* If MM has just blocked and there are kernel signals pending, now is the
   * time to tell MM about them, since it will be able to accept the message.
   */
```

```
   if (sig_procs > 0 && proc_number(caller_ptr) == MM_PROC_NR && src == ANY) {
        mp_switching_unlock();
        inform();
        mp_switching_lock();
   }
   return(OK);
}
```

5.11.7. Función ready()

Esta función activa un proceso, añadiéndolo a la cola de procesos preparados de prioridad correspondiente. Además, en el código original se modifica el valor del puntero de proceso `proc_ptr`, así que se ha modificado el código para que se invoque a `pick_proc()` en su lugar debido a las fuertes modificaciones que ha sufrido dicha función. Este cambio tiene implicaciones sobre la estructura del código (forma de anidar y agrupar en bloques), ya que la modificación de `proc_ptr` era una asignación que se hacía antes de terminar de modificar la cola de procesos, pero `pick_proc()` no se puede invocar antes de terminar de actualizar la cola.

proc.c

```
PRIVATE void ready(rp)
register struct proc *rp;   /* this process is now runnable */
{
/* Add 'rp' to the end of one of the queues of runnable processes. Three
 * queues are maintained:
 *   TASK_Q   - (highest priority) for runnable tasks
 *   SERVER_Q - (middle priority) for MM and FS only
 *   USER_Q   - (lowest priority) for user processes
 */

if (istaskp(rp)) {
       if (rdy_head[TASK_Q] != NIL_PROC) {
              /* Add to tail of nonempty queue. */
              rdy_tail[TASK_Q]->p_nextready = rp;
              rdy_tail[TASK_Q] = rp;
              rp->p_nextready = NIL_PROC;  /* new entry has no successor */
              return;
        }
       else {
              rdy_head[TASK_Q] = rp;       /* add to empty queue */
              rdy_tail[TASK_Q] = rp;
              rp->p_nextready = NIL_PROC;  /* new entry has no successor */
              pick_proc();
              return;
       }
  }
  if (!isuserp(rp)) {             /* others are similar */
       if (rdy_head[SERVER_Q] != NIL_PROC)
              rdy_tail[SERVER_Q]->p_nextready = rp;
       else
              rdy_head[SERVER_Q] = rp;
       rdy_tail[SERVER_Q] = rp;
       rp->p_nextready = NIL_PROC;
       return;
  }
#if (SHADOWING == 1)
  if (isshadowp(rp)) {             /* others are similar */
       if (rdy_head[SHADOW_Q] != NIL_PROC)
              rdy_tail[SHADOW_Q]->p_nextready = rp;
       else
              rdy_head[SHADOW_Q] = rp;
       rdy_tail[SHADOW_Q] = rp;
```

```
        rp->p_nextready = NIL_PROC;
        return;
  }
#endif
  for (xp=rdy_head[USER_Q]; xp!=NIL_PROC; xp=xp->p_nextready)
        if (xp==rp) return;
  if (rdy_head[USER_Q] == NIL_PROC)
        rdy_tail[USER_Q] = rp;
  rp->p_nextready = rdy_head[USER_Q];
  rdy_head[USER_Q] = rp;
/*
  if (rdy_head[USER_Q] != NIL_PROC)
        rdy_tail[USER_Q]->p_nextready = rp;
  else
        rdy_head[USER_Q] = rp;
  rdy_tail[USER_Q] = rp;
  rp->p_nextready = NIL_PROC;
*/
}
```

5.11.8. Función unready( )

Esta función elimina un proceso de la cola de preparados que le corresponde, y, si es necesario, invoca al planificador para que tome un nuevo proceso. Los cambios en esta función son mínimos, y se reducen a indexar con el número de procesador actual el acceso a `proc_ptr`, e invocar el planificador en el caso de que el proceso eliminado sea de usuario y se estuviera ejecutando en el procesador actual.

proc.c

```
PRIVATE void unready(rp)
register struct proc *rp;   /* this process is no longer runnable */
{
/* A process has blocked. */

  register struct proc *xp;
  register struct proc **qtail;  /* TASK_Q, SERVER_Q, or USER_Q rdy_tail */

  int cpu=this_cpu;

  if (istaskp(rp)) {
        /* task stack still ok? */
        if (*rp->p_stguard != STACK_GUARD)
              panic("stack overrun by task", proc_number(rp));

        if ( (xp = rdy_head[TASK_Q]) == NIL_PROC) return;
        if (xp == rp) {
              /* Remove head of queue */
              rdy_head[TASK_Q] = xp->p_nextready;
              if (rp == proc_ptr[cpu]) pick_proc();
              return;
        }
        qtail = &rdy_tail[TASK_Q];
  }
  else if (!isuserp(rp)) {
        if ( (xp = rdy_head[SERVER_Q]) == NIL_PROC) return;
        if (xp == rp) {
              rdy_head[SERVER_Q] = xp->p_nextready;
#if (CHIP == M68000)
              if (rp == proc_ptr[cpu])     /* cpu=0 for non-intel */
#endif
              pick_proc();
              return;
        }
```

```
        qtail = &rdy_tail[SERVER_Q];
  } else
#if (SHADOWING == 1)
  if (isshadowp(rp)) {
        if ( (xp = rdy_head[SHADOW_Q]) == NIL_PROC) return;
        if (xp == rp) {
                rdy_head[SHADOW_Q] = xp->p_nextready;
                if (rp == proc_ptr[cpu])
                        pick_proc();
                return;
        }
        qtail = &rdy_tail[SHADOW_Q];
  } else
#endif
  {
        if ( (xp = rdy_head[USER_Q]) == NIL_PROC) return;
        if (xp == rp) {
                rdy_head[USER_Q] = xp->p_nextready;
#if (CHIP == M68000)
                if (rp == proc_ptr[cpu])     /* cpu=0 for no-intel */
#endif
                pick_proc();
                return;
        }
        qtail = &rdy_tail[USER_Q];
  }

  /* Search body of queue.  A process can be made unready even if it is
   * not running by being sent a signal that kills it.
   */
  while (xp->p_nextready != rp)
        if ( (xp = xp->p_nextready) == NIL_PROC) return;
  xp->p_nextready = xp->p_nextready->p_nextready;
  if (*qtail == rp) *qtail = xp;
  if (rp == proc_ptr[cpu]) pick_proc();
}
```

5.11.9. Función sched( )

Originalmente esta función reordena la cola de procesos de usuario (USER) siguiendo un algoritmo *round-robin*, de manera que el primer proceso de la cola pasa a ser el último. El objetivo es que el proceso que se está ejecutando en un momento dado pase al final de la cola y ceda su lugar al siguiente de la cola. En un entorno multiprocesador, para un procesador concreto, es posible que el proceso de usuario que esté ejecutando no aparezca en primera posición de la cola (el primero sería alguno que se esté ejecutando en otro procesador en ese mismo instante), por lo que es necesario modificar el algoritmo de reordenación.

El nuevo algoritmo consiste en buscar en la cola el proceso asignado actualmente al un procesador que se pasa como parámetro, quitarlo de la cola, y añadirlo por el final. Este algoritmo tiene una carga computacional mucho más alta que el original, por lo que se ha conservado el original para un entorno monoprocesador.

Además, se añade a la función un nuevo parámetro, por lo que cambia su interfaz, y con ella la de la versión protegida: `lock_sched()`. Esto origina una serie de cambios en las declaraciones de estas funciones en `proc.c` (para `sched()`) y `proto.h` (para `lock_sched()`), y en la tarea del reloj (`clock.c`), único punto donde se invoca `lock_sched()` (los cambios en esta tarea serán analizados en el apartado 5.14).

proc.c

```
...
FORWARD _PROTOTYPE( void sched, (int cpu) );
```

```
...
```

proto.h

```
...
_PROTOTYPE( void lock_sched, (int cpu)                                    );
...
```

La última instrucción de `sched()` es una invocación a `pick_proc()` que tiene como objetivo actualizar el puntero de proceso para que tome el nuevo proceso planificado. `sched()` acepta como parámetro el número de procesador cuyo proceso asignado es el que ha agotado el cuantum, y por tanto, el que debe tomar un nuevo proceso, pero `pick_proc()` sólo puede ser invocado por el propio procesador que toma el nuevo proceso. Por esta razón, si el procesador planificado no es el mismo que está ejecutando el código, es necesario un mecanismo alternativo: enviar una interrupción al procesador planificado para que éste invoque por sí mismo a `pick_proc()`. Para ello se le envía un mensaje de tipo `MP_CM_SCHED` cuyo funcionamiento ya estudiamos en el apartado 5.9.

proc.c

```
PRIVATE void sched()
{
/* The current process has run too long.  If another low priority (user)
 * process is runnable, put the current process on the end of the user queue,
 * possibly promoting another user to head of the queue.
 */
#if (ENABLE_MP == 1)
  register struct proc *rp,*xp;
#endif

  if (rdy_head[USER_Q] == NIL_PROC) return;

#if (ENABLE_MP == 1)
/* One or more user processes queued. */
  rp=proc_ptr[cpu];
  if (! isuserp(rp)) return;
  if (rdy_tail[USER_Q]==rp) return;
  rdy_tail[USER_Q]->p_nextready = rp;
  rdy_tail[USER_Q] = rp;
  if (rdy_head[USER_Q]==rp)
     rdy_head[USER_Q] = rp->p_nextready;
  else
     for (xp=rdy_head[USER_Q]; xp->p_nextready!=NIL_PROC; xp=xp->p_nextready)
        if (xp->p_nextready==rp) {
           xp->p_nextready=rp->p_nextready;
           break;
        }
  rp->p_nextready = NIL_PROC;
  if (cpu==this_cpu) pick_proc();
  else interrupt_cpu(cpu,MP_CM_SCHED,MP_CM_ARG_NONE);

