CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS
DEL INSTITUTO POLITÉCNICO NACIONAL

UNIDAD ZACATENCO
DEPARTAMENTO DE COMPUTACIÓN

"Micronúcleo de sistema operativo para tiempo real sobre la arquitectura Intel de 32 bits"

Tesis que presenta

William Martínez Cortés

Para obtener el grado de

Maestro en Ciencias
en Computación

Directores de la Tesis:

Dr. Pedro Mejía Álvarez

Dr. Luis Eduardo Leyva del Foyo

México, Distrito Federal. **Noviembre, 2012**

Dedicatoria

A mis padres,
que a pesar de la distancia me acompañan en este momento tan importante y tan esperado para ellos.

A mi abuela Batistina,
que siempre se ha preocupado por mí.

A la memoria de mi abuelo Pablo Martínez Jarque,
quien siempre se mostró muy orgulloso de mí.

Con amor a Janet Rodríguez Bufanda.

Agradecimientos

Ante todo agradezco a mis padres, por su preocupación y por el apoyo que siempre me han brindado, sin el cual hubiese sido imposible la realización de estos estudios.

Quiero realizar un agradecimiento especial a mi tutor Luis E. Leyva del Foyo, quien me ha servido de guía desde mis años de estudios de licenciatura.

A mi asesor Pedro Mejía, por al apoyo que me brindó durante todo el tiempo que duró la maestría, y no solo durante el trabajo de tesis.

Al CINVESTAV y en particular al Departamento de Computación, por brindarme la oportunidad de estudiar en tan prestigioso centro.

Al Consejo Nacional de Ciencia y Tecnología (CONACyT), por el apoyo económico brindado.

A Liliana Puente Maury, por su amistad y sus sugerencias para mejorar el presente documento.

A todos mis compañeros de grupo, y en especial a Pau, Julio Cesar, Daniel, Ana Helena, y Antonio Pico, que a pesar de las diferencias culturales siempre me hicieron sentir como uno mas de ellos.

A Janet Rodríguez Bufanda, por acompañarme, apoyarme, y por enseñarme facetas de México desconocidas para mí.

A todos los hombres y mujeres comprometidos con la ciencia, por servirme de inspiración y ejemplo.

A todos lo que de una forma u otra han ayudado a la realización de este trabajo, o me han apoyado durante mi estancia en este hermoso país llamado México,

Muchas gracias.

Contenido

Índice de figuras

Índice de tablas

Resumen

En 2003 fue creado un micronúcleo experimental de sistema operativo de tiempo real, denominado PARTEMOS. Este micronúcleo ha servido de plataforma de prueba para muchas investigaciones en el área de sistemas operativos y de tiempo real, contando con características avanzadas como su novedosa técnica de manejo de interrupciones. El micronúcleo fue programado para ejecutarse en el modo real de 16 bits, disponible en los procesadores 80x86 y compatibles, lo que unido al uso de un modelo de memoria "small" limitaba todo el sistema a 64 KB de memoria para código y otros 64 KB para datos. En el presente trabajo, el código de PARTEMOS fue adaptado para poder ejecutarse en el modo protegido de 32 bits de los procesadores 80386 y superiores, como método para obtener una versión fácilmente portable del mismo e identificar los pasos necesarios para portar PARTEMOS a otras plataformas. Adicionalmente, esta adaptación permite la ejecución del sistema en 32 bits sin restricciones de memoria.

Aunque la arquitectura de PARTEMOS fue diseñada pensando en su portabilidad, el sistema nunca había sido portado a otras plataformas, y el código fuente del micronúcleo contenía muchas secciones no portables. Para reducir este problema, el código del sistema fue sometido a modificaciones que mejoraron su portabilidad. Adicionalmente se realizaron cambios para mejorar la eficiencia y hacer más simples algunas secciones de código. La nueva versión de PARTEMOS obtenida no solo puede ejecutarse en el modo protegido de 32 bits, sino que mantiene la capacidad de ejecutarse en la plataforma de 16 bits original, constituyendo la primera versión multiplataforma de este sistema. Adicionalmente, la nueva versión de 32 bits de PARTEMOS es capaz de utilizar el controlador de interrupciones programable avanzado (APIC), como alternativa al clásico PIC 8259, para administrar las interrupciones del sistema.

Para poder desarrollar la versión de 32 bits de PARTEMOS, fue necesario hacerse de las herramientas de desarrollo adecuadas y de sus técnicas de uso, lo que resultó en una metodología para el desarrollo de sistemas operativos de 32 bits. También se crearon metodologías para el desarrollo de aplicaciones sobre PARTEMOS, y para portar PARTEMOS a otras arquitecturas de hardware. Entre otros logros de este trabajo podemos resaltar la asimilación de conocimiento acerca del funcionamiento de la arquitectura Intel de 32 bits, y de los controladores APIC; conocimiento que en buena medida ha quedado plasmado en el presente documento de tesis.

Abstract

In 2003, an experimental real time operating system microkernel, called PARTEMOS, was created. This microkernel has been used as a testing ground for several investigations in the operating systems and real-time areas, having advanced features like a novel technique of interrupt handling. The microkernel was built to run in 16-bit real mode, available in 80x86 and compatible processors; this, in conjunction with use of a “small” memory model, limits the system to 64 KB of memory for code and 64 KB for data. In this work the PARTEMOS code was adapted to run in the 32-bit protected-mode of the 80386 and above processors, as a means to obtain an easily portable version of it, and identify the steps required to port PARTEMOS to other platforms. Additionally, this adaptation allows the 32 bits system execution without memory restrictions.

Although the PARTEMOS architecture was designed thinking in its portability, the system had never been ported to other platforms, and the microkernel source code contained many not portable sections. To reduce this problem, the system code was modified to improve its portability. Further changes were made to improve the efficiency and simplify some sections of code. The new obtained version of PARTEMOS not only can run in the 32-bit protected mode, it also retains the ability to run on the original 16-bit platform, so it is the first multiplatform version of this system. Additionally, the new 32-bit version of PARTEMOS can use the Advanced Programmable Interrupt Controller (APIC), as an alternative to the classic PIC 8259 to manage system interrupts.

In order to develop the 32-bit version of PARTEMOS, it was necessary to take appropriate development tools and the techniques to use them, resulting in a methodology for 32-bit operating system development. Also, methodologies for application development on PARTEMOS and for PARTEMOS porting were created. Among other results of this work we have the knowhow assimilation about the internals of the Intel 32-bit architecture, and the APIC controllers; knowledge that has largely been captured in the present document.

Keywords: Real Time Operating System (RTOS), Embedded Systems, Interrupt handling, 32 bits Intel Architecture (IA-32), Hardware Abstraction layer (HAL), Advanced Programmable Interrupt Controller (APIC)

Capítulo 1
Introducción

En el año 2003 se diseñó un micronúcleo de sistema operativo de tiempo real, denominado PARTEMOS, con el objetivo de servir de plataforma de prueba para investigaciones en el área de sistemas operativos y sistemas de tiempo real. Este micronúcleo experimental, cuya área de aplicación fundamental son los sistemas embebidos y de tiempo real, fue diseñado por Luis E. Leyva del Foyo, e implementado por este investigador junto con otros colaboradores. La creación de PARTEMOS fue de gran importancia para demostrar la efectividad de diferentes propuestas científicas, descritas en varios artículos de impacto internacional [Leyva-del-Foyo y Mejia-Alvarez 2004] [Leyva-del-Foyo *et al.* 2006a] [Leyva-del-Foyo *et al.* 2006b]. Una de las características mas notables de PARTEMOS es su novedosa técnica de manejo de interrupciones, que es tratada más adelante en este documento, y lo convierte en un sistema operativo muy conveniente para la implementación de sistemas de tiempo real. Sin embargo la implementación realizada de PARTEMOS adolece de ciertos defectos que atentaban seriamente contra su utilización práctica.

Debido a la enorme variedad de plataformas de cómputo para dispositivos embebidos existente actualmente, es deseable que los sistemas operativos para estos dispositivos tengan la característica de ser fácilmente portables. Aunque la arquitectura de PARTEMOS fue creada pensando en su portabilidad, hasta la fecha **nunca se había intentado portar su código** para ejecutarlo en otra plataforma, por lo que podía considerarse que este **era un sistema monoplataforma**. La implementación de este sistema (existente antes de la realización del presente trabajo de tesis) contenía muchas secciones de código dependientes de la plataforma de hardware, de la plataforma software de ejecución (MS-DOS), y del compilador usado para generar el sistema (Borland C), lo que **dificultaba el proceso de portarlo a otras plataformas**. Por lo anteriormente mencionado, se decidió que el código de **PARTEMOS debía ser portado a otra plataforma, como un ejercicio práctico con la finalidad de identificar las secciones dependientes de la plataforma, mejorar su portabilidad, e identificar un procedimiento a seguir para portarlo a cualquier otra plataforma**. En lo adelante llamaremos **PARTEMOS16** a esta versión del sistema PARTEMOS, existente antes de la realización del presente proyecto de tesis, cuyo código contenía muchas secciones no portables.

El micronúcleo PARTEMOS fue diseñado para ejecutarse sobre MS-DOS, en computadoras con procesadores x86 y compatibles. Todo el sistema (incluyendo el núcleo y las aplicaciones) era compilado utilizando un modelo de memoria “small” de 16 bits, quedando limitado a un segmento de memoria (64 KB) para código, y otro segmento para datos y pila. Esta cantidad de memoria **es insuficiente para muchas aplicaciones modernas**, más aún si tenemos en cuenta que el propio PARTEMOS utiliza una parte importante del segmento de código.

Muchos dispositivos de cómputo embebido modernos cuentan con gran cantidad de memoria, del orden de varios megabytes e incluso de más de 1 GB. Para poder acceder a esta memoria, los procesadores de estos dispositivos utilizan un modo de direccionado de 32 bits. Es por esto que una solución lógica a las limitaciones de memoria de PARTEMOS16 es **portar su código para que pueda ejecutarse en un procesador de 32 bits.** La adaptación del código PARTEMOS a una arquitectura de 32 bits puede

realizarse de forma que posteriormente sea relativamente sencillo portarlo a otra; permitiendo que el sistema pueda **utilizarse sin restricciones de memoria en diferentes dispositivos de cómputo modernos.**

Una de las plataformas de 32 bits más abundantes y conocidas son las computadoras personales con procesadores Intel o compatibles, las que constituyen una elección lógica para un primer trabajo de portado de PARTEMOS. Por otro lado el desarrollo de sistemas operativos sobre esta plataforma (y otras de 32 bits en general) es un proceso complicado del que no se dispone de mucha documentación, por lo que es necesario realizar una investigación que concluya con **la selección y documentación de las técnicas y herramientas adecuadas para el desarrollo de sistemas operativos** en esta plataforma.

1.1 Objetivos

De lo escrito anteriormente podemos concluir que el objetivo general de este proyecto de tesis es **portar el código del sistema operativo PARTEMOS a la arquitectura Intel de 32 bits como medio para obtener una versión del mismo fácilmente portable a otras plataformas**.

Del objetivo general antes expuesto se derivan varios objetivos particulares, que enumeramos a continuación:

1. **Crear un entorno para el desarrollo de sistemas operativos de 32 bits**. El desarrollo de sistemas sobre 32 bits es una tarea que presenta cierto grado de dificultad y de la cual se posee poca documentación. Por eso es importante que al término de este trabajo queden documentadas las técnicas y herramientas necesarias para este desarrollo.
2. **Mejorar la portabilidad de la capa de abstracción de hardware (HAL) de PARTEMOS.** La capa HAL de PARTEMOS16 estaba completamente implementada en ensamblador, lo que la hacía poco portable y difícil de mantener. Aunque por su función ésta es precisamente la menos portable de las capas, si se utiliza el lenguaje de programación C para implementar la mayor parte de esta se puede lograr un aumento en la portabilidad y mantenibilidad de la misma. Este punto implica también un posible rediseño del código de esta capa, tratando de identificar y separar las secciones de código que pueden ser portadas directamente (o con muy poco esfuerzo) a otras plataformas.
3. **Crear una metodología para portar PARTEMOS a otras arquitecturas.** Para que un sistema operativo para dispositivos embebidos tenga éxito necesita contar con versiones para muchas plataformas de hardware, o al menos ser fácilmente portable a nuevas plataformas. Una vez adaptado el código de PARTEMOS para ejecutarse en arquitecturas Intel de 32 bits, el código resultante debe ser fácilmente portable a otras plataformas. La experiencia de adaptación de PARTEMOS debe quedar documentada de forma que quede sistematizada la tarea de adaptar el código de PARTEMOS a otras arquitecturas de hardware.
4. **Utilizar el APIC para la implementación del HAL de interrupciones.** El código de PARTEMOS contiene un módulo, denominado HAL de interrupciones (ó INTHAL), que brinda ciertos servicios al núcleo relacionados con el manejo de interrupciones. Actualmente el HAL de interrupciones de PARTEMOS se basa en el uso del controlador de interrupciones PIC8259 o compatibles; sin embargo, en las modernas computadoras x86, el controlador de interrupciones avanzado (APIC)

[Intel 2012] ha remplazado grandemente al uso del 8259. Por este motivo, además de implementarse la versión "clásica" del INTHAL, nos hemos planteado como objetivo escribir otra versión del mismo soportada por el controlador APIC, para evaluar su factibilidad de uso como controlador de interrupciones de PARTEMOS.

5. **Crear una metodología para el desarrollo de aplicaciones sobre PARTEMOS.** Un sistema operativo debe contar con aplicaciones que se ejecuten sobre el mismo. Por eso es importante que quede documentado el proceso de crear nuevas aplicaciones para PARTEMOS, incluyendo herramientas para compilar y ejecutar aplicaciones, y sus técnicas de uso.

1.2 Resultados del trabajo y actividades desarrolladas

A pesar de que PARTEMOS16 cuenta con estructuras que fueron creadas pensando en su portabilidad (como la capa de abstracción de hardware, HAL), su código contiene ciertas características dependientes de la plataforma para la cual fue escrito. Como resultado de este trabajo, el código fuente disponible fue mejorado en algunos puntos, y restructurado para hacerlo fácilmente portable a otras plataformas. La nueva versión del sistema obtenida, que llamaremos **PARTEMOS32**, se puede compilar para ejecutarse en una PC con procesador que soporte la arquitectura Intel de 32 bits (que llamaremos **arquitectura IA-32**), ejecutándose en modo protegido de 32 bits. Adicionalmente, PARTEMOS32 mantiene la capacidad de ejecutarse en la plataforma original de 16 bits (que llamaremos abreviadamente **plataforma AT16**), constituyendo la **primera versión multiplataforma de PARTEMOS**. La figura 1 muestra como el código de PARTEMOS fue mejorado para dejar de ser un sistema monoplataforma y convertirse en un código portable, el cual es capaz de ejecutarse en dos plataformas diferentes, y puede ser fácilmente portado a otras plataformas de hardware. El proceso de adaptación de PARTEMOS a una nueva plataforma no solo mejoró la portabilidad del código, sino también permitió corregir deficiencias existentes en la implementación de PARTEMOS16, y aumentar la eficiencia de ejecución de ciertas secciones de código.

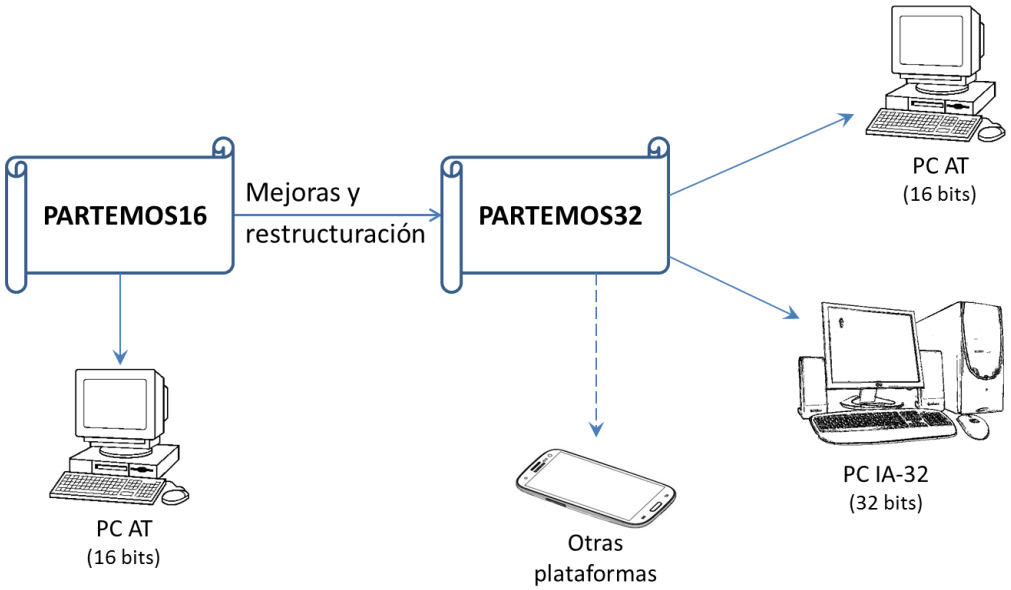


Figura 1: Plataformas soportadas por las versiones de PARTEMOS.

A continuación se exponen las principales tareas realizadas durante la ejecución de este proyecto. Estas se listan aproximadamente en el orden en que fueron realizadas, aunque en muchos casos los períodos de ejecución de las mismas se solaparon en el tiempo, y algunas tareas fueron retomadas luego de haberse trabajado en tareas posteriores, formando algunas iteraciones en el cronograma de trabajo.

1. **Determinación de técnicas y herramientas de desarrollo.** Durante esta fase se determinó fundamentalmente cuales herramientas de desarrollo de software (compiladores, ensambladores, etc.) debían ser utilizadas para generar la imagen binaria del núcleo. En este punto también se determinó la técnica utilizada para cargar el sistema en memoria y ejecutarlo.
2. **Determinación de las técnicas y herramientas de depuración.** La introducción de errores de programación es un hecho inevitable durante el desarrollo de cualquier software más o menos complejo, por eso se hizo necesario desarrollar técnicas para detectar y localizar los errores lo más fácilmente posible. Esta fase está relacionada con la elección de depuradores (*debuggers*) y la creación de técnicas de registro de eventos. En este punto también se incluye la determinación de herramientas de virtualización de hardware, ya que estas permiten una fácil ejecución del sistema compilado, y pueden incluir algún tipo de soporte para depuración.
3. **Implementación del HAL de interrupciones (INTHAL) sobre el PIC8259.** Esta tarea tuvo como objetivo obtener una versión de 32 bits del HAL de interrupciones "clásico" sobre el controlador PIC8259, con las dependencias mínimas necesarias para funcionar. Se dio preferencia al uso del lenguaje C sobre el ensamblador donde era razonablemente posible, cumpliendo con el objetivo de mejorar la portabilidad y mantenibilidad de la capa HAL.
4. **Implementación de toda la capa de abstracción de hardware (HAL).** Este paso extiende al anterior e implicó realizar toda la capa HAL del sistema para la arquitectura IA-32, incluyendo la ejecución de pruebas de software sobre esta capa.
5. **Adaptación del resto del sistema a 32 bits.** Todo el código C del resto del sistema fue adaptado para permitir su compilación y ejecución sobre la arquitectura Intel de 32 bits, eliminándose o encapsulándose las dependencias de la plataforma de hardware y software sobre la que se ejecutaba la versión anterior del sistema, y el uso de características no portables del compilador Borland C.
6. **Evaluación del uso de controladores APIC en PARTEMOS.** Se realizó un estudio del funcionamiento y utilización de los controladores de interrupciones APIC, evaluándose su factibilidad para la implementación del modo de manejo de interrupciones de PARTEMOS; y se codificaron algunas alternativas de implementación del módulo HAL de interrupciones de PARTEMOS sobre APIC.

1.3 Estructura del documento de tesis

El resto del presente documento de tesis está estructurado de la siguiente manera:

El **capítulo 2** sirve de introducción a los sistemas operativos de tiempo real, brindando una panorámica de los sistemas existentes. Este capítulo también describe la forma en que se manejan las interrupciones

en muchos sistemas operativos modernos. Por último se brinda una descripción general de la arquitectura de PARTEMOS, y se describe su novedosa técnica de manejo de interrupciones, comparándola con la forma de manejo de otros sistemas operativos.

En el **capítulo 3** se describen las características de la plataforma de ejecución de la versión de 32 bits de PARTEMOS, incluyendo algunos detalles del funcionamiento de los procesadores Intel de 32 bits, y de los diferentes controladores de interrupción existentes en esta plataforma.

El **capítulo 4** describe la selección, creación y uso de un entorno de desarrollo para sistemas operativos sobre la arquitectura IA-32. Este entorno de desarrollo está formado por un conjunto de herramientas necesarias para generar la versión de 32 bits de PARTEMOS, y las técnicas de uso de estas herramientas.

Los **capítulos 5 y 6** describen el trabajo realizado sobre el código de PARTEMOS, cuyo objetivo fundamental fue obtener una versión portable de este sistema, pero no se limitó a esto. El trabajo fue dividido en trabajo realizado sobre la capa de abstracción de hardware (HAL, **capítulo 5**) y el trabajo sobre el resto del sistema (**capítulo 6**). En estos capítulos se describen además los pasos que deberían seguirse para portar la capa HAL y el resto del sistema (respectivamente) a otras plataformas de hardware.

El **capítulo 7** describe, utilizando una aplicación de ejemplo, como se pueden desarrollar aplicaciones que se ejecuten en PARTEMOS sobre la plataforma de 32 bits elegida, incluyendo los pasos necesarios para compilar la aplicación, y la forma de ejecutarla.

Por último, en el **capítulo 8** se realiza una valoración crítica del trabajo realizado, mostrando sus logros y limitaciones. También se resumen las ventajas y limitaciones de la versión de PARTEMOS actualmente disponible.

En general existe poca dependencia entre los capítulos mostrados, por lo que no es obligatoria la lectura secuencial de los mismos. En algunos casos se incluyen referencias a otras secciones que contienen más detalle de alguna información mencionada.

Capítulo 2
Panorámica de los sistemas operativos para sistemas de tiempo real

El presente capítulo sirve de introducción a conceptos como sistema embebido y de tiempo real, y sistemas operativos de tiempo real (RTOS – *Real Time Operating System*). En el capítulo se describen algunos de los principales sistemas operativos utilizados en el área de tiempo real, mostrando diferentes formas de abordar el manejo de interrupciones de hardware en estos sistemas. Por último, se brinda una descripción del RTOS PARTEMOS, haciendo énfasis en una de sus características más importantes: el manejo de interrupciones.

2.1 Sistemas embebidos y de tiempo real

Denominamos **sistema embebido** a un sistema de cómputo con un alto grado de integración entre el hardware y el software, construido para una aplicación específica [Li y Yao 2003], y que constituye parte integral de un sistema más grande. Los sistemas embebidos no son vistos por sus usuarios como una computadora, sino como un artículo de propósito específico, como por ejemplo un teléfono celular o una cámara digital. Los sistemas embebidos incorporan un conjunto de características significativas que los diferencian de los sistemas de escritorio. Muchas veces los sistemas embebidos están muy limitados en memoria, potencia de cómputo, consumo de energía, y en muchos casos están sometidos a restricciones de tiempo real.

Un **sistema de tiempo real** es un sistema de cómputo con restricciones de tiempo explícitas, de tipo deterministas o probabilísticas. En otras palabras, la corrección del sistema depende no sólo de la corrección lógica de los resultados del cómputo, sino también del tiempo en que estos resultados son producidos [Stankovic 1988]. Cada resultado debe darse en un plazo de tiempo específico, que de incumplirse el producto final se considera erróneo.

De forma general las restricciones de tiempo en los sistemas de tiempo real pueden ser arbitrariamente complicadas. Pero lo más común es que estás se simplifiquen y clasifiquen en plazos de cumplimiento de tiempo que pueden ser de tipo **duro** (*hard*), **firme** (*firm*), o **suave** (*soft*). Un plazo se denomina **duro** si su incumplimiento puede provocar un fallo total en el sistema, llegando en casos mas graves a desencadenar consecuencias catastróficas como pérdidas de vidas humanas y materiales. Un plazo de tiempo real **firme** es aquel cuyo incumplimiento puede ser tolerado, aunque usualmente esto acarrea una degradación de la calidad de servicio del sistema. El resultado de un cómputo de tiempo real **firme** que ha incumplido su plazo se considera inútil (en otras palabras la utilidad del resultado es cero). Un plazo **suave** es similar a uno firme, con la diferencia de que el resultado obtenido en los primeros mantiene cierta utilidad pasado su tiempo de cumplimiento (o sea, la utilidad del resultado vencido está entre cero y la utilidad máxima, obtenida de un resultado en tiempo).

De lo anterior podemos concluir que el objetivo general de un sistema de tiempo real es asegurar el cumplimiento de todos los plazos duros, y optimizar ciertos criterios específicos a la aplicación relativos a

los plazos firmes y suaves. Estos criterios pueden ser maximizar el número de plazos cumplidos, minimizar latencias, maximizar el cumplimiento de plazos para tareas de mayor prioridad, entre otros.

2.2 Sistemas operativos de tiempo real

Para la construcción de sistemas de tiempo real de cierta complejidad generalmente es necesario contar con sistemas operativos de propósito específico, denominados **sistemas operativos de tiempo real** (RTOS - *Real Time Operating System*), como plataforma de construcción. Los RTOS permiten que los sistemas sean programados como un conjunto de actividades independientes, denominadas tareas; al tiempo que brindan a sus usuarios un conjunto de facilidades para el desarrollo de aplicaciones. Generalmente los RTOS brindan servicios para la administración de tiempo, la manipulación de interrupciones, la gestión de memoria y la administración de entrada/salida, así como primitivas para la comunicación y sincronización entre tareas, etc.

Muchas de las responsabilidades de los RTOS son compartidas por otros tipos de sistemas operativos. Sin embargo, prácticamente todos los temas en los que se ha trabajado en los sistemas operativos de propósito general han sido reformados para poder garantizar el determinismo temporal de los RTOS.

El sistema operativo es responsable de administrar los recursos, y planificar la ejecución de las tareas. Sin embargo, este no se responsabiliza de la corrección temporal de las aplicaciones sino de su propio comportamiento, que debe ser determinista. Una de las condiciones necesarias para que exista determinismo es que se conozca el tiempo de ejecución en el peor caso de cada uno de los servicios que brinda el sistema, y que este tiempo sea acotado.

Formando parte esencial de todos los sistemas operativos de tiempo real encontramos al núcleo o kernel, responsable de la administración de tareas y la comunicación entre ellas. Todas las responsabilidades y servicios antes mencionados se encuentran concentrados en el núcleo del sistema operativo. Además del núcleo, los RTOS pueden incluir otros módulos, como puede ser módulos de comunicación TCP/IP, o módulos de gestión de sistema de archivos.

Para cumplir con los requerimientos de tiempo, tamaño y seguridad, los RTOS presentan un conjunto de características fundamentales:

- Poseen bajos requerimientos de memoria.
- Son muy reactivos a eventos.
- Brindan soporte para multitarea.
- Poseen un mecanismo de planificación con expropiación basado en prioridades.
- La mayoría de los servicios brindados tienen tiempo de ejecución acotado.
- Mantienen relojes de alta resolución.
- Brindan mecanismos de supervisión de los parámetros temporales.
- Brindan primitivas para realizar demoras en la ejecución por un tiempo fijo.

2.3 Sistemas operativos de tiempo real de amplio uso

El número de sistemas operativos diseñados para aplicaciones embebidas y de tiempo real existentes en la actualidad es muy grande, superando ampliamente a la cantidad de sistemas operativos usados en

computadoras personales o de propósito general. Mientras las computadoras personales (y con ellas sus sistemas operativos) han adoptado progresivamente las arquitecturas de 64 bits, en el mundo de los dispositivos embebidos y de tiempo real todavía abundan plataformas de 8, 16 y 32 bits. A continuación mostramos una breve descripción de algunos de los sistemas operativos mas usados en el área.

2.3.1 QNX Neutrino

En 1980, dos estudiantes de la universidad de Waterloo, convencidos de la existencia de una necesidad comercial de sistemas operativos de tiempo real, crearon la empresa "Quantum Software Systems" con el objetivo de crear y comercializar un sistema de este tipo. Dos años más tarde, lanzaron la primera versión de un sistema operativo llamado QNX, que se ejecutaba sobre una CPU Intel 8088. Posteriormente el núcleo de QNX sufrió varias rescrituras, hasta llegar a la última versión del sistema, llamada QNX Neutrino, la que fue lanzada en 2001.

QNX Neutrino[1] es un sistema operativo de tiempo real comercial, con **arquitectura de micronúcleo** (introducida por sistemas como Mach [Accetta *et al.* 1986]), dirigido primariamente al mercado de los sistemas embebidos. Fue diseñado desde cero para soportar multiprocesamiento simétrico (SMP), y cumple con todas las normas POSIX actuales. Desde septiembre de 2007, QNX ofrece una licencia para usuarios no comerciales.

QNX Neutrino sigue la filosofía de mantener en el micronúcleo, además de los servicios mínimos indispensables de un micronúcleo (como la conmutación de contexto y comunicación entre procesos), aquellos servicios cuyo tiempo de ejecución es muy pequeño. El micronúcleo de QNX neutrino brinda servicios para soportar las siguientes abstracciones:

- Hilos
- Pase de mensajes
- Señales
- Relojes
- Temporizadores
- Manipuladores de interrupción
- Semáforos
- Cierres de exclusión mutua (mutexes)
- Variables de condición (condvars)
- Barreras

Otras características POSIX que no son implementadas en el micronúcleo (como entrada-salida a dispositivos y archivos) son realizadas por procesos opcionales y bibliotecas compartidas. Todo lo que no está en el micronúcleo (incluyendo los drivers de dispositivos) se ejecuta en procesos independientes con memoria protegida, lo cual hace al sistema muy seguro y robusto ante fallos en un componente. La posibilidad de incluir o no estos elementos externos al micronúcleo hace de QNX neutrino un sistema muy escalable, pudiendo implantarse desde sistemas muy pequeños y restringidos, hasta en grandes sistemas con multiprocesamiento simétrico y varios gigabytes de memoria.

[1] http://www.qnx.com/products/neutrino-rtos/

El desarrollo de aplicaciones para QNX, al igual que muchos otros tipos de aplicaciones embebidas, se realiza desde una **plataforma cruzada**, esto significa que la máquina en que se desarrolla tiene una plataforma diferente a la máquina destino. Para desarrollar aplicaciones QNX se creó un ambiente de desarrollo basado en Eclipse, denominado "QNX Momentics Tool Suite", que incluye muchas herramientas de depuración, medición de desempeño, y desarrollo en general. El sistema operativo de desarrollo puede ser tanto Linux como Windows, y entre los procesadores destino podemos encontrar a ARM, MIPS, PowerPC, SH-4, y los de la familia x86.

Las diferentes versiones del sistema QNX siempre han tenido la reputación de ser muy estables, y han sido usadas en aplicaciones industriales, en sistemas telemáticos para la industria automotriz, aplicaciones multimedia, y algunos dispositivos médicos.

2.3.2 VxWorks

VxWorks es un sistema operativo de tiempo real desarrollado por Wind River Systems[1], y lanzado por primera vez en 1987. Este RTOS de licencia propietaria es uno de los más conocidos y utilizados, debiendo su fama en parte a su uso en muchas de las misiones espaciales de la NASA.

El núcleo de VxWorks soporta planificación por prioridades y round-robin, multiprocesamiento simétrico (SMP) y asimétrico (AMP), y gran variedad de plataformas. Entre las familias de procesadores soportadas tenemos los procesadores ARM y XScale (sólo versiones de 32 bits), ColdFire, IA-32, MIPS (versiones de 32 y 64 bits), y PowerPC. Aunque las versiones mas comunes de este sistema se ejecutan sobre procesadores de 32 bits, también existen versiones de 64 bits, realizadas para procesadores MIPS (de 64 bits) y para procesadores de la arquitectura x86-64, creada por AMD y producida por importantes fabricantes como Intel y la propia AMD.

VxWorks brinda una rica variedad de rutinas de comunicación entre procesos (IPC), implementando un API nativa y además un API compatible con varias normas POSIX, sólo disponible en modo usuario. Es de destacar que este sistema brinda la posibilidad de activar un mecanismo de herencia de prioridad en los semáforos de exclusión mutua (mutexes) como forma de evitar la inversión de prioridad no acotada. Esta opción se volvió muy conocida en el mundo científico cuando la misión de exploración a marte "Mars Pathfinder" comenzó a fallar, debido a que no se había habilitado el mecanismo de herencia de prioridad.

Este RTOS permite la instalación de **rutinas de servicio de interrupción** (ISR) por parte del usuario, las que pueden hacer uso de algunos servicios del sistema. En particular las ISRs no pueden utilizar ningún servicio que pueda causar el bloqueo del invocador, como la espera en un semáforo (**`semTake`**). Una acción permitida y que comúnmente se usa en las ISR es la señalización de un semáforo (**`semGive`**), con el objetivo de activar una tarea para dar tratamiento a la interrupción.

El sistema operativo VxWorks se ha utilizado en un gran número de aplicaciones embebidas y de tiempo real. Ha sido incluido en muchas misiones espaciales, una de las más recientes que podemos señalar es la misión "Mars Science Laboratory", mas conocida por su vehículo robótico "Curiosity". Muchos equipos de comunicaciones y redes usan VxWorks, incluyendo teléfonos, enrutadores, firewalls, módems entre

[1] http://www.windriver.com/products/vxworks/

otros. El sistema también se ha usado para controlar robots, incluyendo el conocido robot humanoide "ASIMO" de Honda, muchos robots industriales, y otros sistemas de control industriales y experimentales. Entre otras áreas de uso podemos señalar aplicaciones automotrices, aeronáuticas, militares, impresoras y equipos de procesamiento de imágenes, equipos médicos, entre otras.

2.3.3 Windows NT

Llamamos Windows NT a una familia de sistemas operativos propietarios de Microsoft que incluye sistemas conocidos comercialmente como Windows NT, Windows 2000, Windows XP, Windows Vista, Windows Server 2003, Windows 7, entre otros. El núcleo NT soporta multitarea con expropiación, y desde sus inicios fue diseñado para soportar multiprocesamiento simétrico.

Básicamente se puede considerar que Windows NT presenta una arquitectura de micronúcleo, aunque no cumple con todos los criterios de un micronúcleo puro. La documentación oficial denomina "Ejecutivo" a lo que típicamente se conoce como núcleo en otros sistemas operativos, utilizando el término "núcleo" para referirse a una parte muy pequeña del ejecutivo que sólo implementa un conjunto de mecanismos básicos.

El núcleo Windows NT se encuentra separado del hardware por una **capa de abstracción de hardware**, lo que facilita su portabilidad a diferentes arquitecturas de hardware. Entre las arquitecturas de CPU a las que se ha portado este sistema se encuentran Intel IA-32, MIPS R3000/R4000, Alpha, PowerPC, Itanium, AMD64 y ARM, aunque Microsoft dejó de dar soporte a las plataformas MIPS, Alpha y PowerPC luego del lanzamiento de Windows NT 4.0. Las plataformas en las que este sistema está mas difundido son los ordenadores personales PC, tanto en versiones de 32 como de 64 bits (procesadores IA-32 y x86-64). Las versiones de 32 bits de NT pueden ejecutar código legado de 16 bits, mientras las versiones de 64 bits son capaces de ejecutar aplicaciones Windows de 32 bits, además de las aplicaciones nativas de 64 bits.

Hasta la versión NT 4.0 la documentación oficial menciona que el núcleo NT es no expropiable [Solomon 1998]; esto significa que aunque el código del núcleo puede ser interrumpido temporalmente para ejecutar una ISR, esta siempre devolverá el control al núcleo sin expropiar al hilo actualmente en ejecución. Los servicios del núcleo son muy simples y se ejecutan muy rápidamente, por lo que el hecho de que el núcleo no sea expropiable ejerce poco impacto en la latencia de expropiación. En Windows NT, la mayor parte del código de sistema se encuentra en el ejecutivo, que se implementa mediante múltiples hilos y es completamente expropiable.

Este sistema operativo brinda una API de núcleo que se ejecuta en modo supervisor, y está muy poco documentada. La API del núcleo NT es muy rica, e incluye objetos como semáforos, mutexes, eventos, y varios tipos de temporizadores. Encima del API del núcleo se implementan otras APIs que se ejecutan en modo usuario, y son las que normalmente son usadas por las aplicaciones. Entre las APIs de usuario implementadas en Windows NT se encuentran Win32, POSIX y OS/2, aunque las dos últimas dejaron de ser soportadas a partir de Windows XP.

Windows NT es un sistema operativo de propósito general y como tal no fue diseñado para implementar sistemas de tiempo real. Sin embargo existen muchas aplicaciones que no tienen requerimientos de tiempo real duro, en las cuales Windows NT ha logrado buena aceptación. Esto se debe en parte a las facilidades que brinda, y también a que es un sistema muy conocido, documentado y por tanto atractivo

(en especial su API Win32 es muy conocida). Entre los usos que se han dado a Windows NT en sistemas embebidos podemos mencionar terminales como cajeros ATM y máquinas despachadoras de billetes de metro, dispositivos de red dedicados como enrutadores, y sistemas de automatización industrial.

Windows NT tiene muchas características técnicas que lo hacen atractivo para construir sistemas con requerimientos de tiempo real suave. NT está diseñado para un rápido despacho de eventos y manejo de interrupciones en dos niveles (ver sección 2.4, "Manejo de interrupciones en sistemas operativos de tiempo real"), y posee niveles de prioridad de "tiempo real", con más prioridad que los niveles usados comúnmente para programar aplicaciones. Su administrador de memoria brinda facilidades para las aplicaciones de tiempo real, por ejemplo la entrada/salida de páginas de memoria no interfiere con el procesamiento a prioridades de tiempo real, las aplicaciones pueden bloquearse en memoria para evitar que sean paginadas por el sistema, y existe la posibilidad de que diferentes procesos compartan una misma zona de memoria física, permitiendo una transferencia de datos muy rápida entre procesos cooperativos.

La empresa IntervalZero[1] comercializa una extensión para los sistemas Windows NT, denominada RTX, que permite la ejecución de aplicaciones de tiempo real duro, al tiempo que mantiene las ventajas brindadas por este sistema, como una API conocida, muchas facilidades de desarrollo, soporte de multiprocesamiento simétrico, independencia de la plataforma, y disponibilidad de hardware muy barato y potente.

2.3.4 Linux

Linux es una familia de sistemas operativos estilo UNIX de código libre, cuyo componente fundamental es el **núcleo Linux**, creado por Linus Torvalds y lanzado por primera vez en 1991. En sus inicios Linux estaba dirigido a computadoras personales basadas en procesadores Intel x86, y desde entonces ha sido portado a más plataformas de hardware que ningún otro sistema operativo. Actualmente Linux es el sistema operativo más usado en servidores y grandes sistemas como mainframes y supercomputadoras, y ha ganado mucha popularidad como sistema operativo para computadoras personales. Linux también está presente en muchos sistemas embebidos, tales como teléfonos móviles, tabletas, enrutadores de red, televisores, consolas de videojuegos, dispositivos de navegación GPS, sistemas de control y automatización industrial, instrumentos médicos, etc. El sistema operativo Android[2] de Google, basado en una versión modificada del núcleo Linux, ha alcanzado recientemente un gran éxito en el mercado de los teléfonos móviles inteligentes y de las tablets PCs.