#else

  /* One or more user processes queued. */
  rdy_tail[USER_Q]->p_nextready = rdy_head[USER_Q];
  rdy_tail[USER_Q] = rdy_head[USER_Q];
  rdy_head[USER_Q] = rdy_head[USER_Q]->p_nextready;
  rdy_tail[USER_Q]->p_nextready = NIL_PROC;
  pick_proc();

#endif
```

```
}
```

5.12. Habilitación e inhabilitación de procesadores

La habilitación e inhabilitación de procesadores es interesante para el uso didáctico del sistema multiprocesador. Además, la inhabilitación es necesaria como paso previo a la detención del sistema, ya que es necesario que sólo el BSP quede activo en el momento de detener el sistema (en el mismo estado en el que arrancó).

La habilitación e inhabilitación de procesadores tiene dos partes: por un lado, activación o desactivación de las interrupciones en el APIC local del procesador, y por otro, el registro en el sistema de los procesadores activos con lo que se cuenta. Para esta última cuestión se ha creado una variable global que almacena esta información. Se declara en `mp.h` como un vector de enteros, donde cada índice indica el estado del procesador correspondiente, que puede ser una de las siguientes cuatro posibilidades (declaradas en `mp.h`):

- `CPU_ENABLED`: El procesador correspondiente está activo y funcionando.
- `CPU_DISABLED`: El procesador correspondiente está inhabilitado y no recibe interrupciones ni ejecuta código.
- `CPU_DISABLING`: El procesador correspondiente está siendo inhabilitado y puede recibir interrupciones y ejecutar código por un tiempo indeterminado.
- `CPU_HALTED`: El procesador correspondiente definitivamente inhabilitado y no recibe interrupciones ni ejecuta código, sin posibilidad de rehabilitación.

mp.h

```
#define CPU_ENABLED         1               /* cpu is runing */
#define CPU_DISABLED        0               /* cpu is not runing */
#define CPU_DISABLING       2               /* cpu is ordered to stop */
#define CPU_HALTED          3               /* cpu is halted */
extern int cpu_available[MP_NR_CPUS];       /* is this cpu running now? */
```

Para realizar la activación de interrupciones en el APIC local es necesario programar dos registros de la LVT (*Local Vector Table*). Además, activamos el APIC local para asegurar su funcionamiento. La función `enable_apic_ints()`, definida en `mp.c`, se encarga de esta operación. Se trata de programar los registros `LINT0` y `LINT1` de la LVT para que dirijan al procesador las interrupciones procedentes del PIC 8259 que entran por la entrada local 0 como interrupción externa y por la entrada local 1 como NMI (*nonmaskable interrupt*).

mp.c

```
void enable_apic_ints(void) {
/* Enable current APIC and open to interrputs from PIC */
   u32_t reg;

   reg = LOCAL_APIC_READ(LOCAL_APIC_SPIV);
   reg |= (1<<ENABLE_APIC_SHIFT);           /* Enable APIC */
   LOCAL_APIC_WRITE(LOCAL_APIC_SPIV, reg);

   reg = LOCAL_APIC_READ(LOCAL_APIC_LVTL0);
   reg &= ~(7<<LVT_DM_SHIFT);               /* clear delivery mode */
   reg &= ~(1<<LVT_MASKED_SHIFT);           /* unmask LINTIN0 */
   reg |= (LVT_DM_EXTINT<<LVT_DM_SHIFT);    /* ExtINT at LINTINT0 */
   LOCAL_APIC_WRITE(LOCAL_APIC_LVTL0, reg);
```

```
    reg = LOCAL_APIC_READ(LOCAL_APIC_LVTL1);
    reg &= ~(7<<LVT_DM_SHIFT);              /* clear delivery mode */
    reg &= ~(1<<LVT_MASKED_SHIFT);          /* ummask LINTIN1 */
    reg |= (LVT_DM_NMI<<LVT_DM_SHIFT);      /* NMI at LINTINT1 */
    LOCAL_APIC_WRITE(LOCAL_APIC_LVTL1, reg);
}
```

De forma similar programamos estos registros para inhabilitar la llegada de interrupciones al procesador, mediante la función `disable_apic_ints()`, también en `mp.c`. Se enmascaran las interrupciones que llegan a las líneas LINT0 y LINT1, pero no se inhabilita el APIC local para permitir que éste siga recibiendo las interrupciones interprocesador (IPI) que se pueden utilizar más adelante para reactivar el procesador.

mp.c

```
void disable_apic_ints(void) {
/* Disable current APIC and close interrupts from PIC */
   u32_t reg;

   reg = LOCAL_APIC_READ(LOCAL_APIC_SPIV);
   reg &= ~(1<<ENABLE_APIC_SHIFT);          /* Disable apic */
   /*LOCAL_APIC_WRITE(LOCAL_APIC_SPIV, reg);*/
   /* Let APIC enabled in order to listen IPI and make posible
      re-enabling */

   reg = LOCAL_APIC_READ(LOCAL_APIC_LVTL0);
   reg |= (1<<LVT_MASKED_SHIFT);            /* mask LINTIN0 */
   LOCAL_APIC_WRITE(LOCAL_APIC_LVTL0, reg);

   reg = LOCAL_APIC_READ(LOCAL_APIC_LVTL1);
   reg &= ~(1<<LVT_MASKED_SHIFT);           /* ummask LINTIN1 */
   LOCAL_APIC_WRITE(LOCAL_APIC_LVTL1, reg);
}
```

La complicación de la activación y desactivación de procesadores radica en que es el propio procesador el que debe ejecutar el acceso a su APIC local para habilitar o inhabilitar las interrupciones. Así, un procesador no puede activar o desactivar directamente a otro, si no que debe ordenarle que lo haga.

Para la activación y desactivación de procesadores (habilitación/deshabilitación de interrupciones más registro en el vector de procesadores disponibles) contamos con dos funciones paralelas (en `mp.c`): `enable_cpu()` y `disable_cpu()`. Ambas aceptan dos parámetros: el primero indica el número de procesador que queremos activar/desactivar y el segundo si queremos o no que se muestre un mensaje en la consola cuando se complete la operación (`WITH_ECHO` o `WITHOUT_ECHO`, valores definidos en `mp.h`).

El código de ambas funciones es semejante. En primer lugar se comprueba si el procesador está ya activado o desactivado, en cuyo caso se retorna inmediatamente. Luego se verifica si el procesador que está ejecutando el código es el que debe ser activado o desactivado:

- Si es el mismo, se habilitan las interrupciones, se marca el procesador como disponible o no disponible y se imprime el mensaje correspondiente si se eligió la opción de eco.
- Si no es el mismo, se envía una interrupción interprocesador al procesador que hay que activar o desactivar para que éste lo haga, y en el caso de la desactivación, se marca el procesador como en proceso de desactivación.

Además, en la desactivación se toma como nuevo proceso la tarea IDLE (sólo si el proceso que ejecutaba el procesador no era una tarea de E/S, que debería continuarse inmediatamente ya que podría haber

bloqueado el planificador) y se marca el anterior proceso en ejecución como no ejecutado por ningún procesador (se trata de la realizar las mismas operaciones que haría `lock_pick_proc()`, pero asegurándonos que se toma IDLE y sin bloquear el planificador), y en reactivación se invoca a `lock_pick_proc()` para que comience a ejecutar algún proceso (no es posible el interbloqueo por el cerrojo del planificador puesto que el procesador en cuestión se debía encontrar previamente ejecutando la tarea IDLE).

La detención definitiva de un procesador es algo más compleja y la estudiaremos en el apartado 5.16.

mp.c

```
PUBLIC void enable_cpu(int cpu, int echo) {
/* Mark a CPU as usable, enables it, and open interrupts.
   If echo is nonzero, print a message of successfully enabling */

   if (cpu_available[cpu]==CPU_ENABLED) return;

   /* A CPU only can only enable its own APIC */
   if (cpu==this_cpu) {
      enable_apic_ints();
      cpu_available[cpu]=CPU_ENABLED;
      lock_pick_proc();
      if (echo) printk("CPU%d enabled\n",cpu);
   }
   else {
      interrupt_cpu(cpu,MP_CM_ENABLE,echo);
   }
}


PUBLIC void disable_cpu(int cpu, int echo) {
/* Mark a CPU as non usable, diables it, and close interrupts.
   If echo is nonzero, print a message of successfully disabling */

   if (cpu_available[cpu]==CPU_DISABLED) return;

   /* A CPU only can disable itself. */
   if (cpu==this_cpu) {
      disable_apic_ints();
      cpu_available[cpu]=CPU_DISABLED;
      if (!istaskp(proc_ptr[cpu])) {
         proc_ptr[cpu]->p_currentcpu=NONE_CPU;
         bill_ptr[cpu]=proc_ptr[cpu]=proc_addr_IDLE[cpu];
      }
      if (echo) printk("CPU%d disabled\n",cpu);
   }
   else {
      cpu_available[cpu]=CPU_DISABLING;
      interrupt_cpu(cpu,MP_CM_DISABLE,echo);
   }
}
```

5.13. Habilitación e inhabilitación de caches

Como vimos en el capítulo 2 la coherencia de cache está asegurada por hardware y es transparente al software, excepto por el detalle de que debemos habilitar e inhabilitar la cache de cada procesador. Las funciones `enable_cache()` y `disable_cache()`, definidas en `xmp.s`, realizan estas operaciones. Los cambios en la configuración de la cache requieren la invalidación del contenido de la cache del procesador, lo que se efectúa con la instrucción de ensamblador `wbinv` (*write back and invalidate cache*). Esta instrucción no está incluida en el repertorio del ensamblador de Minix, por lo que debe declararse según su valor de código máquina.

xmp.s

```
#define wbinv .data1 0x0F, 0x09

! Enable cache on current cpu
_enable_cache:
              mov    ax,    cr0
              and    ax,    0x9FFFFFFF      ! 100111111111...111
              mov    cr0,   ax
              wbinv
              ret

! Disable cache on current cpu
_disable_cache:
              wbinv
              mov    ax,    cr0
              or     ax,    0x60000000      ! 01100000000....000
              mov    cr0,   ax
              ret
```

5.14. La tarea del reloj

La tarea del reloj requiere algunas modificaciones debido a la fuerte interacción que tiene con el planificador, ya que es la tarea encargada de forzar la reordenación de la cola de procesos de usuario cuando el proceso actual agota su cuantum. Además, es la encargada de manipular la tarificación de tiempo de procesador que consumen los procesos.

De las múltiples funciones que forman la tarea del reloj nos interesan ahora sólo dos: `clock_handler()` y `do_clocktick()`.

- `clock_handler()` es un manejador de interrupción que se ejecuta en cada tick de reloj, por lo que su tarea es bastante simple con el objetivo de consumir poco tiempo. Entre sus operaciones está mantener el reloj del sistema y controlar diversos dispositivos cuyo funcionamiento depende de algún tipo de temporización (p.e. terminales, impresoras, etc.). Además, incrementa el tiempo consumido en las tareas en ejecución y verifica si hay alguna operación más compleja, como alguna alarma vencida o la finalización de un cuantum, en cuyo caso invoca `interrupt(CLOCK)` para realizar operaciones más complejas.
- `do_clocktick()`es la rutina de la tarea del reloj que procesa los mensajes procedentes del hardware, es decir, la que se ejecuta cuando se invoca `interrupt(CLOCK)`.

El primer cambio necesario lo encontramos en `clock_handler()`, ya que, como acabamos de indicar, es donde se incrementa el tiempo consumido en las tareas en ejecución. Puesto que tenemos tantas tareas en ejecución como procesadores, habrá que realizar este incremento para cada una de estas tareas.

En Minix se contabilizan dos tipos de tiempo consumible por una tarea: tiempo de usuario (consumido en ejecución de instrucciones) y tiempo del sistema (consumido en ejecución de llamadas al sistema). El puntero `bill_ptr` indica qué proceso de usuario fue el último en pasar por el procesador, y `proc_ptr` indica cual es el proceso (de cualquier prioridad) actual. Cuando se produce una interrupción de reloj, se produce una entrada al núcleo si se estaba ejecutando alguna tarea, o una reentrada si se estaba ejecutando el núcleo. Así, el contador de reentradas nos permite decidir si el tiempo de usuario debe cargarse al proceso de usuario o a la tarea `HARDWARE`. La comprobación del contador de reentradas difiere en el procesador actual (que al ser el que está ejecutando el manejador de interrupción del reloj tiene una entrada más) y en el resto, que si bien no pueden estar dentro del núcleo debido al cerrojo, sí pueden estar intentando entrar y mantener el contador de reentradas ya incrementado.

Las operaciones de tarificación deben repetirse para cada procesador del sistema, lo que convierte las instrucciones correspondientes en el núcleo original en un bucle para el núcleo multiprocesador. Esto obliga reorganizar el código ligeramente.

La comprobación de la finalización del cuantum implica los valores contenidos en `bill_ptr` y `prev_ptr`. `prev_ptr` mantiene el valor registrado en `bill_ptr` durante una interrupción de reloj anterior. Por lo tanto es una variable que debe ser replicada de la misma forma que `bill_ptr`. Entre las condiciones que se deben comprobar para verificar finalización del cuantum está la comparación entre los valores de `bill_ptr` y `prev_ptr`, por lo que esta operación deberá repetirse una vez para cada procesador hasta que para alguno de ellos se dé el conjunto de condiciones. De la misma forma, la actualización de `prev_ptr` al valor de `bill_ptr` se debe repetir para cada procesador.

clock.c

```
PRIVATE int clock_handler(irq)
int irq;
{

  ...
  ...

  register struct proc *rp;
  register unsigned ticks;
  clock_t now;
  int cpu;

  if (ps_mca) {
       /* Acknowledge the PS/2 clock interrupt. */
       out_byte(PORT_B, in_byte(PORT_B) | CLOCK_ACK_BIT);
  }

  /* Update user and system accounting times.
   * First charge the current process for user time.
   * If the current process is not the billable process (usually because it
   * is a task), charge the billable process for system time as well.
   * Thus the unbillable tasks' user time is the billable users' system time.
   */
  ticks = lost_ticks + 1;
  lost_ticks = 0;
  FOR_EACH_ENABLED_CPU(cpu) {
     if (k_reenter[cpu] != ((cpu==this_cpu)?0:255) )
       rp = proc_addr(HARDWARE);
     else
       rp = proc_ptr[cpu];
     rp->user_time += ticks;
     if (rp != bill_ptr[cpu] && rp != proc_addr_IDLE[cpu])
       bill_ptr[cpu]->sys_time += ticks;
  }

  pending_ticks += ticks;
  now = realtime + pending_ticks;
  if (tty_timeout <= now) tty_wakeup(now);  /* possibly wake up TTY */
#if (CHIP != M68000)
  pr_restart();                                 /* possibly restart printer */
#endif
#if (CHIP == M68000)
  kb_timer();                                /* keyboard repeat */
  if (sched_ticks == 1) fd_timer();             /* floppy deselect */
#endif

  FOR_EACH_ENABLED_CPU(cpu)
  if (next_alarm <= now ||
      sched_ticks == 1 &&
      bill_ptr[cpu] == prev_ptr[cpu] &&
```

```
#if (SHADOWING == 0)
      rdy_head[USER_Q] != NIL_PROC) {
#else
     (rdy_head[USER_Q] != NIL_PROC || rdy_head[SHADOW_Q] != NIL_PROC)) {
#endif
       interrupt(CLOCK);
       return 1;     /* Reenable interrupts */
  }

  if (--sched_ticks == 0) {
       /* If bill_ptr == prev_ptr, no ready users so don't need sched(). */
       sched_ticks = SCHED_RATE;   /* reset quantum */
       FOR_EACH_CPU(cpu)
          prev_ptr[cpu] = bill_ptr[cpu];    /* new previous process */
  }
  return 1;   /* Reenable clock interrupt */
}
```

La función `do_clock_tick()` invoca la rutina de reorganización de la cola de procesos de usuario (`lock_sched()`) si es necesario según las condiciones de cada procesador que esté funcionando.

clock.c

```
PRIVATE void do_clocktick()
{
/* Despite its name, this routine is not called on every clock tick. It
 * is called on those clock ticks when a lot of work needs to be done.
 */

  register struct proc *rp;
  register int proc_nr;
  int cpu;

  if (next_alarm <= realtime) {
       /* An alarm may have gone off, but proc may have exited, so check. */
       next_alarm = LONG_MAX;      /* start computing next alarm */
       for (rp = BEG_PROC_ADDR; rp < END_PROC_ADDR; rp++) {
              if (rp->p_alarm != 0) {
                     /* See if this alarm time has been reached. */
                     if (rp->p_alarm <= realtime) {
                            /* A timer has gone off.  If it is a user proc,
                             * send it a signal.  If it is a task, call the
                             * function previously specified by the task.
                             */
                            proc_nr = proc_number(rp);
                            if (watch_dog[proc_nr+NR_TASKS]) {
                                   watchdog_proc= proc_nr;
                                   (*watch_dog[proc_nr+NR_TASKS])();
                            }
                            else
                                   cause_sig(proc_nr, SIGALRM);
                            rp->p_alarm = 0;
                     }

                     /* Work on determining which alarm is next. */
                     if (rp->p_alarm != 0 && rp->p_alarm < next_alarm)
                            next_alarm = rp->p_alarm;
              }
       }
  }

  /* If a user process has been running too long, pick another one. */
  if (--sched_ticks == 0) {
       FOR_EACH_ENABLED_CPU(cpu) {
              if (bill_ptr[cpu] == prev_ptr[cpu])       /* process has run too long */
```

```
                        lock_sched(cpu);
                prev_ptr[cpu] = bill_ptr[cpu];          /* new previous process */
        }
        sched_ticks = SCHED_RATE;           /* reset quantum */
  }
#if (SHADOWING == 1)
  if (rdy_head[SHADOW_Q]) unshadow(rdy_head[SHADOW_Q]);
#endif
}
```

5.15. Cambios en la iniciación del sistema

Recogemos en esta sección los cambios que ha sido necesario introducir en la secuencia de iniciación normal, en la función `main()` del fichero `main.c`. El más importante ya lo conocemos: al final de la función, cuando el sistema está listo para comenzar a ejecutar procesos, invocamos la secuencia de iniciación multiprocesador (`mp_start()`).

Una de las principales funciones de `main()` es iniciar la tabla de procesos, llenándola con las tareas de E/S y procesos iniciales. Se rellenan todas las entradas con sus valores iniciales y luego se introducen las tareas predefinidas. Como parte de los valores iniciales hay que añadir el campo `p_currentcpu`, que hay que establecer a `NONE_CPU`. El resto de la iniciación de procesos no sufre ninguna variación, por lo que se ha omitido ese fragmento del código de la función mostrado a continuación, en el que se resaltan los cambios.

La iniciación del puntero de proceso `proc_ptr`, del puntero de tarificación `bill_ptr` y del puntero precalculado a la tarea IDLE `proc_addr_IDLE` se han unificado en un único bloque, adelantado más al inicio de la función, antes del bucle que introduce las tareas en la tabla de procesos, ya que en su interior se invoca a `ready()`, que a su vez puede invocar a `pick_proc()`, quien espera encontrar valores estables en `proc_ptr` y `proc_addr_IDLE`.

Antes de iniciar el arranque multiprocesador hay que iniciar el vector de procesadores disponibles, estado por defecto todos los procesadores detenidos hasta que los AP por sí mismos cambien este estado. En el caso del BSP, nos aseguramos de que las interrupciones del APIC están habilitadas y lo marcamos como activo.

Por último, y sólo en el caso de que tengamos un núcleo multiprocesador, bloqueamos el cerrojo principal del núcleo e iniciamos la secuencia de arranque de los AP. Durante el arranque, cada AP tratará de adquirir el cerrojo del núcleo, con lo que quedará bloqueado y sincronizado con el BSP hasta que éste lo libere. Esto ocurrirá en el momento en que termine la secuencia de arranque multiprocesador, ya que a continuación invoca `restart()`, que desbloqueará el cerrojo y pondrá el BSP a ejecutar la primera tarea.

main.c