El núcleo Linux presenta una arquitectura monolítica, aunque a diferencia de los núcleos monolíticos tradicionales, los controladores de dispositivos y las extensiones del núcleo se pueden cargar y descargar como módulos, sin necesidad de apagar el sistema. Hasta la versión 2.4, el núcleo Linux sólo permitía la expropiación de procesos en modo usuario. A partir de la versión 2.6, se agregó la posibilidad de interrumpir una tarea ejecutándose en modo núcleo, aunque no todas las secciones del núcleo pueden ser expropiadas.

[1] http://www.intervalzero.com/
[2] http://www.android.com/

La presencia de secciones no expropiables dentro de un gran núcleo monolítico hace que Linux rara vez ofrezca tiempos de respuesta predecibles, y que su capacidad de respuesta ante eventos externos sea generalmente lenta. Esto hace que Linux no pueda ser considerado un sistema operativo para aplicaciones de tiempo real duro, aunque se ha usado en muchas aplicaciones con requerimientos de tiempo real suave.

Se han creado muchos parches para utilizar Linux en aplicaciones de tiempo real, uno de los mas famosos es RTLinux. RTLinux puede considerarse un núcleo de RTOS que ejecuta a Linux como un hilo de menor prioridad que las tareas de tiempo real, de esta forma RTLinux puede ejecutar manejadores de interrupción y tareas de tiempo real sin verse demorado por el código Linux.

2.3.5 LynxOS

Este es un sistema operativo de tiempo real tipo UNIX creado por LynuxWorks[1], que proporciona compatibilidad completa con POSIX. Sus últimas versiones son compatibles con Linux ABI ("*Application Binary Interface"*), esto significa que los archivos binarios de aplicaciones Linux pueden ejecutarse en LynxOS sin necesidad de recompilar el código fuente original. La versión más reciente (LynxOS 5.0) soporta multiprocesamiento simétrico.

La primera versión de LynxOS fue escrita en 1986, y estaba destinada a una computadora con procesador Motorola 68010. Posteriormente, el sistema fue portado para ejecutarse en procesadores Intel 80386, y actualmente están soportadas las familias de procesadores ARM, PowerPC, MIPS, x86, y LEON3.

El sistema LynxOS está diseñado para aplicaciones de tiempo real duro. LynuxWorks posee una patente[2] tecnológica que es utilizada en este RTOS y describe el mecanismo de manejo de interrupciones y dispositivos de E/S que utiliza este sistema para mantener su desempeño de tiempo real duro. El método patentado rompe con el modelo tradicional de manejo de interrupciones, permitiendo que hilos estilo POSIX dentro del núcleo realicen el manejo de las interrupciones. LynxOS trata estos hilos de interrupción como hilos normales de usuario, con prioridades de software, no del hardware de interrupción, minimizando así la interferencia de interrupciones de baja prioridad sobre tareas de alta prioridad. El método patentado también permite que las tareas manejadoras de interrupción hereden su prioridad de la tarea de usuario que abra el dispositivo de E/S asociado.

LynxOS ha sido utilizado en muchas aplicaciones embebidas de tiempo real, incluyendo aplicaciones aeronáuticas y espaciales, en la industria militar, en el control de procesos industriales y las telecomunicaciones. Existe una versión especializada de LynxOS llamada LynxOS-178 (introducida en 2003), que cumple con los estándares de la industria para uso en aplicaciones aeronáuticas, y que mantiene una posición líder dentro de los sistemas aeroespaciales y militares.

2.3.6 eCos

El sistema operativo de tiempo real eCos[3] ("*Embedded Configurable Operating System"*), lanzado en 1986, es uno de los RTOS de código libre más populares actualmente, y está dirigido fundamentalmente

[1] http://www.lynuxworks.com/rtos/rtos.php
[2] http://www.google.com/patents?vid=5469571
[3] http://ecos.sourceware.org/

al desarrollo de aplicaciones embebidas. Muchas veces eCos es preferido a otros sistemas de código libre como Linux debido a que por ser altamente configurable, se puede adaptar a sistemas muy limitados en memoria. Entre los dispositivos que utilizan eCos como sistema operativo podemos citar los receptores de radio satelital Sirius, los módulos Wi-Fi de los Sony PlayStation 3, y los enrutadores NETGEAR.

El desarrollo de aplicaciones eCos puede realizarse desde máquinas con Linux o Windows. Existe una herramienta de configuración que permite a los desarrolladores especificar las características requeridas del sistema operativo, creando una versión específica del RTOS idealmente adaptada a los requerimientos de un sistema embebido particular. La tecnología de configuración intrínseca de eCos permite que este se pueda escalar desde sistemas extremadamente restringidos en memoria como dispositivos SoC ("*System on a Chip*") con decenas de kilobytes de memoria, hasta sistemas sofisticados que requieren niveles de funcionalidad mas complejos.

El núcleo eCos utiliza un modelo de aplicaciones que están formadas por un solo proceso con múltiples hilos. Durante la configuración, el desarrollador puede elegir entre dos planificadores diferentes, ambos con 32 niveles de prioridad. El planificador por mapa de bits ("*bitmap scheduler*") es un poco más eficiente y sólo permite un hilo por nivel de prioridad, mientras que el planificador por colas multinivel (MLQ, "*multilevel queue scheduler*") permite que múltiples hilos se ejecuten a la misma prioridad.

Para reducir la latencia de interrupción, eCos utiliza un esquema de manejo de interrupciones dividido en dos partes. La primera parte se conoce como "rutina de servicio de interrupción" (ISR), y la segunda como "rutina de servicio diferida" (DSR). Las DSRs pueden ejecutarse con interrupciones habilitadas, permitiendo que otras interrupciones (potencialmente de mayor prioridad) ocurran y sean procesadas. Para proteger del acceso concurrente a las estructuras de datos del planificador, eCos mantiene un contador usado para impedir la replanificación, de esta forma evita el tradicional método de deshabilitar las interrupciones en las regiones críticas, y por tanto reduce la latencia de despacho de interrupción.

El sistema eCos está diseñado para ser portable a un amplio rango de plataformas que incluyen microprocesadores (MPUs), microcontroladores (MCUs), y procesadores de señales digitales (DSPs), tanto de 16, 32, como de 64 bits. Todos los componentes del sistema están soportados por una capa de abstracción de hardware (HAL) , y por tanto se ejecutaran en cualquier plataforma para la cual se haya portado la capa HAL y se hayan creado los drives de dispositivos relevantes. Entre las arquitecturas soportadas por eCos podemos mencionar 68K/ColdFire, ARM, CalmRISC, Fujitsu FR-V y FR30, Hitachi H8/300, Intel IA-32, Matsushita AM3x, MIPS, NEC V8xx, PowerPC, SPARC, y SuperH.

Existe una distribución comercial de eCos denominada eCosPro. Esta distribución es creada por eCosCentric[1] e incorpora componentes de software propietario, así como soporte a plataformas no incluidas con la versión libre.

2.3.7 Windows CE

Windows CE (Ahora oficialmente conocido como Windows Embedded Compact) es un RTOS comercial desarrollado por Microsoft, cuya primera versión fue lanzada en 1996. Aunque tiene nombre similar, y

[1] http://www.ecoscentric.com/

expone la misma API Win32, Windows CE es un sistema operativo diferente (con un núcleo diferente) de las versiones de Windows para computadoras de escritorio y servidores.

Windows CE está optimizado para dispositivos con poca capacidad de almacenamiento; todo el núcleo requiere menos de un megabyte de memoria. Microsoft licencia este sistema a los fabricantes de dispositivos, los que pueden modificarlo y crear sus propias interfaces, para lo cual Windows CE proporciona las bases técnicas. A partir de la versión 7.0 (marzo de 2011), este sistema soporta procesadores multi-core con multiprocesamiento simétrico (SMP), actualmente soporta procesadores Intel x86 y compatibles, MIPS y ARM.

La versión 3.0 (junio de 2000) de este sistema sufrió una recodificación mayor, que lo hicieron seleccionable para la implementación de sistemas de tiempo real duro [Thomson y Browne 2002]. Esta versión tiene latencia de interrupciones determinista, y utiliza el protocolo de herencia de prioridad para evitar la inversión de prioridad no acotada. Windows CE 3.0 soporta planificación Round Robin y basada en prioridades con 256 niveles de prioridad, siendo su unidad fundamental de ejecución el hilo.

Windows CE maneja las interrupciones en dos pasos [Frampton 2002], primero el núcleo ejecuta una rutina de servicio de interrupción (ISR), que realiza un procesamiento mínimo y retorna un identificador de interrupción, que el núcleo utiliza para activar un hilo de servicio de interrupción (IST). Como segundo paso el IST realiza el resto del procesamiento de la interrupción. La mayor parte del procesamiento realmente ocurre dentro del IST.

A diferencia de otros sistemas operativos de Microsoft, buena parte de Windows CE se encuentra disponible en forma de código fuente. Esto se hizo así para que los fabricantes pudieran adaptar el código a su hardware. Sin embargo, algunos componentes centrales que no necesitan adaptación a un hardware específico (excepto el tipo de CPU) aún son distribuidos únicamente en forma binaria.

Existe un gran número de herramientas de desarrollo cruzado para Windows CE, la mayoría diseñadas para correr en computadoras de escritorio con sistema operativo Windows. Posiblemente la herramienta mas utilizada sea el paquete de desarrollo “Visual Studio” en sus muchas variantes.

Windows CE es la base de muchas plataformas de desarrollo, entre ellas las conocidas bajo el nombre de “Pocket PC”, “Windows Mobile”, “Smartphone”, “Portable Media Center”, y “Windows Phone”. El sistema ha sido fundamentalmente utilizado en dispositivos móviles interactivos, tales como pocket PCs , teléfonos móviles inteligentes y reproductores multimedia, aunque también a sido incluido en otros sistemas embebidos y dispositivos industriales.

2.4 Manejo de interrupciones en sistemas operativos de tiempo real

Las interrupciones de hardware son eventos asíncronos generados por dispositivos externos a la CPU. Estos eventos pueden ocurrir en cualquier momento sin importar el código actualmente en ejecución. Las interrupciones de hardware son de gran importancia en los sistemas de tiempo real, ya que los hace más reactivos a eventos externos. Sin embargo, el esquema de manejo de interrupciones encontrado en muchos sistemas operativos no siempre es el más conveniente para implementar sistemas de tiempo real.

Los sistemas operativos actuales generalmente tienen dos espacios de prioridades independientes: el de las interrupciones y el de las tareas (ver figura 2). En el espacio de prioridades de interrupción normalmente encontramos a las rutinas de servicio de interrupción (ISRs). Las ISRs tienen más prioridad que las tareas y su prioridad relativa es impuesta por el hardware de interrupciones. Las prioridades de las tareas son modificables por software y manejadas por el sistema operativo. Este esquema de manejo de interrupciones es fuente de impredictibilidad e introduce latencias en el manejo de las interrupciones, como se explica mas adelante.

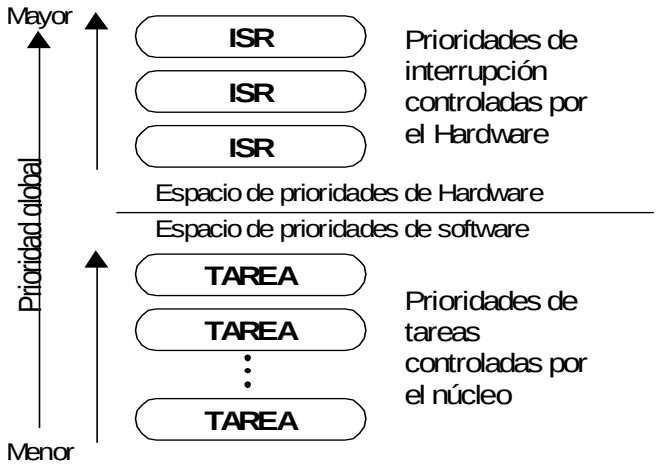


Figura 2: Esquema de prioridades en un sistema multitarea tradicional.

Las interrupciones son fuente de concurrencia en el sistema, lo que implica que las actividades asíncronas en el sistema deban sincronizarse para evitar que el acceso concurrente a estructuras de datos del núcleo las deje en un estado inconsistente. En los sistemas concurrentes generalmente se distinguen dos tipos de actividades asíncronas, las interrupciones y las tareas (procesos, hilos). Las tareas se pueden sincronizar entre sí mediante el uso de primitivas de sincronización; sin embargo, la forma usual de sincronizar los manejadores de interrupción entre sí o con las tareas consiste en deshabilitar la ocurrencia de interrupciones. Al impedirse temporalmente que el núcleo atienda las interrupciones que ocurran, se introduce una latencia en el manejo de interrupciones, lo cual es indeseable porque atenta contra la capacidad de reacción del sistema. Por otro lado deshabilitar las interrupciones es una operación costosa que requiere muchos ciclos de CPU, introduciendo una penalización al desempeño del sistema.

Los enfoques modernos para el tratamiento de las interrupciones dentro de los sistemas operativos, tratando de minimizar la latencia de interrupciones, se pueden dividir en dos grandes grupos: manejar las interrupciones dentro de hilos de interrupción, o dividir el manejo de interrupciones en dos secciones de prioridad diferente.

2.4.1 Hilos de interrupción

La idea general de este esquema de manejo de interrupciones consiste en activar una tarea (hilo o proceso) ante la ocurrencia de una interrupción, tarea que realiza todo el trabajo específico para atender a la interrupción ocurrida. El núcleo del sistema contiene un manejador de interrupción de bajo nivel (y generalmente de más prioridad que las tareas del sistema) que realiza lo necesario para activar una tarea

de interrupción asociada, y posiblemente realice otras actividades comunes al manejo de todas las interrupciones. Según la acción realizada por el manejador de bajo nivel se pueden identificar dos variantes fundamentales: señales de interrupción como eventos de comunicación entre procesos, y las interrupciones como hilos de núcleo.

En la primera variante el núcleo del sistema se encarga de convertir un evento de interrupción en un evento de sincronización entre procesos, como una señal sobre un semáforo, o el envío de un mensaje. Esta técnica ha venido utilizándose desde la primera generación de los sistemas operativos con arquitectura de micronúcleo, en la que resaltan sistemas como Mach [Accetta *et al.* 1986] y Chorus [Rozier *et al.* 1991]. Otro sistema de micronúcleo que utiliza esta variante es MINIX [Tanenbaum y Woodhull 2006], muy utilizado en la enseñanza de sistemas operativos.

Una alternativa al uso de eventos de sincronización entre procesos para activar tareas de interrupción fue introducida por Sun Microsystems en el núcleo de su sistema operativo Solaris 2.x (SunOS 5.0) [Eykholt *et al.* 1992] [Mauro y McDougall 2000], y consiste en activar directamente hilos ligeros de núcleo [Kleiman y Eykholt 1995] ante la ocurrencia de una interrupción. Posteriormente se han realizado varios trabajos orientados a introducir el manejo de interrupciones de Linux al contexto de hilos de núcleo, con el propósito de reducir su latencia de interrupción.

Los hilos de interrupción pueden utilizar los mecanismos de sincronización entre procesos suministrados por el núcleo. De esta forma el problema de la sincronización de las interrupciones se resuelve sin necesidad de deshabilitar las interrupciones en el manejador. Sólo el manejador de bajo nivel del núcleo puede necesitar deshabilitar las interrupciones, pero esto sucede en secciones de tamaño muy pequeño y conocido, con lo que se logra una reducción en la latencia de interrupciones, y una mejor predictibilidad del sistema.

2.4.2 Manejo de interrupciones en dos niveles

El enfoque de manejo de interrupciones en dos niveles consiste en dividir la atención a las interrupciones en dos secciones. La primera sección es una rutina convencional de atención a interrupción (ISR), que se ejecuta con un nivel de privilegio de interrupción alto y por tanto debe ser lo más corta posible para no afectar la latencia de interrupciones. Esta primera sección es responsable de realizar las acciones más urgentes del tratamiento de la interrupción, y diferir las acciones menos urgentes y computacionalmente más costosas a una segunda sección que se ejecuta a menor prioridad, y puede ser interrumpida por nuevas IRQs. Esta solución ha sido adoptada por muchos sistemas operativos de red como VMS 1.0+ [Kenah y Bate 1984], BSD 4.2+ [McKusick *et al.* 1996], Windows NT [Solomon 1998] y Linux [Beck *et al.* 1998]. Aunque la idea general y los beneficios obtenidos son similares, cada sistema difiere del resto en la implementación de la misma, en especial en la implementación de la segunda sección de baja prioridad (AST en VMS, interrupción de software en BSD 4.2+, DPC en NT y mitad inferior –*bottom half*– en Linux).

El manejo de interrupciones en dos niveles logra reducir la latencia de interrupciones del sistema al reducir el tamaño de la ISR (sección de alta prioridad). Sin embargo, en todas las implementaciones actuales de esta técnica la sección diferida tiene más prioridad que todas las tareas de usuario, incluyendo las de tiempo real. Las tareas críticas en tiempo se ven demoradas por las secciones de manejo de interrupción diferidas, por lo que para estas tareas hay poco beneficio derivado del uso de

esta implementación del manejo de interrupciones. Otro problema de esta técnica es el denominado "tiempo robado" (*stolen time*) a las aplicaciones por el procesamiento de interrupciones, ya que el tiempo utilizado por las secciones de manejo de interrupción (principalmente las secciones diferidas) está fuera del control del planificador del sistema. Se han realizado algunas propuestas para reducir estos problemas, por ejemplo [Jung *et al.* 2004] y [Zhang y West 2006].

2.4.3 Reducción del costo de enmascaramiento de interrupciones

Las operaciones que desactivan interrupciones de hardware, como pueden ser fijar la máscara o el nivel de interrupción dentro de un controlador de interrupciones, o modificar un bit para desactivar globalmente todas las interrupciones, son operaciones muy costosas, llegando a tomar en algunos casos cientos de ciclos de CPU [Ousterhout 1990]. Se han realizado algunos trabajos que intentan reducir la sobrecarga introducida por el hardware de interrupción.

En [Stodolsky *et al.* 1993] se propone una técnica que denomina "protección de interrupciones optimista", mediante la cual se fija una máscara de interrupción de software en lugar de fijarla físicamente para establecer el nivel de interrupción del sistema. Si ocurre una interrupción mientras está "enmascarada" por la máscara de software, se coloca la máscara física correspondiente al nivel de interrupción actual, y el manejo de la interrupción ocurrida (de menor prioridad) es diferido hasta que el nivel de interrupción del sistema se reduzca lo suficiente. El beneficio de esta técnica se basa en el hecho de que, en el caso más común, no ocurrirán interrupciones dentro de secciones críticas. De esta forma la mayor parte de las veces sólo se fijará la máscara de software, pero no la de hardware.

De forma similar, en [Shi *et al.* 2006] se propone utilizar una bandera (cierre de sincronización de interrupciones o **`ISLock`**) para las secciones que necesitan protección contra las interrupciones. Las interrupciones que ocurren mientras este cierre está activo son recordadas, y su manejador asociado no se ejecuta hasta que se libera el cierre. De esta forma elimina la necesidad de desactivar las interrupciones físicas en el sistema. Una desventaja de este método es que asume que las interrupciones no son recurrentes (es decir que mientras no se atienda una interrupción de un tipo no pueden ocurrir más interrupciones del mismo tipo), por lo que sólo es aplicable a ciertos tipos de dispositivos.

2.5 PARTEMOS

PARTEMOS es un micronúcleo de un sistema Operativo de Tiempo real, desarrollado con fines científicos y experimentales. Este micronúcleo fue diseñado en 2003 por Luis Eduardo Leyva del Foyo y desarrollado de forma independiente por Luis Eduardo Leyva del Foyo con la colaboración de William Martínez Cortés, Pablo R. López Martínez, Alain Tamayo Fong, Germán Mendoza Silva, Yunior Pupo Peña, y Luis E. Rodríguez Pupo.