```
PUBLIC void main()
{
/* Start the ball rolling. */

  register struct proc *rp;
  register int t;
  int sizeindex;
  phys_clicks text_base;
  vir_clicks text_clicks;
  vir_clicks data_clicks;
  phys_bytes phys_b;
  reg_t ktsb;               /* kernel task stack base */
  struct memory *memp;
  struct tasktab *ttp;
  int cpu;
```

```
  /* Initialize the interrupt controller. */
  intr_init(1);

  /* Interpret memory sizes. */
  mem_init();


  /* Clear the process table.
   * Set up mappings for proc_addr() and proc_number() macros.
   */
  for (rp = BEG_PROC_ADDR, t = -NR_TASKS; rp < END_PROC_ADDR; ++rp, ++t) {
       rp->p_flags = P_SLOT_FREE;
       rp->p_nr = t;        /* proc number from ptr */
       rp->p_currentcpu=NONE_CPU;
        (pproc_addr + NR_TASKS)[t] = rp;        /* proc ptr from number */
  }
  FOR_EACH_CPU(cpu)
     proc_ptr[cpu] = bill_ptr[cpu] = proc_addr_IDLE[cpu] = proc_addr(IDLE(cpu));
     /* it has to point somewhere */

  /* Set up proc table entries for tasks and servers.  The stacks of the
   * kernel tasks are initialized to an array in data space.  The stacks
   * of the servers have been added to the data segment by the monitor, so
   * the stack pointer is set to the end of the data segment.  All the
   * processes are in low memory on the 8086.  On the 386 only the kernel
   * is in low memory, the rest if loaded in extended memory.
   */

  /* Task stacks. */
  ktsb = (reg_t) t_stack;

  for (t = -NR_TASKS; t <= LOW_USER; ++t) {
       rp = proc_addr(t);                  /* t's process slot */
       ttp = &tasktab[t + NR_TASKS];            /* t's task attributes */
       strcpy(rp->p_name, ttp->name);

       ...
       ...

       if (!isidlehardware(t)) lock_ready(rp);   /* IDLE, HARDWARE nevereadv */
       rp->p_flags = 0;

       alloc_segments(rp);
  }

  proc[NR_TASKS+INIT_PROC_NR].p_pid = 1;/* INIT of course has pid 1 */

  FOR_EACH_CPU(cpu) cpu_available[cpu]=CPU_DISABLED;
  enable_cpu(this_cpu, WITHOUT_ECHO);
  /* For the moment only BSP is available */

  lock_pick_proc();

#if (ENABLE_MP == 1)
  /* Multiprocessor initialization */
  lock_mp_kernel(); /* Make other CPUs to stop before enter the kernel */
  mp_start();
#endif

  /* Now go to the assembly code to start running the current process. */
  restart();
}
```

5.16. Detención del sistema multiprocesador

Antes de detener el sistema operativo es necesario volver a configurar el sistema multiprocesador para que vuelva a funcionar como un monoprocesador. Esto incluye dos operaciones:

- Detener todos los AP.
- Reactivar el BSP si no está activo.

Además, puesto que el BSP es el único procesador que va a quedar activo, deberá ser éste el que ejecute la secuencia de detención.

La detención del sistema se realiza mediante la función `wreboot()`, localizada en el fichero `keyboard.c`. Esta función se invoca mediante la combinación de teclas `CTRL+ALT+SUPR`, desde la tarea del sistema (atendiendo una llamada al sistema) o desde la rutina de error crítico en el núcleo (`panic`).

Puesto que la secuencia de detención debe ser ejecutada por el BSP, incluimos esta comprobación al inicio de la función `wreboot()`:

- Si es el BSP quien la ejecuta, invocamos la secuencia de detención multiprocesador (`mp_stop()`) y continuamos la detención normal.
- Si es un AP quien la ejecuta, enviamos una interrupción al BSP para que éste invoque a `wreboot()`, y terminamos. Obsérvese que no se comprueba si el BSP está activo, ya que aunque no lo esté, sí que recibirá la IPI de comunicación y ejecutará el manejador.

keyboard.c

```
PUBLIC void wreboot(how)
int how;            /* 0 = halt, 1 = reboot, 2 = panic!, ... */
{
/* Wait for keystrokes for printing debugging info and reboot. */

  int quiet, code;
  static u16_t magic = MEMCHECK_MAG;
  struct tasktab *ttp;

#if (ENABLE_MP==1)
  /* Only BSP can reboot */
  if (this_cpu!=0) {
    interrupt_cpu(0,MP_CM_REBOOT,how);
    return;
  }
  mp_stop();
#endif

  ...
  ...
}
```

La secuencia de detención multiprocesador tiene varias etapas, y la encontramos en la función `mp_stop()`, localizada en el fichero `mp.c`.

En esta secuencia, cada AP va a recibir una orden de detención a través de una IPI. Puesto que se trata de una interrupción, cada AP necesitará entrar en el núcleo para ejecutar el manejador de la interrupción (IPI), pero el núcleo puede encontrarse bloqueado por el BSP dependiendo de las circunstancias en que se produzca la invocación de `wreboot()`. Por esta razón, antes de comenzar a enviar las interrupciones, desbloqueamos el cerrojo del núcleo. A continuación se envían todas las órdenes de detención y se espera un tiempo prudencial para que los AP se marquen a sí mismos como detenidos, antes de continuar con la secuencia.

La detención de los AP se logra mediante la función `halt_cpu()`. En esta función (que encontramos en `mp.c`) se envía un mensaje por interrupción entre procesadores de tipo `MP_CM_HALT` al AP que se quiere detener, el cual desactivará completamente sus interrupciones y esperará en una instrucción `HLT`. Para detener realmente los procesadores, la especificación Intel MP recomienda otro mecanismo, que es el que se invoca a continuación. Se trata de enviarles una interrupción (IPI) de tipo `INIT`, que reinicia el procesador y lo hace esperar en un estado de detención.

El último paso es asegurarse de que el BSP está funcionando normalmente, para lo cual comprobamos su estado de activación, y, si es necesario invocamos la rutina de habilitación.

mp.c

```
void mp_stop(void) {
/* Stop all processor excepting BSP */
   int dummy_trampoline_addr=0;
   int cpu,n;

   if (this_cpu!=0) {
      printk("Only BSP can stop MP!!!!\n");
      return;
   }
   printk("Disabling CPUs...\n");

   /* Disabling CPUs need to interrupt them, so unlock the kernel
      and let them CPUs to enter kernel for disabling */
   unlock_mp_kernel();

   FOR_EACH_AP(cpu)
      halt_cpu(cpu);
   FOR_EACH_AP(cpu)
      for (n=0; n<500; n++)
         if (cpu_available[cpu]==CPU_DISABLED) break;
         else milli_delay(1);

   /* Stop all APs */
   /* Sending INIT to a processor makes it to wait in a halt state */
   FOR_EACH_AP(cpu)
      send_init_ipi(dummy_trampoline_addr, cpu);

   /* Ensure BSP has APIC interrupts enabled */
   if (cpu_available[0]!=CPU_ENABLED) enable_cpu(0,WITH_ECHO);
}
```

La función de detención de los AP (`halt_cpu`) tiene una estructura similar a las de habilitación e inhabilitación de procesadores. Un procesador sólo puede detenerse a sí mismo, por lo que si la función se invoca para detener otro procesador, se debe enviar un mensaje para ordenar al procesador que se detenga.

La detención total del procesador significa que nunca se va a terminar de ejecutar esta función, ya que la invocación a `halt()` nunca debe retornar. El procesador que ejecuta esta función se encuentra en el contexto del núcleo, por lo que ha debido bloquear el cerrojo de entrada al mismo, cerrando el paso a los demás procesadores. Por tanto, antes de detenerse, debe liberar el cerrojo para que el sistema siga funcionando.

Puesto que el procesador no va a volver a trabajar, no necesita seguir actualizando su caché. Por esta razón se inhabilita, lo que mejorará el rendimiento del sistema de memoria del computador, al no seguir siendo necesario realizar los procedimientos de coherencia multiprocesador.

mp.c

```
PUBLIC void halt_cpu(int cpu) {
/* Stop a CPU */

   /* A CPU only can stop itself. */
   if (cpu==this_cpu) {
      disable_apic_ints();
      cpu_available[cpu]=CPU_HALTED;
      if (isuserp(proc_ptr[cpu])) {
         proc_ptr[cpu]->p_currentcpu=NONE_CPU;
         bill_ptr[cpu]=proc_ptr[cpu]=proc_addr_IDLE[cpu];
         printk("CPU%d halted\n",cpu);
         unlock_mp_kernel();
         disable_cache();
         halt();     /* Never return */
      }
}
   else {
      cpu_available[cpu]=CPU_DISABLING;
      interrupt_cpu(cpu,MP_CM_HALT,MP_CM_ARG_NONE);
   }
}
```

Pero la detención no se puede hacer en el caso de que el procesador estuviera ejecutando alguna tarea de E/S en el momento de recibir la orden, ya que dicha ejecución no se debe completar ya que podría estar realizaran operaciones importantes, y, además, podría haber bloqueado el cerrojo del planificador, dejando detenidos a otros procesadores. Por lo tanto, la detención sólo se puede completar si el procesador no estaba ejecutando una tarea de E/S. En caso contrario queda pendiente hasta que se complete la ejecución de la tarea E/S, lo que se comprueba dentro del planificador, en `pick_proc`. En caso de que el procesador que invoca `pick_proc` esté marcado como detenido después de tomar la tarea IDLE, se vuelve a invocar `halt_cpu()`, pero previamente se libera el cerrojo del planificador, que ha debido ser adquirido antes de llegar a `pick_proc` (ver sección 5.11.4).

pick_proc() en proc.c