Entre las principales características de PARTEMOS podemos mencionar:

- Un número mínimo de abstracciones que al mismo tiempo brindan una gran flexibilidad. Esta característica facilita la asimilación y programación del sistema.
- Planificador por prioridades con expropiación que soporta un número suficientemente amplio de niveles de prioridad.

- Un mecanismo flexible de administración de interrupciones independiente de la plataforma y completamente integrado con la administración y planificación de tareas.
- Soporte de mecanismos de sincronización entre procesos con tiempos de bloqueo deterministas.
- Un núcleo completamente expropiable, lo cual garantiza un tiempo de respuesta ágil a las tareas de usuario críticas en tiempo. Esto debido a que una tarea de usuario pudiera tener mayor prioridad que una tarea del núcleo.
- Mecanismo estructurado de manejo de excepciones que posibilita la construcción de software robusto y fiable.

La versión de PARTEMOS disponible al comienzo del presente trabajo (PARTEMOS16) se compila utilizando versiones MS-DOS de las herramientas de desarrollo de Borland, entre las que destacan el compilador Borland C++, y el ensamblador Turbo Assembler. PARTEMOS16 también utiliza MS-DOS como cargador, y el archivo del núcleo generado tiene un formato ".exe".

2.5.1 Arquitectura de PARTEMOS

Como se ha mencionado, PARTEMOS presenta una **arquitectura de micronúcleo**. Este es un método de estructuración de los sistemas operativos que elimina todos los componentes no esenciales del núcleo y los implanta como procesos a nivel de sistema y nivel de usuario, obteniendo como resultado un núcleo más pequeño. Existe poco consenso referente a qué servicios deben permanecer en el núcleo y cuáles deben implantarse en el espacio de usuario. Por regla general, los micronúcleos suministran administración de procesos y de memoria mínima, además de facilidades de comunicación y sincronización entre procesos. En lo que resta de este documento **utilizaremos el término núcleo y micronúcleo indistintamente** cuando nos refiramos al micronúcleo PARTEMOS.

El beneficio principal de la arquitectura de micronúcleo es la mayor facilidad de extender el sistema operativo. Esto es posible debido a que todos los nuevos servicios se añaden en el espacio de usuario y en consecuencia no requieren modificación del núcleo. Cuando por alguna razón se hace necesario modificar el núcleo, estos cambios suelen ser menores, precisamente por su pequeño tamaño. Con la arquitectura de micronúcleo el sistema operativo resultante es más fácil de transportar de una arquitectura de hardware a otra. Este modelo también suministra mayor seguridad y fiabilidad, dado que la mayoría de los servicios se ejecutan como procesos de usuario, en lugar de ejecutarse en el núcleo. Ante el fallo de un servicio, el resto del sistema operativo permanece ileso.

El micronúcleo PARTEMOS (ver figura 3) está dividido internamente en varias capas, siguiendo un modelo estratificado [Dijkstra 1968]. La **capa de abstracción de actividad** (AAL, *Activity Abstraction Layer*) implanta los mecanismos básicos necesarios para la ejecución de actividades concurrentes y brinda un mecanismo primitivo de sincronización y comunicación entre éstas. La **capa de abstracción de tareas** (TAL, *Task Abstraction Layer*) está encargada de implementar la abstracción de tarea, incluyendo políticas de planificación, y mecanismos de comunicación y sincronización [López-Martínez 2003]. Por encima de todas las capas se encuentra la capa de interfaz de aplicación **API**, la cual define la forma estándar en que las tareas deben acceder a los servicios del núcleo. En la versión actual de PARTEMOS el código de las aplicaciones y del núcleo se enlazan juntos en un único módulo ejecutable, por lo que la capa API es muy sencilla y está formada por un único archivo de encabezado ".h" ("`kernel.h`") que

lista los servicios accesibles desde las aplicaciones. Futuras versiones de la capa API pueden ser más complejas, con características como acceso a los servicios mediante trampas del procesador, y soporte de cadena de caracteres para nombrar abstracciones.

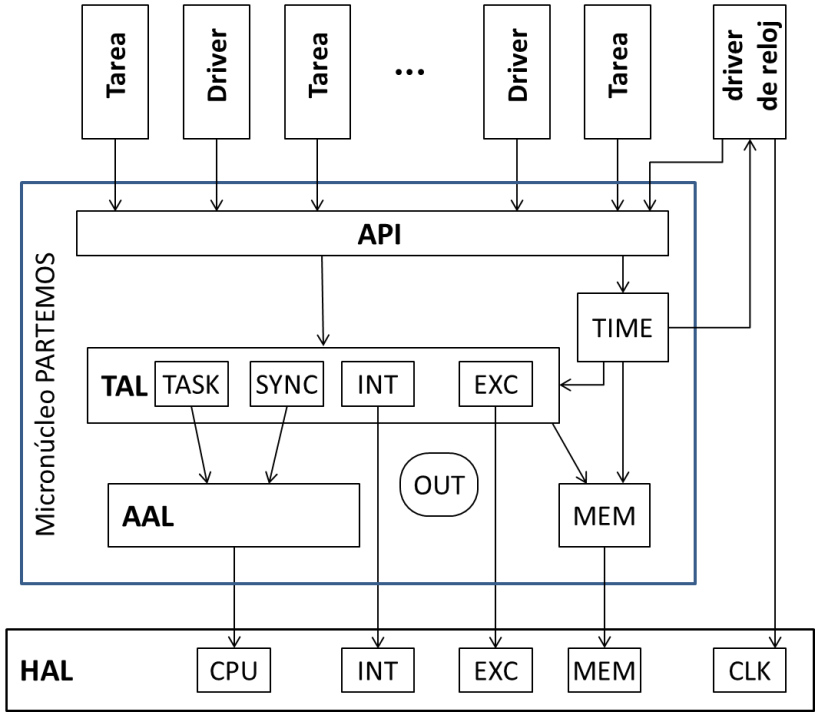


Figura 3: Esquema simplificado del micronúcleo PARTEMOS.

Todo el micronúcleo está soportado por la **Capa de abstracción de hardware** (HAL, *Hardware Abstraction Layer*), encargada de crear una interfaz que oculta las especificidades del hardware subyacente al resto del sistema. Esta capa es esencial para proporcionar portabilidad al núcleo PARTEMOS, ya que teóricamente se puede compilar el código del núcleo sin cambios en cualquier plataforma para la cual se haya creado una capa HAL. Sin embargo durante la ejecución del presente trabajo de tesis se encontraron muchos detalles en la implementación de PARTEMOS16 que afectaban la portabilidad del núcleo.

Cada capa está dividida en módulos, lo que facilita la extensión, modificación y configuración de sus abstracciones fundamentales. Entre estos módulos se destacan los módulos de la capa TAL: **KRNLTASK**, **KRNLSYNC**, **KRNLINT**, y **KRNLEXC**. Los módulos KRNLTASK y KRNLSYNC son los encargados de implementar la abstracción de tarea y objetos de sincronización, respectivamente; para lo cual utilizan servicios brindados por la capa AAL. Mientras que los módulos KRNLINT y KRNLEXC se encargan de convertir las interrupciones y excepciones de CPU (respectivamente) en abstracciones de mayor nivel dentro del núcleo.

Algunos módulos del núcleo se encuentran aislados, sin formar parte de ninguna capa. Entre estos uno de los más complejos es el módulo KRNLMEM, formado por varios archivos que implementan el administrador de memoria del núcleo. El módulo KRNLTIME utiliza los servicios de la capa TAL y del driver de reloj para brindar funcionalidades de alto nivel relacionadas con el tiempo, como tareas

periódicas o esperas temporizadas en objetos de sincronización. El módulo KRNLOUT brinda funcionalidades de salida de texto, que pueden ser utilizadas por cualquier módulo del núcleo.

En la figura 3 se puede apreciar como los drivers de dispositivos no forman parte del micronúcleo. Estos drivers son ejecutados por tareas, sin hacerse distinción entre ellos y otras tareas de usuario. Solo el driver de reloj tiene una característica que lo distingue del resto de las tareas, y es que proporciona servicios de tiempo específicos al núcleo (módulo KRNLTIME). Por lo demás el driver de reloj es una tarea como cualquier otra.

2.5.2 Manejo de interrupciones en PARTEMOS

El tratamiento de interrupciones en el micronúcleo PARTEMOS es radicalmente diferente del manejo de interrupciones en los sistemas multitarea convencionales. En la mayoría de los sistemas revisados anteriormente los manejadores de interrupción tienen más prioridad que el resto de las tareas, lo cual no siempre se ajusta a los requerimientos de los sistemas de tiempo real, donde pueden existir tareas de usuario con requerimientos de tiempo de respuesta crítico. Incluso en los casos cuando las tareas de interrupción pueden tener menor prioridad que otras tareas de usuario críticas, siempre existe una sección de código (aunque pequeña) que puede interrumpir el resto del sistema. Esto puede provocar afectaciones en los tiempos de ejecución de las tareas debido a sobrecargas temporales de interrupciones en el sistema.

Una característica distintiva de PARTEMOS es que las interrupciones y tareas comparten un único espacio de prioridades (**espacio de prioridades unificado**, introducido en [Leyva-del-Foyo y Mejia-Alvarez 2004]), pudiendo una tarea tener más prioridad que una interrupción. Tanto las tareas como las interrupciones tienen una prioridad cuyo valor se encuentra entre 0 y 255, siendo 255 el nivel más prioritario. Una IRQ (petición de interrupción) nunca interrumpirá a la CPU mientras la tarea actualmente en ejecución tenga una prioridad mayor o igual que la suya. Esto garantiza que las tareas de alta prioridad nunca serán interferidas por interrupciones de baja prioridad, eliminándose la variabilidad en los tiempos de ejecución provocada por dichas interrupciones en los sistemas convencionales. En especial se eliminan los efectos dañinos de una sobrecarga temporal de interrupciones.

Las interrupciones en PARTEMOS deben manejarse completamente dentro del código de las tareas, ya que este micronúcleo no soporta la instalación de ISRs de usuario, pero brinda los mecanismos necesarios para sincronizar las interrupciones con las tareas. A estas tareas que ejecutan un código en respuesta a una interrupción las denominaremos **tareas de servicio de interrupción** (IST - *Interrupt Service Task*), las cuales generalmente tienen la misma prioridad que la interrupción a la que dan tratamiento. Como cualquier otra tarea, las IST pueden acceder a todos los servicios brindados por el núcleo, lo que constituye una ventaja sobre otros esquemas de manejo de interrupción.

La figura 4 muestra el espacio de prioridades de PARTEMOS. Se puede apreciar como una IST, y por tanto su interrupción asociada, pueden tener cualquier prioridad dentro del espacio de prioridades del sistema.

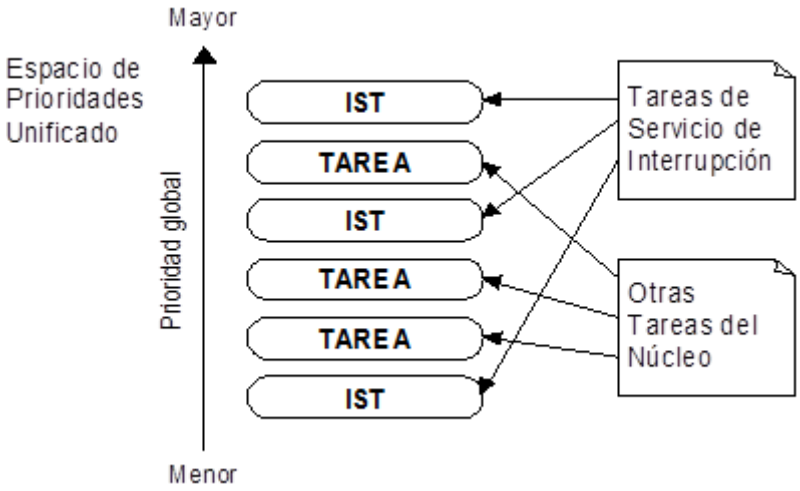


Figura 4: Espacio de prioridades unificado.

Al igual que el resto de los sistemas que manejan las interrupciones como hilos, las tareas de PARTEMOS (sin distinguir entre las tareas de interrupción y las restantes) pueden utilizar los mecanismos de sincronización brindados por el sistema. Esto elimina la necesidad de deshabilitar las interrupciones en el código de las tareas, garantizándose que la latencia de las interrupciones en el peor caso sea conocida y acotada. Como todas las actividades asíncronas del sistema son tareas, la cantidad de mecanismos disponibles para la sincronización entre éstas es reducida, lo que le da simplicidad a la implementación de sistemas de tiempo real sobre PARTEMOS.

Para sincronizar la ejecución de una IST con la ocurrencia de una interrupción, PARTEMOS brinda unos semáforos especiales, que pueden asociarse a las IRQs. Cuando ocurre una interrupción, el núcleo la convierte en una señal sobre un semáforo; si una tarea (IST) se bloquea tras realizar una operación de espera (*wait*) en dicho semáforo, será desbloqueada una vez ocurrida la interrupción. Este sencillo método permite que se oculte la asincronía propia de las interrupciones en las capas más bajas del sistema.

Previo a este trabajo, se han realizado varios experimentos que demuestran el determinismo de este modelo de manejo de interrupciones. Uno de estos experimentos consistió en someter a PARTEMOS a una sobrecarga de interrupciones de baja prioridad, y observar cómo se veía afectado el desempeño de una tarea crítica (de alta prioridad) [Martínez-Cortés 2005]. La figura 5 muestra los resultados medidos en esta prueba, con un conjunto de tres tareas de prueba: A, B y C. La tarea A es la tarea de alta prioridad, B es una tarea asociada a una interrupción de menor prioridad, mientras que C es la tarea de más baja prioridad en el sistema. Denotamos con T_x al período de ejecución de la tarea x, y con C_x a su tiempo de cómputo. La línea discontinua indica el instante en que inicia la ocurrencia de la interrupción asociada a la tarea B, con una frecuencia más alta de lo que el sistema puede manejar (situación de stress). Puede observarse como la tarea de baja prioridad C se deja de ejecutar, mientras la tarea de alta prioridad A no sufre afectación significativa en sus parámetros temporales. Los resultados mostrados en la figura fueron medidos en una computadora personal con procesador 80486. Las ejecuciones de la prueba en otras computadoras y con otros métodos de medición arrojaron resultados similares.

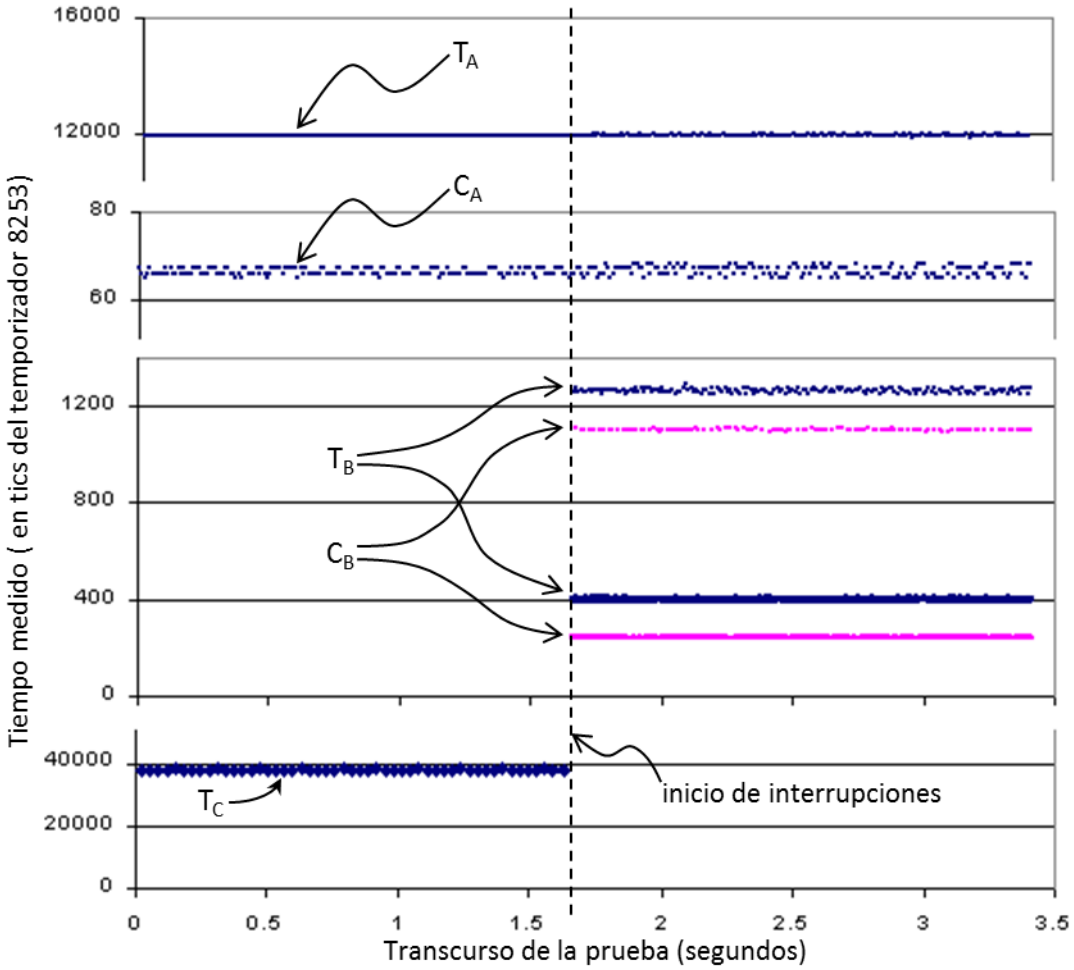


Figura 5: Prueba de determinismo.

El responsable del tratamiento de las interrupciones a más bajo nivel es el módulo de abstracción del hardware de interrupciones, que forma parte de la capa HAL. Este módulo, también denominado **HAL de interrupciones** (ó INTHAL), se ocupa de los aspectos dependientes del hardware de interrupción de la máquina. El INTHAL juega un papel crítico en la implantación del modelo de prioridades unificado de PARTEMOS, ya que adapta el modelo de prioridades brindado por el hardware de interrupción, al modelo de prioridades de PARTEMOS.

Actualmente, el HAL de interrupciones de PARTEMOS está implementado sobre el controlador de interrupciones 8259, y se apoya fundamentalmente en el uso de máscaras de interrupción para lograr el modelo de prioridades de PARTEMOS[1]. En PARTEMOS es posible activar el llamado "enmascaramiento de interrupciones virtual", que utiliza una técnica similar a las descritas en la sección 2.4.3 para reducir la sobrecarga introducida por el uso del hardware de interrupciones [Leyva-del-Foyo *et al.* 2006b]. En PARTEMOS también se puede activar el denominado "modo EOI automático", que elimina la necesidad de enviar comandos de fin de interrupción (EOI) a los controladores de interrupción, con la consiguiente reducción de la sobrecarga de manejo de las interrupciones [Leyva-del-Foyo *et al.* 2006b].

[1] En este trabajo se hace una implementación sobre el controlador de interrupciones sobre APIC, ver sección 5.3.4.

Capítulo 3
Panorámica de la arquitectura Intel de 32 bits relevante a PARTEMOS

La arquitectura Intel de 32 bits está presente en todas las computadoras personales (PCs) modernas, motivo por el cual fue elegida para crear la primera versión de 32 bits de PARTEMOS. En este capítulo se describen las principales características de esta arquitectura, haciendo énfasis en aquellas características utilizadas en la implementación de PARTEMOS.

A lo largo del capítulo se usará el término núcleo para referirse a los **núcleos de CPU**, no al núcleo de los sistemas operativos. Un núcleo de CPU es una unidad de procesamiento que puede estar integrada en el mismo chip con otros elementos, incluyendo otros núcleos de CPU. También se utilizará la terminación "H" y "B" para indicar números en hexadecimal y binario respectivamente, a diferencia de los restantes capítulos que usan la nomenclatura de C (prefijo 0x) para los números en hexadecimal.

3.1 Arquitectura de la CPU

La arquitectura Intel de 32 bits, denotada abreviadamente como IA-32, fue introducida por primera vez por esta compañía en 1985, con el lanzamiento de su procesador 80386. Procesadores posteriores como el 80486, todos los Pentiums, y las familias de procesadores "Core" han mantenido esta arquitectura, aunque las últimas generaciones incluyen además soporte de 64 bits. Para abreviar, llamaremos procesadores IA-32 a todos los procesadores compatibles con dicha arquitectura. En [Intel 2012] puede encontrarse información detallada de los procesadores incluidos en esta arquitectura.

Los procesadores con arquitectura IA-32 tienen un modo de trabajo denominado **modo real**. En modo real el procesador se comporta de forma similar a sus antecesores de 16 bits: los procesadores 8086 y 8088 (con algunas extensiones), por lo que puede ejecutar aplicaciones antiguas diseñadas para este procesador. PARTEMOS16 estaba diseñado para ejecutarse en modo real, y la nueva versión (PARTEMOS32) también debe poder compilar para ejecutarse en este modo. El procesador es colocado en modo real cada vez que se reinicia la máquina.

Aunque el modo real existe por compatibilidad, es el denominado **modo protegido** de 32 bits el que permite expresar toda la potencia del procesador, y puede considerarse el estado nativo del mismo. Este modo brinda ventajas como acceso lineal a toda la memoria física (hasta 4 GB), paginación, protección de memoria y otras características deseables en sistemas multitarea. En modo protegido, la ejecución de código se puede realizar con diferentes **niveles de privilegio**, lo que permite restringir la realización de ciertas operaciones al código de sistema, que se ejecuta en **modo privilegiado** (**modo supervisor**). A menos que se especifique lo contrario, todas las características mostradas en este capítulo asumen que el procesador se encuentra en modo protegido de 32 bits.

Existen extensiones a la arquitectura IA-32 que permiten al procesador acceder a más de 4 GB, las cuales no serán abordadas en este documento. Tampoco se brindará información sobre las arquitecturas de 64 bits soportadas por los procesadores más modernos.

3.1.1 Registros básicos de ejecución

La arquitectura IA-32 proporciona 16 registros básicos de ejecución de programas, disponibles para uso general en la programación de sistemas y aplicaciones (ver figura 6). Estos registros pueden ser agrupados como: registros de propósito general, registros de segmento, registro EFLAGS (control y estado de programa), y registro EIP (apuntador de instrucciones).

Los 8 registros de propósito general EAX, EBX, ECX, EDX, EBP, ESP, ESP, ESI, y EDI ejercen algunas de las siguientes funciones:

- Operandos para operaciones aritméticas y lógicas
- Operandos para cálculos de direcciones
- Apuntadores a memoria

Los registros de propósito general tienen un tamaño de 32 bits, cuyos 16 bits menos significativos coinciden con los registros de propósito general del 8086, como se muestra en la figura 6. Por ejemplo, la mitad inferior de EAX puede accederse como un registro de 16 bits y su valor coincide con el del registro AX de la arquitectura de 16 bits predecesora. También es posible acceder a los subregistros de 8 bits definidos en el procesador 8086 (AH, AL, BH, BL, CH, CL, DH, y CL).

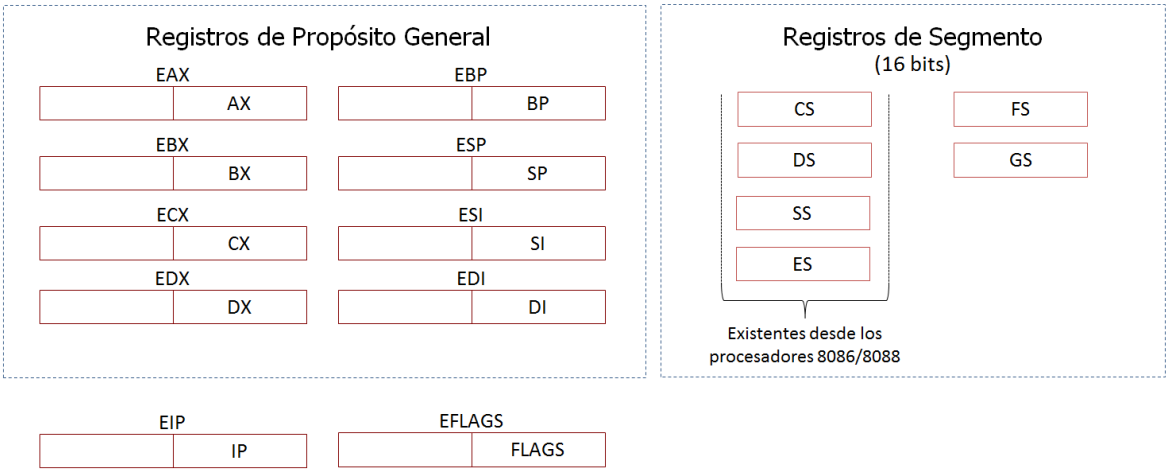


Figura 6: Registros básicos de ejecución de la arquitectura IA-32.

Para una descripción detallada de los diferentes usos que tienen los registros de propósito general, consultar el capítulo 5 ("*Instruction Set Summary*"), volumen 1 de [Intel 2012].

Los registros de segmento (CS, DS, SS, ES, FS, y GS) almacenan valores de 16 bits denominados selectores de segmento. Un selector de segmento es un apuntador especial que identifica un segmento en memoria. Cada registro de segmento está asociado con uno de tres tipos de almacenamiento: código, datos, o pila. El registro CS contiene el selector para el segmento de código, SS contiene el selector para el segmento de pila, y los restantes registros de segmento contienen selectores para segmentos de datos.

El registro apuntador de instrucciones (EIP) contiene el desplazamiento (de 32 bits) de la próxima instrucción a ser ejecutada dentro del segmento de código actual.

El registro de 32 bits EFLAGS contiene un grupo de banderas de estado, una bandera de control, y un grupo de banderas de sistema, marcadas con las letras S, C y X en la figura 7. Una bandera que guarda especial interés es el bit 9 (IF). Cuando IF está en cero, el núcleo del procesador ignora todas las peticiones de interrupción que le llegan, en este caso se dice que las interrupciones están **deshabilitadas a nivel de CPU**.

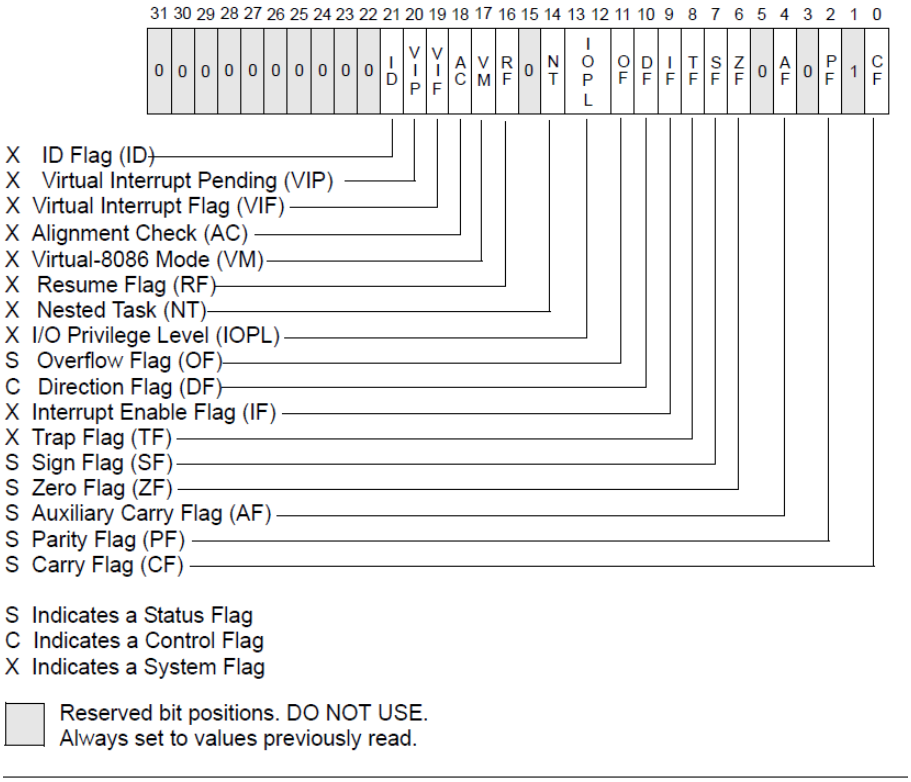


Figura 7: Registro EFlags (tomado de [Intel 2012]).

Todos los accesos a memoria son referenciados mediante un selector de segmento y un desplazamiento, que en conjunto conforman lo que se denomina **dirección lógica ó apuntador FAR**. Muchas veces el selector de segmento se elige implícitamente de uno de los registros de segmento, dependiendo del tipo de operación realizada por el procesador. El desplazamiento puede indicarse de muchas formas, en los casos mas complejos se pueden calcular a partir de registros base, registros índices, valores inmediatos y un factor de escala, como se muestra en la figura 8. El desplazamiento se suma a la dirección base del segmento accedido para formar la **dirección lineal** de acceso a memoria.

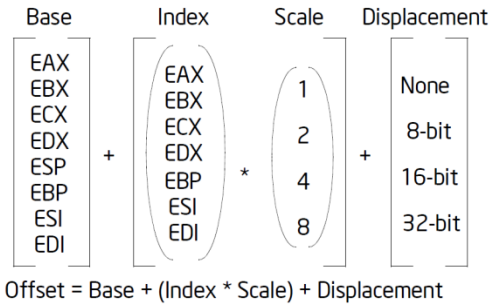


Figura 8 Cálculo de desplazamiento durante el acceso a memoria (tomado de [Intel 2012]).

3.1.2 Modelo de administración de memoria de IA-32

Las facilidades de administración de memoria de la arquitectura IA-32 se dividen en dos partes: segmentación y paginación (ver figura 9). La segmentación proporciona un mecanismo para aislar secciones individuales de código, datos y pila de forma que muchos programas puedan ejecutarse en el mismo procesador sin interferirse mutuamente. La paginación brinda un mecanismo para implementar un sistema típico de memoria virtual con paginación por demanda, y también puede usarse para proporcionar aislamiento entre tareas. El mecanismo de segmentación de los procesadores IA-32 no puede ser desactivado, por otro lado, el uso de la paginación es opcional. En la figura 9 se muestra como una dirección de memoria lógica es transformada mediante el mecanismo de paginación en una **dirección lineal** (32 bits), y esta última es opcionalmente traducida en una **dirección física** (también de 32 bits) por el mecanismo de paginación.

Si el mecanismo de paginación esta desactivado, la dirección lineal coincide con la dirección física usada para acceder a memoria. En este capítulo no entraremos en detalle del mecanismo de paginación, por no ser relevante a la arquitectura de PARTEMOS.

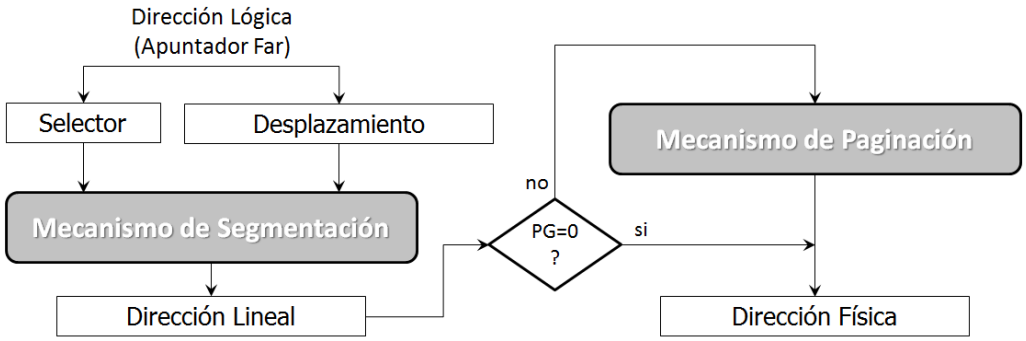


Figura 9 Flujo general de conversión de una dirección lógica en física durante el acceso a memoria.

La figura 10 muestra los elementos fundamentales que intervienen en el funcionamiento del mecanismo de segmentación. Los datos que caracterizan a cada segmento de memoria disponible en el sistema están contenidos en una estructura denominada **descriptor de segmento.** Todos los descriptores de segmento utilizables en un momento dado se encuentran en unas tablas especiales del sistema, denominadas **tablas de descriptores,** de la cual la principal es la **tabla de descriptores globales** (**GDT**).

Como se puede apreciar en la figura 10, un selector de segmento contiene (entre otras cosas) un índice en una de las tablas de descriptores del sistema.

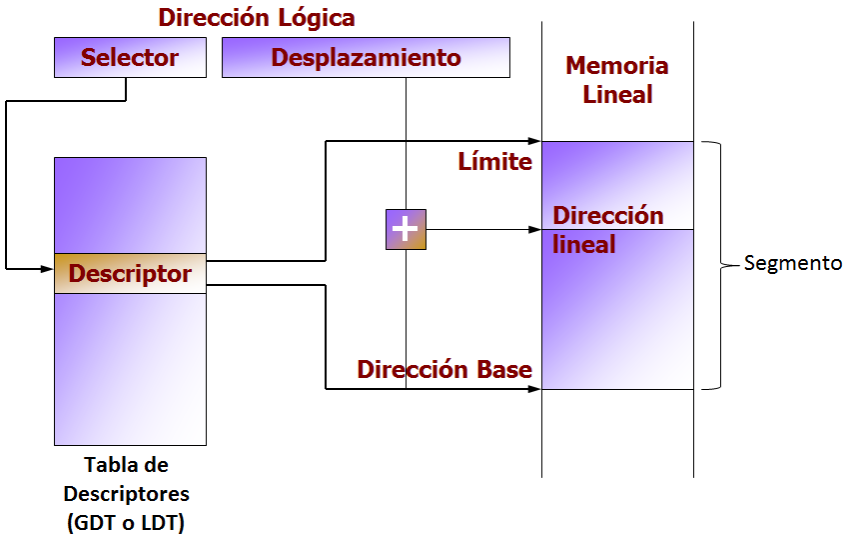


Figura 10: Mecanismo de segmentación.

La tabla de descriptores globales (GDT) es una estructura en memoria referenciada por un registro especial de 48 bits, GDTR, que contiene su dirección base y limite. Esta compuesta por elementos denominados descriptores, que pueden ser **descriptores de segmento o del sistema**. El primer descriptor de la GDT (descriptor cero) no puede utilizarse, ya que su índice está reservado por el procesador para apuntadores nulos.

Similar a la GDT en estructura y funcionalidad podemos encontrar a la **tabla de descriptores locales** (**LDT**). Las tablas LDT son definidas por descriptores de segmentos del sistema (en la GDT), y aunque pueden existir muchas tablas LDT, sólo una puede estar activa en un momento dado. El uso de tablas LDT es opcional y generalmente se utilizan para ocultar segmentos entre diferentes tareas.

Cada descriptor de segmento es una estructura de 64 bits, cuyo formato (para descriptores de segmentos de código y datos) se muestra en la figura 11. Además de los segmentos de código y datos, existen **descriptores de segmentos del sistema**, sólo utilizables por código prioritario. Los descriptores de segmento del sistema pueden referenciar a una tabla de descriptores locales, o a un tipo especial de segmento denominado **segmento de estado de tarea**.

También existen descriptores especiales del sistema que no definen segmentos en memoria, sino que se utilizan como vía para realizar cambios en el flujo de ejecución de la CPU, y por eso se les denomina **descriptores de compuerta**. Los descriptores de compuerta se pueden clasificar en cuatro grupos fundamentales: compuertas de tarea, compuertas de llamada, compuertas de interrupción, y compuertas de trampa.

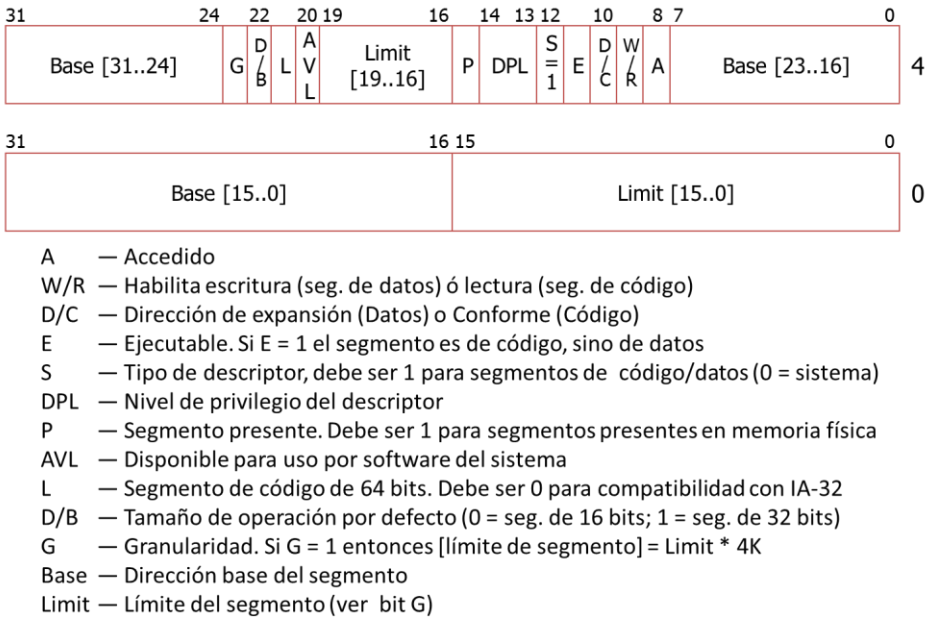


Figura 11: Formato de un descriptor de segmento.

Las compuertas de tarea, junto con los segmentos de estado de tarea (TSS), son elementos especiales brindados por la arquitectura IA-32 para permitir conmutación de tarea por hardware. La conmutación de tarea por hardware generalmente es una operación costosa, y muchos sistemas multitareas prefieren usar una alternativa de conmutación por software. No entraremos en detalle de esta funcionalidad, debido a que no es utilizada en PARTEMOS. Los restantes descriptores de compuerta se describen brevemente en la próxima sección.

La figura 12 muestra la estructura de un selector de segmento. Cada vez que se asigna un valor de un selector de segmento a un registro de segmento, los campos “Índice” y TI son utilizados para localizar un descriptor (de segmento) en una de las tablas GDT o LDT. El campo RPL es usado para brindar protección de segmento.

Figura 12: Selector de segmento.

Cada registro de segmento tiene asociado una parte “oculta” que llamaremos **registro descriptor de segmento**. Cuando se carga un valor en un registro segmento, el registro descriptor de segmento asociado también se carga con la dirección base, límite e información de control de acceso, obtenidos del descriptor de segmento referenciado. Los registros descriptores de segmento sirven como una cache

de los descriptores reales en memoria, y permiten que el procesador traduzca direcciones sin requerir accesos adicionales a memoria.

Como hemos mencionado previamente, el mecanismo de segmentación no puede ser desactivado, sin embargo, es muy común que en los sistemas operativos modernos se utilice un modelo de memoria no segmentado (**modelo de memoria plano**), donde cada proceso tiene acceso a un espacio de memoria que se extiende desde la dirección 0 hasta la máxima posible[1] (FFFF FFFFH en 32 bits), y en este espacio único de memoria se ubican tanto código como datos (incluyendo memoria de pila). Para lograr un modelo de memoria plano en arquitectura IA-32, todos los registros de segmento deben cargarse con selectores para segmentos con base 0 y dirección máxima FFFF FFFFH (ver figura 13). Típicamente se definen dos segmentos superpuestos, uno para código y otro para datos y pila. El registro CS se hace apuntar al segmento de código y los restantes registros al segmento de datos y pila.