```
...
...
  /* Check if there is a pending halting */
  if (cpu_available[cpu]==CPU_HALTED) {
     mp_switching_unlock();
     halt_cpu(cpu);
  }
...
...
```

La función `halt()` está escrita en ensamblador en `xmp.s`. Se limita a inhabilitar las interrupciones y ejecutar una instrucción `hlt` para detener el procesador.

xmp.s

```
_halt:
       cli
       hlt
       jmp _halt
```

5.17. Manipulación del estado multiprocesador

Se han programado algunos mecanismos que permiten controlar manualmente la habilitación de los procesadores, así como conocer su estado.

5.17.1. Activación y desactivación manual de procesadores

Se han programado 4 teclas que permiten activar y desactivar el BSP y el primer AP:

- F5: Activar BSP.
- F6: Desactivar BSP.
- F7: Activar primer AP.
- F8: Desactivar primer AP.

Estas teclas se programan en keyboard.c, dentro de la función func_key(), que contiene la programación de otras teclas especiales. Obsérvese que al desactivar un procesador cualquiera se comprueba que el otro está activo para asegurar que nunca quedan los dos procesadores desactivados, lo cual causaría el bloqueo del sistema.

keyboard.c

```
PRIVATE int func_key(scode)
int scode;                  /* scan code for a function key */
{
/* This procedure traps function keys for debugging and control purposes. */

  unsigned code;

  code = map_key0(scode);                   /* first ignore modifiers */
  if (code < F1 || code > F12) return(FALSE);    /* not our job */

  switch (map_key(scode)) {                 /* include modifiers */

  case F1:    p_dmp(); break;               /* print process table */
  case F2:    map_dmp(); break;    /* print memory map */
  case F3:    toggle_scroll(); break;       /* hardware vs. software scrolling */
#if (MP_NR_CPUS > 1)
  case F5:    enable_cpu(0,WITH_ECHO);
              break;
  case F6:    disable_cpu(0,WITH_ECHO);
              if (cpu_available[1]!=CPU_ENABLED) enable_cpu(1,WITH_ECHO);
              break;
  case F7:    enable_cpu(1,WITH_ECHO);
              break;
  case F8:    disable_cpu(1,WITH_ECHO);
              if (cpu_available[0]!=CPU_ENABLED) enable_cpu(0,WITH_ECHO);
              break;
#endif

#if ENABLE_NETWORKING
  case F5:    dp_dump(); break;             /* network statistics */
#endif
  case CF7:   sigchar(&tty_table[CONSOLE], SIGQUIT); break;
  case CF8:   sigchar(&tty_table[CONSOLE], SIGINT); break;
  case CF9:   sigchar(&tty_table[CONSOLE], SIGKILL); break;
  default:    return(FALSE);
  }
  return(TRUE);
}
```

5.17.2. Información del estado multiprocesador

Minix provee un mecanismo que muestra el estado del sistema (procesos activos), que se activa mediante la pulsación de las teclas `F1` y `F2`. La pulsación de `F1` hace que se invoque la función `p_dmp()`, que muestra en la consola un listado de los procesos activos. Se ha ampliado esta función para que muestre información adicional sobre el estado multiprocesador:

Para cada proceso listado, a la derecha de la columna `pc` (registro contador de programa) se indica el número de procesador que está ejecutado el proceso (un guión cuando no está en ejecución).

Además, al final del listado se incluye una nueva tabla con una fila por cada procesador y las siguientes columnas:

- Número de procesador
- Estado (0=Deshabilitado, 1=Activo, 2=En proceso de desactivación).
- Recuento de interrupciones atendidas (número de veces que ha ejecutado la rutina `interrupt()`).
- Recuento de llamadas al sistema atendidas (número de veces que ha ejecutado la rutina `sys_call()`).
- Recuento de procesos planificador (número de veces que ha ejecutado la rutina `pick_proc()`).
- Estado del contador de reentradas al núcleo.
- Nombre del proceso que está ejecutando actualmente.

dmp.c

```
  PUBLIC void p_dmp()
  {
  /* Proc table dump */

    register struct proc *rp;
    static struct proc *oldrp = BEG_PROC_ADDR;
    int n = 0;
    phys_clicks text, data, size;
    int proc_nr;

  #if (ENABLE_MP == 1)
    printf("\n--pid --pc-CPU--sp- flag -user --sys-- -text- -data- -size- -recv-
command\n");
  #else
    printf("\n--pid --pc- ---sp- flag -user --sys-- -text- -data- -size- -recv-
command\n");
  #endif
    for (rp = oldrp; rp < END_PROC_ADDR; rp++) {
         proc_nr = proc_number(rp);
         if (rp->p_flags & P_SLOT_FREE) continue;
         if (++n > (21-MP_NR_CPUS)) break;
         text = rp->p_map[T].mem_phys;
         data = rp->p_map[D].mem_phys;
         size = rp->p_map[T].mem_len
                + ((rp->p_map[S].mem_phys + rp->p_map[S].mem_len) - data);
  #if (ENABLE_MP == 1)
         printf("%5d %5lx %c%6lx %2x %7U %7U %5uK %5uK %5uK ",
  #else
         printf("%5d %5lx %6lx %2x %7U %7U %5uK %5uK %5uK ",
  #endif
                proc_nr < 0 ? proc_nr : rp->p_pid,
                (unsigned long) rp->p_reg.pc,
  #if (ENABLE_MP == 1)
                (rp->p_currentcpu==NONE_CPU)?'-':'0'+rp->p_currentcpu,
  #endif
                (unsigned long) rp->p_reg.sp,
                rp->p_flags,
                rp->user_time, rp->sys_time,
```

```
                click_to_round_k(text), click_to_round_k(data),
                click_to_round_k(size));

        if (rp->p_flags & RECEIVING) {
                printf("%-7.7s", proc_name(rp->p_getfrom));
        } else
        if (rp->p_flags & SENDING) {
                printf("S:%-5.5s", proc_name(rp->p_sendto));
        } else
        if (rp->p_flags == 0) {
                printf("       ");
        }
        printf("%s\n", rp->p_name);
   }
   if (rp == END_PROC_ADDR) rp = BEG_PROC_ADDR; else printf("--more--\r");
   oldrp = rp;
#if (ENABLE_MP == 1)
   printf("CPU Enab.  interr. scall pickproc kreenter Task\n");
   for(n=0; n<MP_NR_CPUS; n++)
   printf("%1d    %1d %8d %8d %8d      %3d %s\n",
          n,cpu_available[n],interrupt_count[n],scall_count[n],pickproc_count[n],
          k_reenter[n],proc_ptr[n]->p_name);
#endif                        /* ENABLE_MP */
}
```

Para mantener el recuento de interrupciones, llamadas al sistema y planificaciones realizadas se han creado 3 vectores de contadores (enteros), localizados en `proc.h`, que se incrementan según el índice del procesador que las ejecuta, en las funciones `interrupt()`, `sys_call()` y `pric_proc()`.

proc.h

```
EXTERN int pickproc_count[MP_NR_CPUS];
EXTERN int scall_count[MP_NR_CPUS];
EXTERN int interrupt_count[MP_NR_CPUS];
```

5.18. Otras rutinas y definiciones

Se listan a continuación algunas rutinas y definiciones empleadas en la construcción del núcleo multiprocesador, así como el contenido del fichero Makefile donde se especifican las dependencias entre todos los ficheros fuente.

5.18.1. Definiciones de la Tabla de Configuración MP

Estas definiciones se utilizan para interpretar los campos de la Tabla de Configuración MP, y se han puesto en el fichero `mptable.h`.

mptable.h

```
/* Definitions of MP table and Floating Pointer Structure
   From Intel MP 1.4 specification */

#define SIGN_FPS        "_MP_"   /* MP signature of floating pointer struct */
#define FP_IMCRP_MASK   128

/* Describes a FPS (floating pointer structure) */
struct floating_pointer {
  char  fp_signature[4];    /* must be _MP_ */
  u32_t  fp_mp_table;             /* address to MP table */
```

```
  u8_t   fp_length;         /* FPS size in 16-byte parragraphs */
  u8_t   fp_version;        /* version number: 04h for 1.4 */
  u8_t   fp_cheksum;        /* bytes in FPS must sum 0 */
  u8_t   fp_sd_config_num;  /* standar config number (0 for none) */
  u8_t   fp_imcrp;          /* bit 7 is IMCR present and PIC mode */
  char   unused[3];         /* last 3 bytes are reserved */
};


#define SIGN_MP_C_HEADER  "PCMP"    /* signature for MP config table */

/* Describes the estruct of MP configuration table header */
struct mp_config_header {
   char  mpch_signature[4];        /* must be "PCMP" */
   u16_t mpch_length;              /* base table size including header */
   u8_t  mpch_version;             /* MP specification version number*/
   u8_t  mpch_checksum;            /* base table + checksum must sum 0 */
   char  mpch_oem_id[8];           /* manufacturer id (space filled) */
   char  mpch_product_id[12];      /* product id (space filled) */
   u32_t mpch_oem_table;           /* optional manufacturer table address */
   u16_t mpch_oem_table_size;      /* manufacturer table size */
   u16_t mpch_entry_count;         /* entry count in MP table */
   u32_t mpch_apic_addr;           /* IO address for local APIC */
   u16_t mpch_extended_length;     /* extended table length */
   u8_t  mpch_extended_checksum;   /* extended table checksum */
   char  mpch_reserved;            /* unused */
};