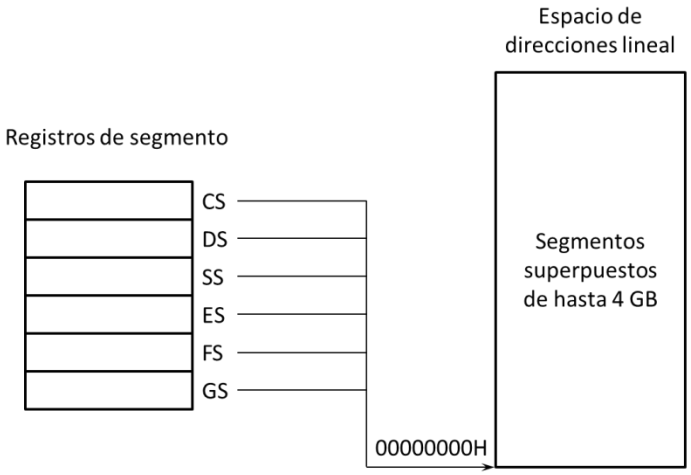


Figura 13: Modelo de memoria plano.

3.1.3 Modelo de administración de interrupciones de IA-32

Las interrupciones y excepciones son eventos que indican la existencia de una condición en el sistema que requiere atención del procesador. Estos eventos típicamente provocan una transferencia de control desde el código actualmente en ejecución hacia una rutina especial denominada manejador de interrupción o de excepción.

A diferencia de las interrupciones, las excepciones ocurren cuando el procesador detecta una condición de error en la instrucción actualmente en ejecución, como un fallo de página o una división por cero. El procesador utiliza el mismo mecanismo de interrupciones como forma de notificación de una excepción. Las excepciones pueden considerarse interrupciones síncronas, ya que son causadas por la ejecución de una instrucción en lugar de ser activadas como respuesta a un evento externo.

Esta sección describe el mecanismo de manejo de interrupciones y excepciones presente en los procesadores IA-32. Más adelante, en la sección 3.2 se brinda información del hardware involucrado en

[1] Para considerarse un modelo plano el espacio de memoria no necesita llegar a la máxima dirección, pero es muy común que se use esta configuración, sobre todo en sistemas con memoria virtual paginada.

el control de las interrupciones. A menos que se especifique explícitamente, todo lo descrito en esta sección para las interrupciones de hardware también aplica para las excepciones.

Cada fuente de interrupción y cada excepción definida por la arquitectura tiene asociado un número único entre 0 y 255, denominado **número de vector**. El procesador utiliza el número de vector como un índice dentro de una tabla del sistema denominada **Tabla de Descriptores de Interrupción** (**IDT**). Los números de vectores en el rango de 0 a 31 están reservados por la arquitectura IA-32 para excepciones e interrupciones especiales, mientras los restantes números (de 32 a 255) están disponibles para las interrupciones de hardware y para uso general por parte del software de sistema.

Es posible deshabilitar temporalmente la atención a las interrupciones de hardware por parte del procesador, para lo cual se debe limpiar el bit IF del registro EFLAGs (ver figura 7). Las únicas interrupciones de hardware que no se pueden deshabilitar son las denominadas **interrupciones no enmascarables** (NMI), que generalmente se notifican por un pin específico del procesador, y siempre se generan con número de vector 2. El bit IF no tiene efecto sobre las excepciones ni sobre interrupciones generadas por software con la instrucción INT.

Cada vez que se genera una interrupción, y antes de invocarse al manejador asociado, el procesador coloca en la pila ciertos valores necesarios para restaurar el estado de la CPU. Los valores específicos apilados dependen del tipo de interrupción y del código interrumpido. En el caso de que el código interrumpido tenga el mismo nivel de privilegio que el manejador de interrupción, se apilarán los registros EFlags, CS y EIP, en ese orden. Algunas excepciones provocan que la CPU coloque una palabra adicional en la pila (**código de error**) para brindar información sobre la excepción específica. Para retornar al código interrumpido, el manejador de interrupción debe ejecutar la instrucción IRET, que restaura todos los registros salvados en la pila. La instrucción IRET no extrae el código de error, por lo que es responsabilidad del manejador de excepción extraerlo antes de ejecutar IRET. También es responsabilidad del manejador salvar y restaurar cualquier otro registro que vaya a modificar.

La tabla de descriptores de interrupción (IDT) establece la asociación entre los vectores de interrupción y las rutinas que les dan servicio[1]. De forma similar a la GDT, la IDT es una tabla global del sistema, y está formada por descriptores de 64 bits. A diferencia de la GDT, la primera entrada de la IDT puede contener un descriptor, y sólo están permitidos descriptores de compuerta (de interrupción o de trampa). El procesador localiza la IDT usando el registro IDTR, que contiene la dirección base de 32 bits y un límite de 16 bits para esta tabla.

En la figura 14 se muestra el esquema de funcionamiento del mecanismo de interrupciones de la CPU. Cuando ocurre una interrupción, su número de vector se utiliza para localizar un descriptor de compuerta dentro de la IDT. Los descriptores de compuerta contienen un apuntador “far” (selector de segmento y desplazamiento) que el procesador usa como dirección de la rutina de atención a interrupción.

[1] También es posible asociar un vector de interrupción con una tarea de hardware utilizando compuertas de tarea, pero no daremos detalles de esta alternativa.

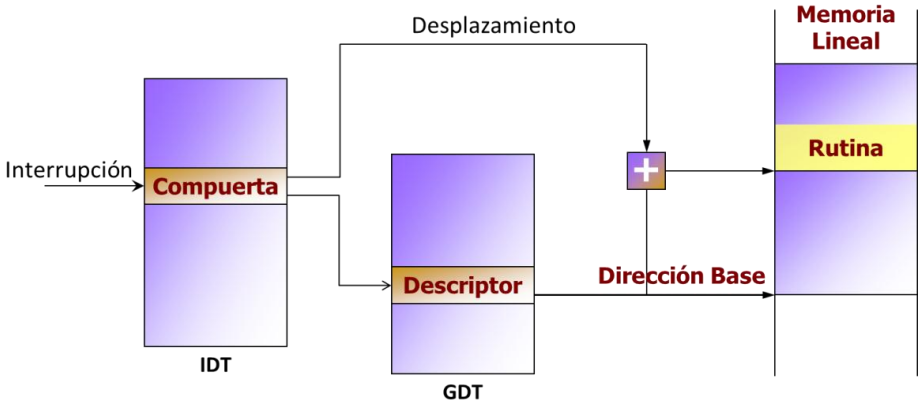
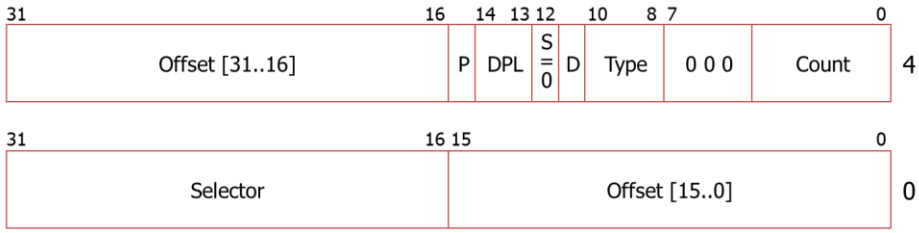


Figura 14: Mecanismo de manejo de interrupciones.

La figura 15 muestra la estructura de un descriptor para compuertas de llamada, interrupción o trampa. De ellos, sólo los descriptores de interrupción y trampa pueden ubicarse en la IDT. Puede apreciarse como estos descriptores contienen un selector para un segmento (de código), y un desplazamiento dentro de ese segmento como punto de entrada de la rutina de servicio.

Cuando ocurre una interrupción cuyo número de vector corresponde con una **compuerta de interrupción** en la IDT, además de salvar en la pila los registros mencionados previamente, el procesador limpia la bandera de interrupción (IF) para luego ceder el control a la rutina de atención a interrupción (ISR). La dirección lógica (selector y desplazamiento) de la ISR está contenida dentro del descriptor de la compuerta (ver figura 15). Al limpiar la bandera IF se logra que la rutina de atención se ejecute con interrupciones deshabilitadas (a menos que las habilite explícitamente), lo que impide que esta pueda ser nuevamente interrumpida.



Offset — Desplazamiento del punto de entrada a la rutina
Selector — Selector para el segmento de código destino
Count — Cantidad de parámetros de pila a copiar (solo compuertas de llamada)
Type — Tipo de compuerta: 100 = llamada; 110 = interrupción; 111 = trampa.
D — Tamaño de compuerta: 1 = 32 bits; 0 = 16 bits.
S — Tipo de descriptor, debe ser 0 para descriptores del sistema.
DPL — Nivel de privilegio del descriptor
P — Segmento presente

Figura 15: Descriptor de compuerta de llamada, interrupción y trampa.

Las compuertas de trampa funcionan de forma similar a las compuertas de interrupción, con la diferencia que no deshabilitan las interrupciones en el núcleo de CPU. Un uso común de estas compuertas es la creación de puntos de entrada a los servicios del sistema operativo, utilizando el mecanismo de interrupciones como medio para desacoplar el código de aplicación del código del sistema.

Los selectores de compuerta de llamada son muy similares a los otros selectores de compuerta, pero no se deben ubicar en la IDT, sino en la GDT o LDT. Las compuertas de llamada pueden ser útiles para brindar una forma de invocación de servicios del sistema operativo, como alternativa al uso del mecanismo de interrupciones con esta intención. A diferencia de las restantes compuertas mencionadas, las compuertas de llamada permiten la copia de parámetros entre pilas. No brindamos detalle de este tipo de compuerta ya que no es relevante a la arquitectura de PARTEMOS.

3.1.4 Protección

Los procesadores IA-32 brindan mecanismos de protección que funcionan a nivel de segmento y a nivel de página (si el mecanismo de paginación está activo). La protección es útil tanto en etapas de desarrollo de software (para detectar y localizar errores), como en productos finales para hacer mas robustos a los sistemas operativos, por ejemplo se puede impedir que una tarea modifique estructuras del sistema operativo u otra tarea.

Aunque no entraremos en detalle de las diferentes funcionalidades de protección, por no ser relevantes a la arquitectura de PARTEMOS, a manera informativa listamos las diferentes categorías en que podemos clasificar las verificaciones realizadas por el mecanismo de protección:

- **Verificación de límites**: impide que un acceso a un segmento de memoria esté localizado en una dirección fuera de los límites del segmento.
- **Verificación de tipo**: impide accesos a segmentos que no son permitidos según su tipo definido en el descriptor. Por ejemplo no se puede escribir en un segmento de código, ni ejecutar un segmento de datos.
- **Verificación de nivel de privilegio**: impide que código poco privilegiado acceda a segmentos con mayor nivel de privilegio.
- **Restricción de instrucciones**: impide que ciertas instrucciones (**instrucciones privilegiadas**) sean ejecutadas por código no privilegiado. También se restringen los efectos de algunas instrucciones, por ejemplo un código no privilegiado puede modificar el registro EFLAGS, pero no cambiar el valor de la bandera IF. Otros tipos de instrucciones que pueden restringirse son las operaciones de entrada/salida.
- **Protección a nivel de páginas**: El mecanismo de paginación identifica sólo dos niveles de privilegio: modo usuario y modo supervisor. En modo supervisor no hay limitación a las páginas accedidas, sin embargo en modo usuario es posible impedir el acceso o permitir acceso de solo lectura a ciertas páginas.

3.1.5 Soporte para depuración de la arquitectura IA-32

Los procesadores IA-32 brindan muchas facilidades de depuración y monitoreo de desempeño. Una de las facilidades más interesantes son las brindadas por los registros de depuración (DR0 a DR7), que describimos en esta sección debido a que son utilizados dentro del núcleo de PARTEMOS. Para descripción completa de todas las características de depuración y monitoreo de desempeño ver [Intel 2012], volumen 3, capítulo 17.

Los registros DR0-DR7 permiten especificar breakpoints de hardware, que son posiciones seleccionadas dentro de un programa, área de almacenamiento de datos en memoria o puertos de entrada/salida, en las que se desea detener la ejecución de un programa y examinar el estado del procesador. Cada vez que

se realiza un acceso a memoria o puerto de E/S cuya dirección está especificada en un breakpoint de hardware se genera una excepción de depuración (con número de vector 1).

Los registros de depuración pueden leerse y escribirse mediante variantes de la instrucción MOV, donde un registro de depuración puede ser el operando fuente o destino de la instrucción. Estos registros permiten configurar hasta cuatro breakpoints, numerados de 0 a 3. Para cada breakpoint se puede especificar (o leer) la siguiente información:

- La dirección lineal donde se desea que ocurra el breakpoint.
- El tamaño de la zona de breakpoint (1, 2, 4, u 8 bytes).
- La operación que debe realizarse en la dirección para que se genere la excepción de depuración.
- Si el breakpoint está o no habilitado.
- Si la condición del breakpoint estaba o no presente cuando se generó la excepción de depuración.

Los registros DR0 a DR3 contienen la dirección lineal de cada breakpoint. El registro DR7 (*debug control*) permite especificar propiedades de cada breakpoint. Cuando ocurre una excepción de depuración, se debe leer el registro DR6 (*debug status*) para identificar cual (o cuales) de los cuatro breakpoints generó la excepción. Los registros DR4 y DR5 están reservados y no deberían usarse.

En la figura 16 se puede apreciar la estructura del registro DR7. El cual contiene las siguientes banderas y campos:

- **Banderas L0 a L3 (habilitación de breakpoint local):** Cuando están en 1 habilitan la condición de ocurrencia para el breakpoint asociado. El procesador automáticamente limpia estos bits cuando ocurre una conmutación de tarea por hardware.
- **Banderas G0 a G3 (habilitación de breakpoint global):** Similares a las banderas Lx, pero no se limpian cuando ocurre conmutación de tarea.
- **Banderas LE y GE (habilitación de breakpoint exacto):** Esta característica no está soportada en los procesadores más recientes. Por compatibilidad se recomienda dejar estos bits en 1.
- **Bandera GD (habilitación de detección general):** Cuando está en 1 cualquier acceso a los registros de depuración causa que se genere una excepción de depuración.
- **Campos R/W0 a R/W3 (lectura/escritura):** Especifican la condición que provoca la activación del breakpoint correspondiente. Sus valores pueden ser:
  00 — se activa con la ejecución de una instrucción.
  01 — se activa con escritura a memoria.
  10 — se activa con el acceso a un puerto de E/S (sólo procesadores recientes).
  11 — se activa con lectura o escritura a memoria.
- **Campos LEN0 a LEN3 (longitud):** Especifican el tamaño de la zona de memoria referenciada por el correspondiente registro de dirección (DR0 a DR3). Sus valores posibles son:
  00 — 1 byte de longitud
  01 — 2 bytes de longitud
  10 — 8 bytes de longitud (sólo en algunos procesadores)
  11 — 4 bytes de longitud

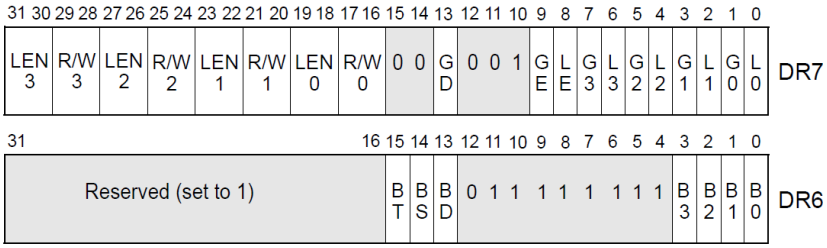


Figura 16: Registros de depuración DR6 y DR7 (tomado de [Intel 2012]).

Es importante aclarar que el procesador utiliza máscaras de bits para verificar los accesos a memoria. Por ejemplo, para comprobar si se cumple la condición de un breakpoint cuya zona de memoria es de 4 bytes de longitud, se utilizará la máscara 11b para poner a cero los dos bits menos significativos de la dirección accedida, antes de compararla con la dirección de breakpoint. La dirección de breakpoint siempre debe estar alineada según el tamaño de la zona a verificar.

La figura 16 también muestra la estructura del registro de estado de depuración (DR6). Sus bits más importantes son las banderas B0 a B3 que indican si la condición de sus breakpoints asociados se encuentra activa (indicado por el valor 1). Las restantes banderas tienen usos mas especializados y se explican con detalle en [Intel 2012]. El manejador de la excepción de depuración debería limpiar este registro antes de retornar.

3.1.6 Otras características de la arquitectura IA-32

Los procesadores de la arquitectura IA-32 presentan un gran número de características, cuya descripción sería demasiado extensa y se va del alcance de este documento. Algunas características no estaban presentes desde los primeros procesadores, sino que fueron agregadas posteriormente y se han vuelto parte importante de la arquitectura. En esta sección mencionaremos algunas de estas características, sin entrar en detalle. Para una descripción detallada de cada una de ellas consultar el manual [Intel 2012].

- **Hasta 5 registros de control (CR0 – CR4).** Determinan el modo de operación del procesador y características de la tarea en ejecución. Un registro de especial interés es el CR0, ya que (entre otras funciones) permite activar el modo protegido de 32 bits, y habilitar la paginación.
- **Registros de tipo de rango de memoria (MTRRs).** Son usados para asignar tipos de memoria a algunas regiones de memoria.
- **Registros específicos de máquina (MSR).** Utilizados para el control y reporte del desempeño del procesador. Como su nombre lo indica estos registros son específicos de algunos modelos de procesador y, a menos que el fabricante lo especifique explícitamente, no existe garantía de que van a ser soportados por otros. Prácticamente todos los registros MSR controlan funciones relacionadas con el sistema y no son accesibles a programas de aplicación (una excepción a esta regla es el registro *timestamp counter* o TSC, útil para mediciones de tiempo). Los registros MSR se leen y escriben mediante instrucciones especiales (RDMSR y WRMSR).
- **Contadores de monitorización de desempeño.** Permiten monitorizar eventos relacionados con el desempeño del procesador.
- **Registros y operaciones de la unidad de punto flotante x87.** El primer procesador de la arquitectura IA-32 (el 80386) no soportaba operaciones de punto flotante, aunque brindada la

posibilidad de trabajar en paralelo con un coprocesador matemático (el 80387) que soportaba este tipo de operaciones. Los siguientes procesadores de la arquitectura integraron la funcionalidad del coprocesador matemático en lo que denominamos unidad de punto flotante (FPU) x87.

- **Registros y operaciones multimedia.** Con la evolución de los procesadores IA-32 se ha introducido nuevos conjuntos de operaciones que soportan el modelo de "una instrucción, múltiples datos" (SIMM). Entre estos conjuntos tenemos la tecnología Intel® MMX™, y las extensiones SSE y SSE2, entre otros.

3.2 Arquitectura del controlador de interrupciones

Los controladores de interrupción son componentes de hardware que reciben líneas de interrupción, detectan su ocurrencia y se encargan de despachar las mismas al núcleo de CPU. Estos controladores son capaces de recordar la ocurrencia de ciertos tipos de interrupciones, para enviarlas de una en una a la CPU, de acuerdo con un orden de prioridad predefinido. Para mantener la cola de interrupciones pendientes, generalmente los controladores utilizan un registro (máscara de bits) denominado **registro de petición de interrupciones** (**IRR**).