/* describes struct of a PROCESSOR entry type */
#define CPU_ENTRY_ID     0
#define CE_EN_FLAG_MASK  1
#define CE_BP_FLAG_MASK  2

struct cpu_entry {
   u8_t  ce_type;               /* 0 for this type */
   u8_t  ce_local_apic_id;      /* local APIC id number */
   u8_t  ce_local_apic_ver;     /* local APIC version */
   u8_t  ce_flags;              /* CPU enabled and CPU is bootstrap */
   u32_t ce_signature;          /* CPU type */
   u32_t ce_features;           /* features of this CPU */
   char  reserved[8];           /* reserved */
};


/* indentifies CPU type masking ce_signature field of a cpu_entry struct */
#define CPU_SIGN_STEPING_MASK  0x0000000F
#define CPU_SIGN_MODEL_MASK    0x000000F0
#define CPU_SIGN_FAMILY_MASK   0x00000F00
#define CPU_SIGN_MASK          CPU_SIGN_MODEL_MASK | CPU_SIGN_FAMILY_MASK

#define INVALID_CPU(signature)  (((signature & 0x00000FFF) == 0) ||        \
                                 ((signature & 0x00000FFF) == 0xFFF))

/* masking ce_signature with CPU_SIGN_MASK we can identify CPU type */
#define CPU_486DX      0x00000400
#define CPU_486DX_     0x00000410
#define CPU_486SX      0x00000420
#define CPU_487        0x00000430
#define CPU_DX2        0x00000430
#define CPU_486SL      0x00000440
#define CPU_SX2        0x00000450
#define CPU_DX4        0x00000480
#define CPU_P510_567   0x00000510
#define CPU_P735_815   0x00000520


/* masks bits of ce_featrues field of a cpu_entry struct */
```

```
#define CPU_FPU_MASK     1
#define CPU_MCE_MASK   128
#define CPU_CX8_MASK   256
#define CPU_ONCHIP_APIC_MASK  512

/* describes struct of a BUS entry type */
#define BUS_ENTRY_ID  1

struct bus_entry {
   u8_t be_type;              /* 1 for this type */
   u8_t be_id;                /* BUS type ID */
   char be_type_str[6];       /* manufacturer's description */
};

/* describes struct of a I/O APIC entry type */
#define IO_APIC_ENTRY_ID  2
#define AE_EN_FLAG_MASK   1
struct io_apic_entry {
   u8_t ae_type;              /* 2 for this type */
   u8_t ae_id;                /* IO APIC identification */
   u8_t ae_version;           /* IO APIC version number */
   u8_t ae_flags;             /* bit 0 means enabled */
   u32_t ae_address;          /* IO APIC memory address */
};

/* describes struct of a I/O INTERRUPT ASSING entry type */
#define IO_INT_ENTRY_ID  3
#define IE_PO_FLAG_MASK  3
#define IE_EL_FLAG_MASK  12

#define PO_BUS           0
#define PO_HIGH          1
#define PO_LOW           3

#define EL_BUS           0
#define EL_EDGE          1
#define EL_LEVEL         3

#define INT_INT          0
#define INT_NMI          1
#define INT_SMI          2
#define INT_EXTINT       3

struct io_int_entry {
   u8_t ie_type;              /* 3 for this type */
   u8_t ie_int_type;          /* iinterupt type */
   u16_t ie_flags;            /* bits 0-1 are PO and 2-3 are EL */
   u8_t ie_bus_id;            /* source bus */
   u8_t ie_bus_irq;           /* IRQ in source bus */
   u8_t ie_apic_id;           /* destination I/O APIC */
   u8_t ie_apic_int;          /* INTN in destination I/O APIC */
};


/* describes struct of a LOCAL INTERRUPT ASSING entry type */
#define LOCAL_INT_ENTRY_ID  4
#define LIE_PO_FLAG_MASK    3
#define LIE_EL_FLAG_MASK    12

struct l_int_entry {
   u8_t lie_type;              /* 4 for type type */
   u8_t lie_int_type;          /* interrupt type */
   u16_t lie_flags;            /* bits 0-1 are PO and 2-3 are EL */
   u8_t lie_bus_id;            /* source bus */
   u8_t lie_bus_irq;           /* IRQ in source bus */
   u8_t lie_apic_id;           /* destination local APIC */
   u8_t lie_apic_int;          /* INTN in destination local APIC */
};
```

5.18.2. Rutinas de información del arranque multiprocesador

Estas rutinas, incluidas en el fichero `mpinfo.c`, muestran en pantalla, a través del mecanismo de información del arranque multiprocesador, los datos recogidos en la Tabla de Configuración MP. La invocación a muchas de estas rutinas se ha omitido durante el arranque para evitar exceso de información en pantalla.

mpinfo.c

```
/*************************************************************/
/* Print Floaing Pointer Structure information               */
/*************************************************************/
void print_fps_info(void){
   ADDS_MP_STATUS("\tMP specification version ");
   ADDN_MP_STATUS(fps.fp_version / 16 +1, 10, 0);
   ADDS_MP_STATUS(".");
   ADDN_MP_STATUS(fps.fp_version % 16, 10, 0);
   ADDS_MP_STATUS("\n\tStandard MP configuration ");
   ADDN_MP_STATUS(fps.fp_sd_config_num, 10, 0);
   if (! fps.fp_sd_config_num)
       ADDS_MP_STATUS(" (custom config)");
   ADDS_MP_STATUS("\n\tIMCR&PIC mode ");
   if (!(fps.fp_imcrp & FP_IMCRP_MASK))
       ADDS_MP_STATUS("not ");
   ADDS_MP_STATUS("present\n");
   ADDS_MP_STATUS("\tMP config table at 0x");
   ADDN_MP_STATUS(fps.fp_mp_table,16,0);
   ADDS_MP_STATUS("\n");
}


/*************************************************************/
/* Print MP table information                                */
/*************************************************************/
void print_mp_configuration(void) {
   char buffer[13];
   int i;

   ADDS_MP_STATUS("\tMP especification version ");
   ADDN_MP_STATUS(mph.mpch_version / 16 +1, 10,0);
   ADDS_MP_STATUS(".");
   ADDN_MP_STATUS(mph.mpch_version % 16, 10,0);
   ADDS_MP_STATUS("\n\tOEM signarute ");
   for (i=0; i<8; i++) buffer[i]=mph.mpch_oem_id[i];
   buffer[i]=0;
   ADDS_MP_STATUS(buffer);
   ADDS_MP_STATUS(" product ID ");
   for (i=0; i<12; i++) buffer[i]=mph.mpch_product_id[i];
   buffer[i]=0;
   ADDS_MP_STATUS(buffer);
   ADDS_MP_STATUS("\n\tLocal APIC mem address: 0x");
   ADDN_MP_STATUS(mph.mpch_apic_addr, 16, 8);
   ADDS_MP_STATUS("\n\tMP table entry count: ");
   ADDN_MP_STATUS(mph.mpch_entry_count, 10, 0);
   ADDS_MP_STATUS("\n");
}


/*************************************************************/
/* Print I/O APIC information                                */
/*************************************************************/
void show_io_apic_info(void) {
```

```
   u32_t val;
   int i;

   ADDS_MP_STATUS("I/O APIC id");
   ADDN_MP_STATUS((IO_APIC_READ(IO_APIC_ID_REG) & IOA_ID_MASK)
                  >>IOA_ID_SHIFT, 10,0);
   ADDS_MP_STATUS(" version ");
   ADDN_MP_STATUS(((val=IO_APIC_READ(IO_APIC_VERSION_REG)) & IOA_VER_MASK)
                  >>IOA_VER_SHIFT, 10,0);
   ADDS_MP_STATUS(", ");
   ADDN_MP_STATUS((val & IOA_MXENT_MASK)>>IOA_MXENT_SHIFT,10,0);
   ADDS_MP_STATUS(" max. redirection entries, arbitration prior. at ");
   ADDN_MP_STATUS((IO_APIC_READ(IO_APIC_ARBITR_REG) & IOA_ARB_MASK)
                  >>IOA_ARB_SHIFT,10,0);
   ADDS_MP_STATUS("\n");

   for (i=0; i<=5; i++) {
      ADDS_MP_STATUS("Red.ent#");
      ADDN_MP_STATUS(i,10,2);
      ADDS_MP_STATUS("-");
      val=IO_APIC_READ(IO_APIC_REDTBL_REG_HIGH(i));
      ADDN_MP_STATUS(val,2,32);
      val=IO_APIC_READ(IO_APIC_REDTBL_REG_LOW(i));
      ADDN_MP_STATUS(val,2,32);
      ADDS_MP_STATUS("\n");
   }
}


/*************************************************************/
/* Print MP table entries information                        */
/*************************************************************/
#define PRINT_CPU_ENTRY(cpu)                                          \
            ADDS_MP_STATUS("CPU type ");                        \
            ADDN_MP_STATUS(ce.ce_signature,16,0);                     \
            ADDS_MP_STATUS(" APIC id");                               \
            ADDN_MP_STATUS(ce.ce_local_apic_id,10,0);                 \
            if (ce.ce_flags & CE_EN_FLAG_MASK) ADDS_MP_STATUS(" en"); \
                                       else ADDS_MP_STATUS("dis"); \
            ADDS_MP_STATUS("abled as ");                        \
            if (ce.ce_flags & CE_BP_FLAG_MASK) ADDS_MP_STATUS("BSP\n");      \
                                       else ADDS_MP_STATUS("AP\n")

#define PRINT_IO_APIC_ENTRY(ae)                                       \
            ADDS_MP_STATUS("I/O APIC id");                      \
            ADDN_MP_STATUS(ae.ae_id,10,0);                      \
            ADDS_MP_STATUS(" v");                               \
            ADDN_MP_STATUS(ae.ae_version,10,0);                       \
            if (ae.ae_flags & AE_EN_FLAG_MASK) ADDS_MP_STATUS(" en"); \
                                       else ADDS_MP_STATUS("dis"); \
            ADDS_MP_STATUS("abled at 0x");                      \
            ADDN_MP_STATUS(ae.ae_address,16,8);                       \
            ADDS_MP_STATUS("\n")

#define PRINT_IO_INT_ENTRY(ioie)                                \
           ADDS_MP_STATUS("IOInt:t");                                 \
           ADDN_MP_STATUS(ioie.ie_int_type,10,0);                     \
           ADDS_MP_STATUS("irq");                               \
           ADDN_MP_STATUS(ioie.ie_bus_irq,10,0);                \
           ADDS_MP_STATUS("->IOAPIC");                                \
           ADDN_MP_STATUS(ioie.ie_apic_id,10,0);                \
           ADDS_MP_STATUS("LINTIN");                                  \
           ADDN_MP_STATUS(ioie.ie_apic_int,10,0);                     \
           ADDS_MP_STATUS("\t\t")

#define PRINT_LOCAL_ENTRY(lie)                                        \
           ADDS_MP_STATUS("LOCAL INT ENTRY: type ");                  \
           ADDN_MP_STATUS(lie.lie_int_type,10,0);                     \
```

```
            ADDS_MP_STATUS(" irq  ");                                   \
            ADDN_MP_STATUS(lie.lie_bus_irq,10,0);               \
            ADDS_MP_STATUS(" conected to L.APIC ");                     \
            ADDN_MP_STATUS(lie.lie_apic_id,10,0);               \
            ADDS_MP_STATUS(" LINTIN");                                  \
            ADDN_MP_STATUS(lie.lie_apic_int,10,0);                      \
            ADDS_MP_STATUS("\n")

#define PRINT_MP_SUMMARY(cpu_c,bus_c,apic_c,ioint_c,lint_c)    \
          ADDS_MP_STATUS("\tSUMMARY MP info: ");                \
          ADDN_MP_STATUS(cpu_c,10, 0);                                  \
          ADDS_MP_STATUS(" CPUs, ");                                    \
          ADDN_MP_STATUS(bus_c,10, 0);                                  \
          ADDS_MP_STATUS(" BUS definition(s), ");                       \
          ADDN_MP_STATUS(apic_c,10, 0);                         \
          ADDS_MP_STATUS(" I/O APIC(s),\n\t\t");                \
          ADDN_MP_STATUS(ioint_c,10, 0);                        \
          ADDS_MP_STATUS(" I/O int. assignation(s), ");         \
          ADDN_MP_STATUS(lint_c,10, 0);                         \
          ADDS_MP_STATUS(" local int. assignation(s).\n")


/*********************************************************/
/* Print trampoline machine code and information     */
/*********************************************************/
void dump_trampoline(u32_t trampoline_addr, u32_t trampoline_sz) {
   int i;
   u8_t c;

   ADDS_MP_STATUS("Trampoline allocated at 0x");
   ADDN_MP_STATUS(trampoline_addr,16,8);
   ADDS_MP_STATUS("\n");

   for (i=0; i<trampoline_sz; i++) {
       phys_copy(trampoline_addr+i, vir2phys(&c),1);
       ADDS_MP_STATUS(" ");
       if (i % 26 == 0) ADDS_MP_STATUS("  ");
       ADDN_MP_STATUS(c,16,2);
   }
   ADDS_MP_STATUS("\n");
}
```

5.18.3. Fichero `Makefile`

Makefile

```
# Makefile for kernel

# Directories
u = /usr
i = $u/include
s = $i/sys
h = $i/minix
b = $i/ibm
l = $u/lib

# Programs, flags, etc.
CC =   exec cc
CPP =  $l/cpp
LD =   $(CC) -.o
CFLAGS = -m -I$i
LDFLAGS = -i

HEAD = mpx.o
```

```
OBJS = start.o protect.o klib.o table.o main.o proc.o \
       exception.o system.o clock.o memory.o tty.o keyboard.o \
       console.o i8259.o rs232.o dmp.o misc.o driver.o \
       drvlib.o floppy.o wini.o at_wini.o bios_wini.o esdi_wini.o \
       xt_wini.o printer.o aha_scsi.o dp8390.o pty.o \
       wdeth.o ne2000.o mcd.o sb16_dsp.o sb16_mixer.o \
       mp.o xmp.o

# What to make.
kernel: $(HEAD) $(OBJS)
       $(LD) $(LDFLAGS) -o $@ $(HEAD) $(OBJS)
       install -S 0 $@

all install:
       cd keymaps && $(MAKE) -$(MAKEFLAGS) $@

clean:
       cd keymaps && $(MAKE) -$(MAKEFLAGS) $@
       rm -f *.o *.bak kernel

# Dependencies
a =    kernel.h $h/config.h $h/const.h $h/type.h $h/syslib.h \
       $s/types.h $i/string.h $i/limits.h $i/errno.h \
       const.h type.h proto.h glo.h
d =    driver.h $h/callnr.h $h/com.h $h/partition.h proc.h
dl =   drvlib.h $b/partition.h
mp =   mp.h xmp.h

klib.o: $h/config.h $h/const.h const.h sconst.h protect.h
klib.o: klib88.s klib386.s
mpx.o: $h/config.h $h/const.h $h/com.h const.h protect.h sconst.h xmp.h
mpx.o: mpx88.s mpx386.s

clock.o:      $a
clock.o:      $i/signal.h
clock.o:      $h/callnr.h
clock.o:      $h/com.h
clock.o:      proc.h
clock.o:      $(mp)

console.o:    $a
console.o:    $i/termios.h
console.o:    $h/callnr.h
console.o:    $h/com.h
console.o:    protect.h
console.o:    tty.h
console.o:    proc.h

start.o:      $a
start.o:      $i/stdlib.h
start.o:      $h/boot.h
start.o:      protect.h

dmp.o:        $a
dmp.o:        $h/com.h
dmp.o:        proc.h
dmp.o:        $(mp)

exception.o:  $a
exception.o:  $i/signal.h
exception.o:  proc.h
exception.o:  $(mp)

driver.o:     $a $d
driver.o:     $s/ioctl.h
driver.o:     $(mp)
```

```
drvlib.o:     $a $d $(dl)
drvlib.o:     $(mp)

floppy.o:     $a $d $(dl)
floppy.o:     $b/diskparm.h

i8259.o:      $a

keyboard.o:   $a
keyboard.o:   $i/termios.h
keyboard.o:   $i/signal.h
keyboard.o:   $i/unistd.h
keyboard.o:   $h/callnr.h
keyboard.o:   $h/com.h
keyboard.o:   $h/keymap.h
keyboard.o:   tty.h
keyboard.o:   keymaps/us-std.src
keyboard.o:   $(mp)

main.o:       $a
main.o:       $i/unistd.h
main.o:       $i/signal.h
main.o:       $h/callnr.h
main.o:       $h/com.h
main.o:       proc.h
main.o: $(mp)

memory.o:     $a $d
memory.o:     $s/ioctl.h

misc.o:       $a
misc.o:       $i/stdlib.h
misc.o:       $h/com.h
misc.o:       assert.h

mp.o:  $a proc.h mpinfo.c mptable.h
mp.o:  $(mp)

xmp.o: $a
xmp.o: xmp.h
xmp.o: protect.h

printer.o:    $a
printer.o:    $h/callnr.h
printer.o:    $h/com.h
printer.o:    proc.h

proc.o:       $a
proc.o:       $h/callnr.h
proc.o:       $h/com.h
proc.o:       proc.h
proc.o: $(mp)

protect.o:    $a
protect.o:    proc.h
protect.o:    protect.h
protect.o: $(mp)

pty.o: $a
pty.o: $i/termios.h
pty.o: $i/signal.h
pty.o: $h/callnr.h
pty.o: $h/com.h
pty.o: tty.h
pty.o: proc.h

rs232.o:      $a
rs232.o:      $i/termios.h
```

```
rs232.o:      $i/signal.h
rs232.o:      tty.h
rs232.o:      proc.h

system.o:     $a
system.o:     $i/signal.h
system.o:     $i/unistd.h
system.o:     $s/sigcontext.h
system.o:     $s/ptrace.h
system.o:     $h/boot.h
system.o:     $h/callnr.h
system.o:     $h/com.h
system.o:     proc.h
system.o:     protect.h
system.o:     $(mp)

table.o:      $a
table.o:      $i/termios.h
table.o:      $h/com.h
table.o:      proc.h
table.o:      tty.h
table.o:      $(mp)

tty.o:        $a
tty.o:        $i/termios.h
tty.o:        $i/sgtty.h
tty.o:        $s/ioctl.h
tty.o:        $i/signal.h
tty.o:        $h/callnr.h
tty.o:        $h/com.h
tty.o:        $h/keymap.h
tty.o:        proc.h
tty.o:        tty.h
tty.o:        $(mp)

aha_scsi.o:   $a $d $(dl)
aha_scsi.o:   $i/fcntl.h
aha_scsi.o:   $s/mtio.h
aha_scsi.o:   $s/ioctl.h

wini.o:       $a $d $(dl)

at_wini.o:    $a $d $(dl)

bios_wini.o:  $a $d $(dl)

esdi_wini.o:  $a $d $(dl)

xt_wini.o:    $a $d $(dl)

mcd.o: $a $d $(dl)
mcd.o: $h/cdrom.h
mcd.o: $s/ioctl.h
mcd.o: $(mp)