Generalmente los controladores de interrupción utilizan un registro (máscara de bits), denominado **registro de interrupciones en servicio** (**ISR**), para recordar las interrupciones que están siendo atendidas por la CPU. El ISR posibilita que no se notifiquen a la CPU nuevas interrupciones que ya están siendo atendidas, o interrupciones de menos prioridad a las interrupciones en servicio. Cuando se termina de dar servicio una interrupción, la rutina de servicio debe notificar al controlador mediante un **comando de fin de interrupción** (**EOI**), o de lo contrario la interrupción no volverá a ocurrir nuevamente. Ante un comando EOI la respuesta típica del controlador es limpiar el bit correspondiente a la interrupción de más prioridad del registro ISR.

Los controladores de interrupción también permiten enmascarar (deshabilitar) algunas líneas de IRQ específicas.

Las primeras PCs con arquitectura IA-32 traían controladores de interrupción de tipo PIC 8259A, de forma similar a su predecesora la PC/AT. Posteriormente se agregó la posibilidad de utilizar controladores de interrupción avanzados (APIC). Ambos tipos de controladores serán explicados en subsecciones siguientes.

3.2.1 Formas de activación de interrupciones

Los controladores de interrupción disponibles en las PC convencionales suelen distinguir entre dos formas de detectar la activación de una línea de interrupción: IRQ activada por flanco (*edge triggered*) e IRQ sensible al nivel (*level triggered*). Una IRQ activada por flanco se detecta cuando existe una transición de nivel de voltaje en la línea de IRQ (por ejemplo un cambio de nivel bajo a nivel alto), la línea puede permanecer en alto sin generar otra interrupción. Por otro lado las IRQs sensibles al nivel se activan mientras exista un nivel de voltaje (por ejemplo un nivel alto) en la línea de IRQ. La figura 17 explica gráficamente las dos formas de activación de las interrupciones.

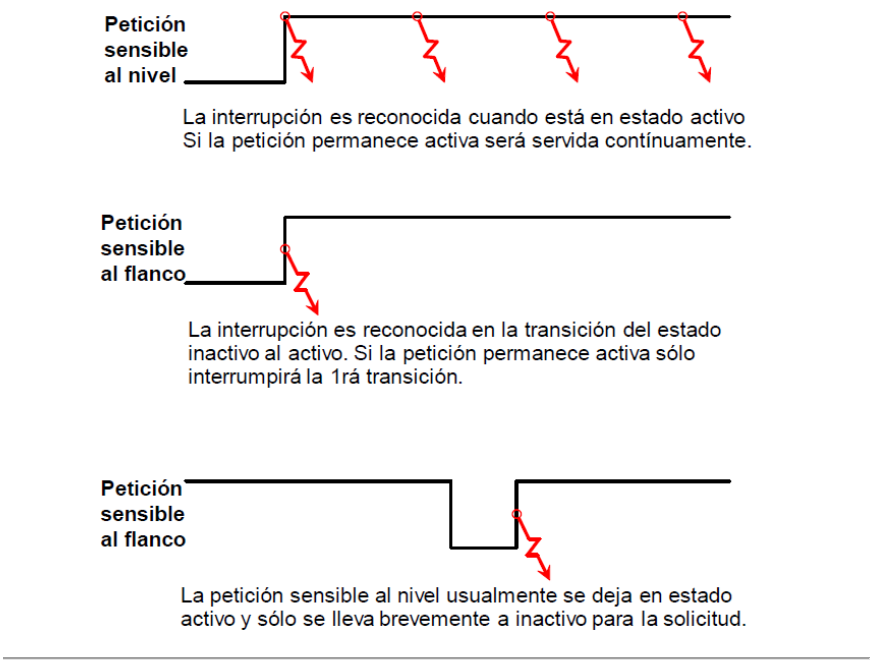


Figura 17: Activación de Interrupciones por nivel y por flanco.

Cuando ocurre una interrupción activada por nivel se debe retirar la solicitud de interrupción de la línea (eliminando la condición que la provocó) antes de que la interrupción quede habilitada nuevamente, de lo contrario la interrupción será notificada por segunda vez y la rutina de servicio será ejecutada de nuevo. Esto significa que la condición de interrupción debe eliminarse antes de enviar el comando EOI al controlador de interrupciones, a menos que la línea de IRQ sea enmascarada de otra forma. Si una línea de IRQ activada por nivel regresa al nivel lógico 0 antes de que el procesador sea notificado de su ocurrencia, la solicitud de interrupción será ignorada. Una ventaja de esta forma de activación de las interrupciones es que permite que la misma línea pueda ser fácilmente compartida entre varios dispositivos.

3.2.2 Arquitectura del controlador de interrupciones 8259

El Controlador de Interrupciones Programable (PIC) 8259A es un chip capaz de controlar hasta ocho fuentes de interrupción. Opcionalmente, a cada entrada de interrupción de un 8259 (Maestro) puede conectarse otro 8259A (Esclavo) para aumentar el número de IRQs que puede manejar el sistema.

La figura 18 muestra un esquema simplificado con los componentes fundamentales de un 8259A. Los pines IRx se conectan a las distintas líneas de interrupción que serán manejadas por el PIC. Los registros de 8 bits IRR, IMR, e ISR almacenan el estado de las ocho fuentes de interrupción a controlar. Los pines INT e INTA se conectan a la CPU. El 8259 tiene otras conexiones externas no mostradas por simplicidad en la figura.

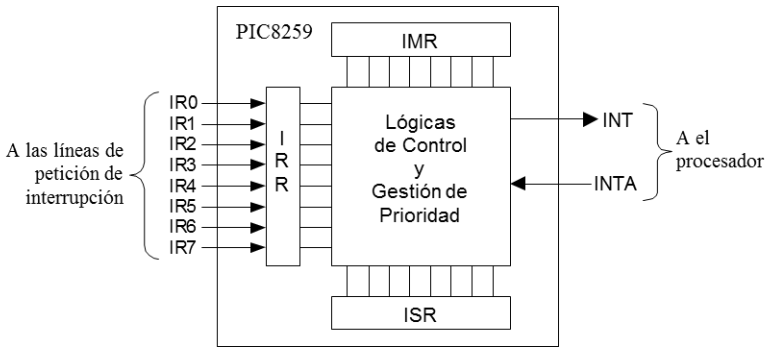


Figura 18: Esquema simplificado del PIC 8259A.

El registro IMR (*Interrupt Mask Register*) contiene la máscara de interrupciones del 8259. Cuando un bit de este registro está en 1, la interrupción correspondiente está enmascarada y de ocurrir no se señalará a la CPU.

Cuando se activa una línea IR esto se recuerda fijando su bit correspondiente en el registro IRR (*Interrupt Request Register*). Si la interrupción en cuestión no está enmascarada en el IMR, entonces es elegible por la lógica de gestión de prioridades para interrumpir la CPU. En caso de que el registro IRR indique la existencia de una interrupción pendiente de más prioridad que todas las que está atendiendo la CPU, el 8259 notificará a la CPU activando la línea de salida INT.

La CPU reconoce la interrupción activando la línea INTA del 8259, lo que provoca que el PIC limpie el bit correspondiente a la IRQ pendiente de más prioridad en IRR, y active el mismo bit en el registro ISR (*In Service Register*). Así, el registro ISR almacena todas las interrupciones que están actualmente en servicio.

Luego el procesador envía una segunda señal por la línea INTA, que provoca que el PIC deposite en el bus de datos del sistema un valor de ocho bits indicando el número del vector de interrupción correspondiente a la IRQ ocurrida. La CPU lee dicho número del bus para identificar la interrupción a la que debe dar servicio.

Al final de la rutina que da tratamiento a una interrupción, la CPU debe enviarle al 8259 el comando EOI. Cuando el PIC recibe este comando limpia el bit correspondiente a la IRQ en servicio de más prioridad en el registro ISR, indicando que la IRQ ya fue atendida. Una opción especial que soporta el 8259 es el **modo EOI automático** (AEOI), en este modo se hace innecesario el envío de comandos EOI, ya que el registro ISR se limpia automáticamente cuando la CPU reconoce la petición de interrupción.

Por compatibilidad con el PC/AT, todas las PC con procesadores IA-32 incluyen dos controladores PIC 8259A[1], conectados en cascada como muestra figura 19. Esta configuración permite soportar 15 líneas de IRQ.

[1] Las máquinas modernas realmente no contienen chips 8259 independientes, sino que estos están emulados por circuitos integrados en el chipset.

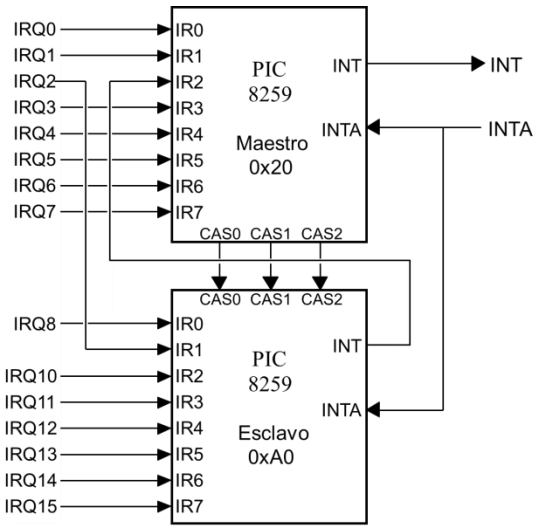


Figura 19: Conexión de los PICs 8259 en cascada.

3.2.3 Arquitectura del controlador de interrupciones APIC

El acrónimo APIC (*Advanced Programmable Interrupt Controller*) se usa para denominar a algunos tipos de dispositivos controladores de interrupción, que presentan características superiores al tradicional PIC 8259A. Podemos diferenciar dos tipos importantes de controladores APIC, los APIC locales (LAPIC) y los APIC de entrada/salida (IOAPIC), diseñados para trabajar en conjunto. Los APICs locales están integrados en el mismo chip junto con procesadores de la familia Intel P6 (Pentium Pro, II, III) y mas recientes, aunque se llegaron a producir APICs locales como chips independientes, para su uso con procesadores mas antiguos.

Los controladores APIC fueron especialmente diseñados para su uso en sistemas multiprocesador. En estos sistemas, cada núcleo de CPU tiene un LAPIC asociado que se encarga de procesar y despachar interrupciones al núcleo. Típicamente estos sistemas tienen un único IOAPIC, que recibe las líneas de IRQ y las envía a los APIC locales por medio de mensajes en un bus. De esta forma el IOAPIC distribuye las interrupciones entre todos los procesadores, ya sea de forma estática o dinámica. Aunque no es común, un sistema puede tener más de un IOAPIC, lo que puede ser útil cuando para más líneas de IRQ de las que soporta un solo IOAPIC.

En lo adelante, cuando mencionemos características que conciernan a tanto a los controladores APIC externos como a los locales, nos referiremos a ellos sencillamente como APIC.

Además de ser la solución en sistemas multiprocesador, el uso de controladores APIC presenta ciertas ventajas sobre los 8259s, incluso en sistemas con un solo procesador. Los IOAPIC nos permiten tener más de 15 IRQs lo que elimina o reduce grandemente la necesidad de multiplexar líneas IRQs para varias fuentes de interrupción. El acceso a los controladores LAPIC también es más rápido, por estar físicamente en el mismo chip que el núcleo de la CPU. Las prioridades de las IRQs pueden variarse sin importar la línea física, y a cada fuente de interrupción se le puede asignar un vector independiente de

las restantes fuentes. Es por eso que muchas computadoras modernas con un solo procesador también incluyen controladores IOAPIC[1] en su placa.

Aunque el LAPIC puede trabajar sin la existencia de un IOAPIC, es el uso combinado de IOAPIC y LAPIC los que nos permite explotar toda la potencia de estos controladores. El IOAPIC puede detectar muchas interrupciones y enviarlas a algún LAPIC mediante un mensaje en un bus[2]. Por su parte el LAPIC es capaz de priorizar y decidir que interrupciones deben despacharse al núcleo de CPU y cuales no. En esta sección nos enfocaremos en las características disponibles cuando se usan ambos tipos de controladores de conjunto.

El APIC local, además de recibir y priorizar las interrupciones provenientes de otros APICs (usualmente el IOAPIC), también puede manejar otras fuentes de interrupción, denominadas **fuentes de interrupciones locales**. Entre estas fuentes podemos destacar las interrupciones provenientes de un temporizador interno al LAPIC, y dos fuentes de interrupción conectadas a pines externos (LINTIN0 y LINTIN1). Es usual que la línea LINTIN0 esté conectada a un controlador tipo 8259 y LINTIN1 esté conectada a la interrupción no enmascarable (NMI), aunque esto puede variar según la configuración del sistema (ver [Intel 1997] sección 3.6 para mas información).

La figura 20 muestra un esquema de conexión típica de los diferentes componentes involucrados en el manejo de interrupciones, asumiendo un sistema multiprocesador antiguo donde el LAPIC era un chip independiente de la CPU. La figura 21 muestra un esquema similar, pero describe a los sistemas más modernos donde el LAPIC se encuentra integrado al chip de la CPU. Se puede notar como las líneas de IRQ llegan tanto a los 8259A como al IOAPIC, y como el 8259A puede estar conectado al IOAPIC, a los APIC locales, o directamente al procesador principal (BSP). La línea de interrupción no enmascarable se puede conectar directamente al BSP o a la entrada LINTIN1 de los APIC locales. Otra característica que queremos hacer notar es como **la IRQ0 (proveniente del temporizador 8254) se conecta a la entrada 2 del IOAPIC**, ya que la entrada 0 se utiliza para conectar la línea de interrupción proveniente del 8259A maestro.

Esta sección no constituye una guía detallada sobre el uso de los controladores APIC, sino que brinda un panorama general de que se puede hacer con estos, y proporciona las referencias necesarias para profundizar en los detalles que desee. La información brindada está enfocada a las características que pueden ser útiles a programadores de sistemas monoprocesadores (como PARTEMOS). Para una referencia detallada de los controladores APIC se recomienda consultar las referencias [Intel 1996], [Intel 1997], y [Intel 2012].

[1] Y por supuesto también incluyen un controlador LAPIC, que viene integrado dentro del chip de la CPU.
[2] Dependiendo del tipo de procesador este bus puede ser el propio bus del sistema, o un bus dedicado.

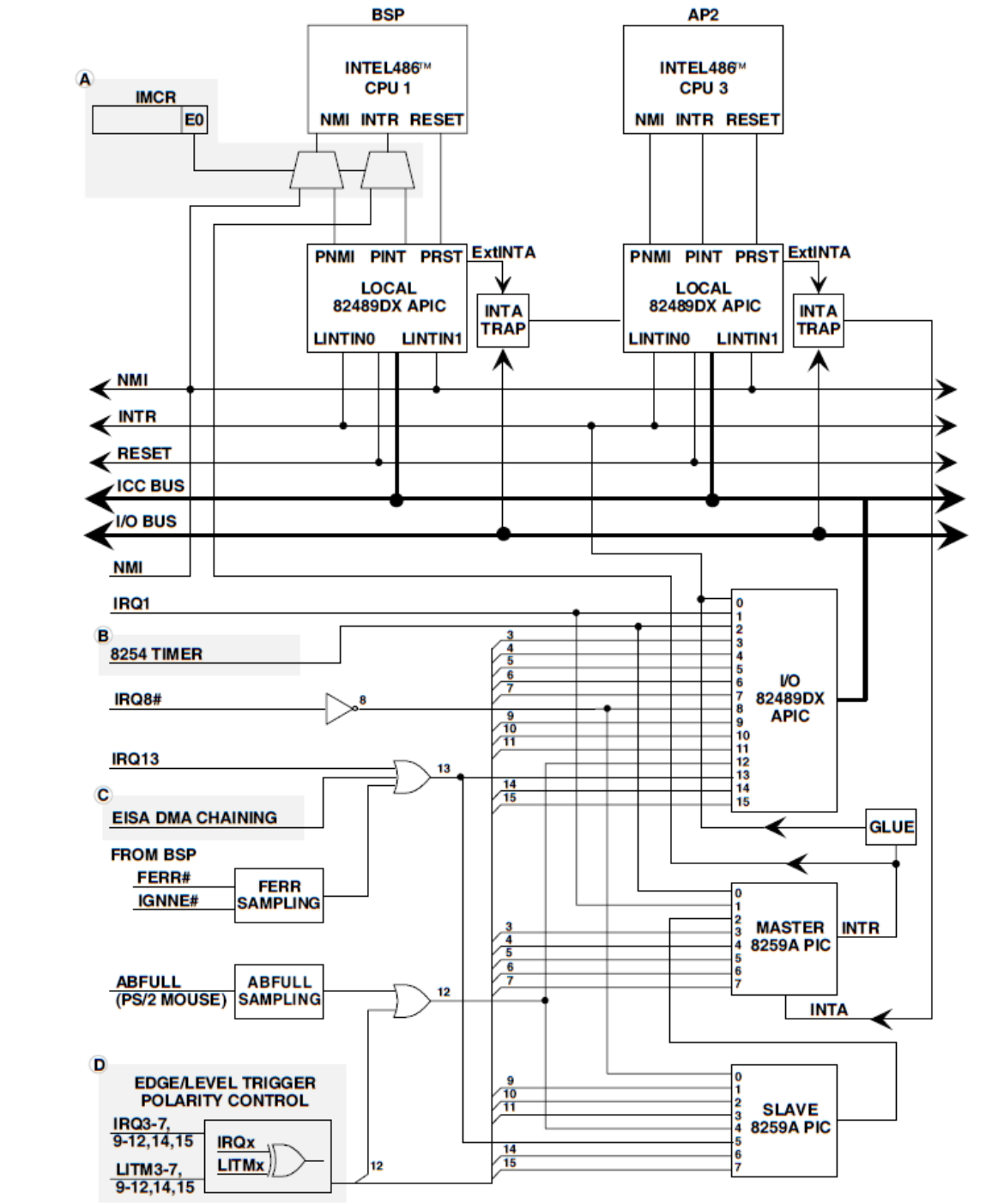


Figura 20: Configuración por defecto de sistemas con LAPIC discreto (tomado de [Intel 1997]).