dp8390.o:     $a
dp8390.o:     $i/stdlib.h
dp8390.o:     $h/com.h
dp8390.o:     $i/net/hton.h
dp8390.o:     $i/net/gen/ether.h
dp8390.o:     $i/net/gen/eth_io.h
dp8390.o:     assert.h
dp8390.o:     protect.h
dp8390.o:     dp8390.h
dp8390.o:     proc.h
dp8390.o:     $(mp)

wdeth.o:      $a
```

```
wdeth.o:      $i/net/gen/ether.h
wdeth.o:      $i/net/gen/eth_io.h
wdeth.o:      assert.h
wdeth.o:      dp8390.h
wdeth.o:      wdeth.h

ne2000.o:     $a
ne2000.o:     $i/net/gen/ether.h
ne2000.o:     $i/net/gen/eth_io.h
ne2000.o:     dp8390.h
ne2000.o:     ne2000.h

sb16_dsp.o:   $a
sb16_dsp.o:   $h/com.h
sb16_dsp.o:   $h/callnr.h
sb16_dsp.o:   $s/ioctl.h
sb16_dsp.o:   sb16.h

sb16_mixer.o: $a
sb16_mixer.o: $h/com.h
sb16_mixer.o: $h/callnr.h
sb16_mixer.o: $s/ioctl.h
sb16_mixer.o: $h/sound.h
sb16_mixer.o: sb16.h
```

6 Conclusiones y trabajo futuro

Una vez construido Minix SMP, podemos afirmar que la extensión de un sistema microkernel como Minix a una arquitectura SMP ha resultado relativamente sencilla. Bastan un conjunto de cambios sobre unas pocas líneas del código original de Minix y la escritura de unas cuantas nuevas para tener el sistema funcionando. Hemos visto que las modificaciones necesarias son bastante fáciles de comprender. Un sistema sencillo, con modificaciones sencillas da como resultado otro sistema sencillo, fácil de entender, extender, modificar y aprender: una herramienta ideal para utilizar en la docencia avanzada de Sistemas Operativos.

Si comparamos este trabajo con el realizado en su día para conseguir la versión SMP del núcleo de Linux *[3]*, observaremos que los principales pasos no difieren mucho, ni tampoco con los documentados para otros trabajos similares *[2]*. Esto no es ninguna casualidad, sino que viene dado por el gran parecido entre ambos sistemas y el hecho de que la arquitectura multiprocesador es como es, y no admite muchas variaciones. Las principales diferencias radican en el hecho de que Linux posea un núcleo monolítico.

Sin embargo, el que este trabajo resulte ahora sencillo de entender no significa que su implementación haya sido fácil. Todo lo contrario, principalmente debido al desconocimiento del funcionamiento preciso de la arquitectura subyacente. Estas dificultadas son la principal razón que nos conduce a reafirmar el interés de este trabajo. En su realización hemos invertido mucho esfuerzo, pero a cambio hemos obtenido una valiosísima experiencia en lo referente a las arquitecturas multiprocesador, en sus aspectos tanto hardware como software.

Esta experiencia se materializa en dos elementos muy importantes. Por un lado, disponemos un sistema operativo con multiprocesamiento simétrico, sencillo, real, manejable y accesible. Una herramienta ideal para ser estudiada y para practicar con ella. Además, está basada (y en su mayor parte conserva la estructura original) en la herramienta docente ideal para los pasos previos al aprendizaje de conceptos avanzados de las arquitecturas SMP.

Por otra parte, el código fuente de Minix SMP constituye una documentación especializada en la materia. Abarca, los pasos necesarios para comprender el paso del sistema monoprocesador al SMP, así como la descripción del hardware multiprocesador, los problemas que plantea el multiprocesamiento desde el punto de vista del software, y otros detalles del hardware tradicional que es necesario conocer para abordar el problema.

No obstante, esta es la primera versión de Minix SMP. Existen infinidad de nuevos proyectos que emprender y detalles que mejorar. Por ejemplo, es deseable abordar la gestión avanzada de las interrupciones mediante el I/O APIC, que permite un reparto más inteligente de las interrupciones externas, por ejemplo, a los procesadores menos ocupados, en contraste con el controlador de interrupciones PIC i8259 utilizado hasta ahora, conectado al conjunto de procesadores, de manera que cada interrupción se distribuye a todos, y uno de ellos será el primero en reconocerla y ejecutarla.

Otros aspectos menos relevantes por mejorar se refieren al rendimiento de determinados algoritmos. Por ejemplo, el acceso a la identificación de procesador (APIC local) es un servicio utilizado con mucha frecuencia dentro del núcleo y requiere muchas instrucciones debido a que el acceso a la dirección de memoria del APIC local queda fuera del espacio de direccionamiento del núcleo. La dirección de acceso del APIC es configurable, por lo que sería deseable colocarla dentro del espacio del núcleo, lo que ahorraría una notable cantidad de tiempo, aunque no afectaría en nada a la codificación del núcleo.

El proyecto de Minix SMP no terminará con la construcción del sistema de multiprocesamiento simétrico. Aparte de los objetivos propuestos descritos en la introducción de este trabajo, Minix SMP servirá como punto de partida para seguir desarrollando sistemas operativos para arquitecturas modernas.

Uno de los primeros proyectos previstos es la extensión de Minix SMP con hilos POSIX *[14]*, lo que daría como resultado un sistema microkernel multiprocesador con soporte para hilos. Para ello se dispone de la experiencia del proyecto PONNHI *[17]*, una librería de Pthreads en espacio de usuario con arquitectura microkernel ya implementada para Minix, así como para otras arquitecturas, como los DSK de *Texas Instruments*.

Dicho sistema constituirá una magnífica plataforma de prueba donde ensayar la construcción de software para arquitecturas DSP. Nos permitirá disponer de un sistema sencillo y conforma a POSIX que proporcionará un entorno muy parecido al que encontramos en dicha arquitectura: múltiples procesadores sin memoria virtual.

Se pretende migrar *Kaffe [26]* (una máquina virtual Java) y *DISK [21]* (una máquina virtual Java DSM basada en *Kafee*) a la versión multihilo de Minix SMP, como paso previo a su migración a plataformas DSP. El objetivo final es disponer de una máquina virtual Java DSM para arquitecturas DSP multicomputador *[20]*, basada en primitivas POSIX.

7 Referencias

[1] Akl, S.G., ***Parallel Computation. Models and Methods***, Prentice Hall, 1997.

[2] Cammeresi, S., ***Bringing SMP to Your UP Operating System***. Copyright © 1998 by Sidney Cammeresi, http://www.cheesecake.org/~sac/smp.html.

[3] Cox, A. ***An Implementation Of Multiprocessor Linux***, Caldera Inc. 1995, http://kos.enix.org/pub/smp_linux.pdf.

[4] Deitel, H.M., ***Introducción a los Sistemas Operativos***. Addison-Wesley Iberoamericana, 1987.

[5] Intel Corporation, ***82374EB/82374SB EISA System Component (ESC)***. Intel Corporation, 1996.

[6] Intel Corporation, ***Intel Architecture Software Developer's Manual***. Intel Corporation, 1999.

[7] Intel Corporation, ***Intel Pentium III Processor Specification Update***. Intel Corporation, 2000.

[8] Intel Corporation, ***Multiprocessor Specification Version 1.4.*** Intel Corporation, 1997.

[9] ***Minix 2.0.0***, http://www.cs.vu.nl/pub/minix/2.0.0/.

[10] ***Minix 2.0.2***, http://www.cs.vu.nl/pub/minix/2.0.2/.

[11] ***Minix 2.0.3***, http://www.cs.vu.nl/pub/minix/2.0.3/.

[12] ***Minix RT***, http://www.sce.carleton.ca/faculty/wainer/rt-minix/rt-minix.html.

[13] ***Minix VMD***, http://www.minix-vmd.org.

[14] Mueller, F., ***A Library Implementation of POSIX Threads under UNIX***. In Proceedings of the USENIX Conference, 1993.

[15] Patterson, D.A., Hennessy, J.L., ***Organización y Diseño de Computadores. La Interfaz Hardware/Software***. McGraw-Hill, 1995.

[16] Ramesh . K.S., ***Design and development of MINIX distributed operating system***. Proceedings of the 1988 ACM sixteenth annual conference on Computer science, 1988.

[17] Rodríguez, J.M., Rico, J.A., Álvarez, J.M., Díaz, J.C., ***PONNHI. Una nueva arquitectura microkernel Pthreads en espacio de usuario***. XII Jornadas de Paralelismo, Edicions de la Universitat de Lleida, 2002.

[18] Schimmel, C., ***UNIX Systems for Modern Architectutes***. Addison-Wesley, 1994.

[19] Singhal, M., Shivaratri, N., ***Advanced Concepts in Operating Systems: Distributed, Database, and Multiprocessor Operating Systems***, McGraw-Hill, 1994

[20] ***Sundance Multiprocessor Technology Ltd.***, http://www.sundance.com.

[21] Surdeanu, M., Moldovan, D., ***Design and Performance Analysis of a Distributed Java Virtual Machine***. IEEE Transactions on Parallel and Distributed Systems, Vol. 13, No. 6, pp. 611-627, June 2002

[22] Tanenbaum, A.S. et al. ***Minix 1.5 Reference Manual***. Prentice Hall, 1991.

[23] Tanenbaum, A.S., ***Operating Systems, Design and Implementation, 2nd Edition***, Prentice-Hall, 1997.

[24] Tanenbaum, A.S., ***Operating Systems, Design and Implementation***. Prentice Hall, 1987.

[25] Tanenbaum, A.S., ***Sistemas Operativos Modernos***. Prentice Hall, 1992.

[26] ***The Kaffe Project***, http://www.kaffe.org.

[27] Tobin, P., ***Design considerations for the transformation of MINIX into a distributed operating system***. Proceedings of the 1988 ACM sixteenth annual conference on Computer science, Atlanta, 1988.