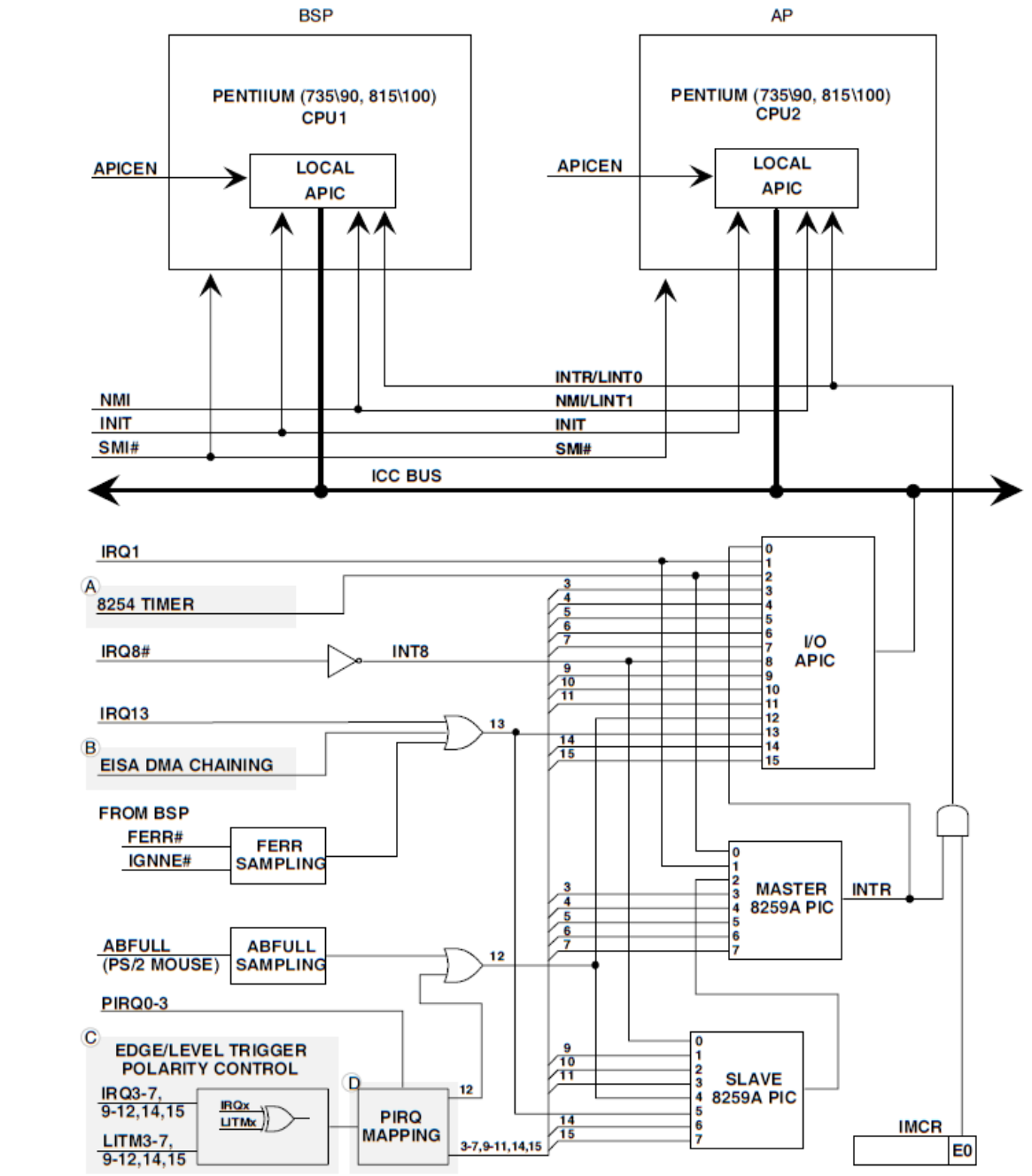


Figura 21: Configuración por defecto de sistemas con LAPIC integrado (tomado de [Intel 1997]).

Detección los controladores APIC

Para detectar la existencia de un IOAPIC (o varios) se deben leer las **tablas ACPI** ("*Advanced Configuration and Power Interface Specification*", [ACPI 2011]) o las **tablas multiprocesador** de Intel (tablas MP) [Intel 1997]. En las tablas MP, las tablas de configuración con valor de identificación de 2 describen IOAPICs, leyendo estas tablas se puede determinar cuantos IOAPICs existen (si es que existe alguno), cual es su ID, su dirección base y su primera IRQ. El capítulo 4 de [Intel 1997] brinda más información sobre las tablas MP.

Una PC típica contiene un único IOAPIC que soporta 24 líneas de IRQ, conectado como se muestra en la figura 21.

Comenzando con la familia de procesadores P6, la presencia de un APIC local en el mismo chip de la CPU puede detectarse usando la instrucción CPUID. Algunos códigos (en particular el código fuente de LINUX[1]) utilizan técnicas de detección más complejas como leer y escribir algunos registros del (supuestamente existente) LAPIC en busca de un comportamiento válido.

Aunque generalmente la presencia de un controlador externo IOAPIC garantiza que nuestro sistema dispone de APIC local, no es recomendable basarse únicamente en la detección de un IOAPIC para asegurar la presencia de LAPIC, ya que algunas PCs brindan la posibilidad de desactivar el LAPIC permanentemente, como una opción de configuración del setup-BIOS.

Programación del IOAPIC.

Cada IOAPIC tiene 2 ó 3 registros de 32 bits (dependiendo de la versión) y muchos registros de 64 bits (uno por IRQ). Los registros de 64 bits deben accederse mediante dos lecturas o escrituras de 32 bits. La tabla 1 lista los registros disponibles del IOAPIC, mostrando para cada uno su índice (utilizado para accederlo) y una breve descripción. Esta tabla se muestra a manera informativa, no entraremos en detalle de la funcionalidad de cada registro.

El acceso a los registros del IOAPIC se realiza mediante un indexado en memoria, para lo que se dispone de dos registros mapeados en memoria, un registro selector de índice llamado IOREGSEL, y otro para lectura/escritura de valores llamado IOREGWIN. Para leer o escribir un valor a un registro interno del IOAPIC, es necesario realizar dos operaciones: primero escribir el índice del registro interno deseado en IOREGSEL, y luego leer o escribir en IOREGWIN.

La tabla de redirección de interrupciones contiene un registro de 64 bits por cada IRQ que soporta el IOAPIC. Cada uno de estos registros se accede como dos registros de 32 bits. Por ejemplo, el registro asociado a la primera IRQ (IOREDTBL0) se accede por los índices 10H y 11H, para la segunda IRQ se usarían los registros 12H y 13H, y así sucesivamente.

La dirección base por defecto para el acceso al IOAPIC es FEC0000H, estando el registro IOREGSEL accesible en la dirección FEC00000H y el registro IOREGWIN en la FEC00010H. Para aumentar la flexibilidad del sistema, el espacio de direcciones de los dos registros de acceso al IOAPIC es relocalizable, aunque las direcciones siempre tendrán la forma FEC0 xy00h y FEC0 xy10h.

[1] ver código en http://lxr.free-electrons.com/source/arch/x86/kernel/apic/apic.c

Tabla 1: Registros del IOAPIC.

Registro	Índice	Descripción
IOAPICID	0	Registro de solo lectura que contiene el ID del IOAPIC.
IOAPICVER	1	Contiene la versión del IOAPIC en los bits 0-8, y la cantidad de IRQs que puede manejar el IOAPIC menos uno, codificada en los bits 16-23.
IOAPICARB	2	Contiene la prioridad de arbitraje del bus para el IOAPIC.
IOREDTBL	Desde 10H	Tabla de redirección de interrupciones, contiene dos registros de 32 bits por cada IRQ.

Programación del LAPIC

Todos los registros del LAPIC están mapeados a memoria y por tanto se pueden acceder mediante instrucciones de lectura o escritura a memoria. Los registros de LAPIC son de 32, 64 o 128 bits, y siempre están alineados en fronteras de 128 bits. Los registros de más de 32 bits deben ser accedidos mediante varias lecturas o escrituras de 32 bits, con el primer acceso alineado en 128 bits. Los accesos a las direcciones de memoria del LAPIC son interceptados por el procesador y nunca provocan acceso al bus externo, lo que permite que las operaciones sobre el LAPIC sean más rápidas, y que además varias CPUs puedan utilizar el mismo rango de direcciones para acceder a sus respectivos APICs locales.

La dirección base por defecto para el acceso al APIC local es FEE00000H. Esta se puede relocalizar a otra dirección física modificando el campo de dirección base de 24 bits del registro MSR denominado IA32_APIC_BASE. La tabla 2 muestra un listado de todos los registros del LAPIC y sus direcciones de acceso, asumiendo que se usa la dirección base por defecto. En próximas secciones explicaremos el uso de algunos de los registros más relevantes.

Las direcciones de memoria para acceso a los APICs deben ser configuradas como fuertemente no cacheables (UC, ver [Intel 2012], Vol. 3, capítulo 11 "*Memory Cache Control*").

Tabla 2: Mapa de direcciones de los registros del LAPIC.

Dirección	Nombre de Registro (inglés)
FEE0 0000H	Reservado
FEE0 0010H	Reservado
FEE0 0020H	Local APIC ID Register
FEE0 0030H	Local APIC Version Register
FEE0 0040H	Reservado
FEE0 0050H	Reservado
FEE0 0060H	Reservado
FEE0 0070H	Reservado
FEE0 0080H	Task Priority Register (TPR)
FEE0 0090H	Arbitration Priority Register (APR)
FEE0 00A0H	Processor Priority Register (PPR)
FEE0 00B0H	EOI Register
FEE0 00C0H	Remote Read Register1 (RRD)
FEE0 00D0H	Logical Destination Register
FEE0 00E0H	Destination Format Register
FEE0 00F0H	Spurious Interrupt Vector Register
FEE0 0100H a FEE0 0170H	In-Service Register (ISR)
FEE0 0180H a FEE0 01F0H	Trigger Mode Register (TMR)

Dirección	Nombre de Registro (inglés)
FEE0 0200H a FEE0 0270H	Interrupt Request Register (IRR)
FEE0 0280H	Error Status Register
FEE0 0290H a FEE0 02E0H	Reservado
FEE0 02F0H	LVT CMCI Register
FEE0 0300H	Interrupt Command Register (ICR); bits 0-31
FEE0 0310H	Interrupt Command Register (ICR); bits 32-63
FEE0 0320H	LVT Timer Register
FEE0 0330H	LVT Thermal Sensor Register2
FEE0 0340H	LVT Performance Monitoring Counters Register
FEE0 0350H	LVT LINT0 Register
FEE0 0360H	LVT LINT1 Register
FEE0 0370H	LVT Error Register
FEE0 0380H	Initial Count Register (for Timer)
FEE0 0390H	Current Count Register (for Timer)
FEE0 03ª0H a FEE0 03D0H	Reservado
FEE0 03E0H	Divide Configuration Register (for Timer)
FEE0 03F0H	Reservado

Activación de los APIC

Todos las PC, aun cuando dispongan de controladores APIC, mantienen por compatibilidad controladores de interrupción tipo 8259. Cuando la máquina enciende, esta queda inicializado de forma que son los 8259 quienes manejan las interrupciones del sistema. El sistema operativo debe encargarse de desactivar los 8259 y activar los APICs. Para desactivar los 8259, se deben enmascarar todas sus interrupciones.

Algunos sistemas permiten desconectar completamente el LAPIC del núcleo de la CPU de arranque (BSP o *Bootstrap processor*), y conectar el núcleo directamente a un controlador tipo PIC8259 (ver figura 20). El registro IMCR (*interrupt mode configuration register*) controla si el núcleo del BSP será conectado al PIC8259 o al LAPIC, por lo que en estos sistemas debemos habilitar la conexión entre el LAPIC y el núcleo del BSP escribiendo el valor adecuado en el registro IMCR. Para más información sobre el IMCR y otras configuraciones posibles de conexión entre los componentes de control de interrupciones ver [Intel 1997] sección 3.6.2.

Cuando la computadora enciende, los APIC locales pueden encontrarse desactivados temporalmente. Por eso es conveniente siempre activar el LAPIC poniendo a 1 el bit 8 del registro de interrupciones espurias, explicado más adelante.

El controlador IOAPIC no necesita activarse, sólo configurar de forma correcta cada línea de IRQ que le llega. Sin embargo para poder usar este controlador externo el APIC local debe estar activo, y conectado correctamente a través del IMCR.

Identificador de APIC (APIC ID)

Todos los controladores APIC (ya sean locales o externos) existentes en el sistema necesitan identificarse mediante un número único, esto permite que puedan enviarse mensajes entre si. Este identificador único para cada APIC, conocido como **APIC ID**, es un número de hasta 8 bits (puede ser menos según la versión del APIC). Al iniciar el sistema, el hardware asigna un APIC ID único a cada APIC existente.

El APIC ID puede leerse en los bits 24-31 del registro APIC ID (ver tabla 1 y tabla 2). La capacidad de modificar el APIC ID es dependiente de la versión específica del APIC, por lo que no se recomienda escribir a este registro.

Una de las pocas cosas para las que puede ser útil conocer el ID de LAPIC en un sistema monoprocesador es a la hora de configurar el IOAPIC. Es este caso se le puede decir al IOAPIC que dirija todas las interrupciones al APIC local pasándole el ID de este último.

Prioridades de interrupción, tarea y procesador

Cada fuente de interrupción tiene una prioridad, que es utilizada por el APIC local para determinar cuando se le puede dar servicio a la interrupción, comparándola con otras actividades en el procesador, incluyendo otras interrupciones en servicio.

La **prioridad de una interrupción** se obtiene a partir de los cuatro bits más significativos de su número de vector. El APIC sólo permite fijar vectores entre 16 y 255, por lo que los valores de prioridad válidos para las interrupciones van de 1 a 15, donde 1 es el valor más prioritario. Debido a que la arquitectura IA-32 reserva los vectores de 0 a 31 para excepciones del procesador, las prioridades de las interrupciones disponibles al usuario quedan en el rango de 2 a 15. A cada nivel de prioridad en el APIC también se le conoce como clase de prioridad de interrupción, ya que cada nivel comprende 16 vectores de interrupción.

El APIC local también define una **prioridad de tarea** y una **prioridad de procesador**, usadas para determinar que interrupciones pueden ser manejadas. La prioridad de tarea es un valor entre 0 y 15, y es fijado por el software escribiendo en el **registro de prioridad de tarea** (registro **TSKPRI**). La prioridad de tarea permite al software establecer un umbral de prioridad para interrumpir al procesador. Todas las interrupciones con prioridad menor o igual a la especificada en el registro de prioridad de tarea quedarán deshabilitadas. Los sistemas operativos pueden utilizar este mecanismo para impedir que ciertas interrupciones (de baja prioridad) interrumpan una tarea de alta prioridad actualmente en ejecución.

El **registro de prioridad de tarea**, al que llamaremos de forma abreviada **TSKPRI** (también conocido como **TPR** en la documentación de Intel) es un registro de 32 bits de los cuales sólo los primeros 8 bits han sido definidos. Los bits 4 a 7 definen la prioridad de tarea en al APIC, mientras los bits 0 a 3 determinan la “subclase de prioridad de tarea”. La subclase de prioridad de tarea no es tratada con

detalle en la documentación revisada, y al parecer sólo tiene importancia como valor de prioridad para envío de mensajes en sistemas multiprocesador. Los restantes bits (del 8 al 31) quedan reservados.

La **prioridad de procesador** es un valor calculado por el LAPIC para determinar que interrupciones pendientes pueden ser despachadas al núcleo del procesador. La prioridad de procesador puede determinarse leyendo el **registro de prioridad de procesador** (**PPR**, ver tabla 2), que es de solo lectura. El LAPIC calcula la prioridad de procesador como la mayor entre la prioridad de tarea actual y las prioridades de todas las interrupciones en servicio.

Configuración de las fuentes de interrupción

Cada APIC posee una tabla con una entrada por cada fuente de interrupción que puede manejar. En el caso del IOAPIC la tabla se denomina "tabla de redirección" (IOREDTBL) y posee una entrada de 64 bits para cada línea de IRQ que entra al controlador (típicamente 24 líneas). En el caso del APIC local la tabla se denomina "tabla de vectores locales" (LVT) y posee una entrada de 32 bits para cada fuente de interrupción local (ver [Intel 2012] Vol. 3 capítulo 10). Un LAPIC típico tiene entre 4 y 6 entradas en su LVT, dependiendo de la versión del procesador. Una entrada de la LVT que puede tener cierto interés es la 0, pues configura las interrupciones provenientes del temporizador interno del APIC. Cada entrada en las tablas LVT o IOREDTBL puede programarse de forma independiente para **indicar el vector de la interrupción** (y por ende **su prioridad**). Adicionalmente para cada fuente de interrupción se puede indicar si su línea de interrupción es sensible a flanco o nivel (para interrupciones provenientes de pines externos), el procesador de destino (en el caso del IOAPIC), si está o no enmascarada, el modo de trabajo del temporizador (para la interrupción de temporizador interno al APIC), entre otros datos.

El primer byte de cada entrada de configuración de las interrupciones contiene el vector de interrupción. Por tanto modificando este valor podemos cambiar la prioridad de la interrupción correspondiente. Para mas detalle sobre los restantes campos de estas entradas y su formato consultar [Intel 1996] e [Intel 2012].

Debemos resaltar que el IOAPIC encuesta las líneas de IRQ que le llegan de forma rotativa, sin importar el vector de interrupción configurado para las mismas en el controlador externo. Esto significa que el IOAPIC no utiliza ninguna prioridad para el envío de las interrupciones que le llegan. Es el APIC local quien se encarga de priorizarlas según su número de vector una vez que ha recibido el mensaje de interrupción proveniente del IOAPIC.

Comando de fin de interrupción (EOI)

Generalmente a cada interrupción despachada por el LAPIC hacia el núcleo de CPU se le debe dar un reconocimiento o comando fin de interrupción (EOI), para permitir que esta pueda ocurrir nuevamente. El comando EOI también permite que otras interrupciones pendientes de menos prioridad (registradas en el registro IRR) puedan ser despachadas al núcleo. Existen algunos tipos de interrupciones a las que no se les debe enviar EOI, entre ellas tenemos las de tipo NMI (no enmascarables), las interrupciones espurias (descritas más adelante), las interrupciones provenientes de un controlador tipo 8259 (conocidas como tipo ExtINT, en este caso el comando EOI debe enviarse como es usual al controlador 8259), y otras interrupciones de propósito especial.

Para enviar un comando EOI a un LAPIC, se debe escribir un valor cualquiera al **registro EOI** de dicho LAPIC (ver tabla 2). La escritura al registro EOI causa que el LAPIC borre la interrupción de la cola de interrupciones en servicio (registro **ISR**) y, en caso de ser activada por nivel, difunda un mensaje a los IOAPICs (si es que existe mas de uno) para indicar que la interrupción a sido manejada[1].

Interrupciones espurias

En algunas situaciones el LAPIC puede señalar incorrectamente la ocurrencia de una interrupción al núcleo de CPU. Este fenómeno se da, por ejemplo, cuando se eleva el nivel de prioridad del APIC (modificando el registro TASKPRI) por encima de la prioridad de una interrupción que ya ha sido señalada al núcleo, y está en fase de reconocimiento por parte de este. En esta situación el LAPIC está obligado a proporcionar un vector de interrupción al núcleo, pero como la IRQ que inicialmente provocó la interrupción ahora se encuentra enmascarada, el LAPIC proporcionará un **vector de interrupción espurio**.

Típicamente el software debe capturar el vector de interrupción espurio e ignorar la ocurrencia de dicha interrupción. Como las interrupciones espurias no afectan el registro ISR, el manejador de este vector debe retornar sin EOI.

El vector de interrupción espurio por defecto se fija al valor 255, aunque puede ser modificado por el software. Para más información ver [Intel 2012] Vol. 3, capítulo 10.

Manejo de errores en APIC

El LAPIC es capaz de detectar y registrar diferentes tipos de errores que pueden ocurrir durante su funcionamiento. Entre estos errores podemos mencionar errores en la suma de verificación de mensajes enviados o recibidos, direcciones de registros ilegales y vectores de interrupción ilegales. Cada vez que una situación de esta ocurre, el LAPIC fija un bit indicativo dentro del **registro de estado de error** (**ESR**, ver tabla 2). El formato del registro ESR y más indicaciones de como acceder correctamente al mismo se pueden encontrar en [Intel 2012] Vol. 3 sección 10.5.3.

El LAPIC es también capaz de generar una interrupción cuando ocurre alguno de los errores mencionados. Una entrada especial en la LVT (llamada *LVT error register*) permite configurar el vector de interrupción a generar en caso de error. Mediante esta entrada también se puede enmascarar la interrupción de error.

3.3 Decisiones de diseño para PARTEMOS sobre la arquitectura IA-32

La arquitectura IA-32 soporta características que no están presentes en la versión original de PARTEMOS, como la protección de memoria, la paginación, entre otras. Sacar provecho de estas características implicaría un rediseño de la arquitectura de PARTEMOS, lo cual no es objetivo del presente trabajo. En base a esto podemos afirmar lo siguiente acerca de la versión de 32 bits de PARTEMOS:

[1] Algunos APICs brindan la posibilidad de inhibir la difusión del mensaje de fin de interrupción hacia los IOAPICs, para que el software pueda enviar el mensaje a un IOAPIC específico.

- Al igual que su predecesor de 16 bits, PARTEMOS32 será un sistema monoprocesador . Actualmente existen muchas PC compatibles con la arquitectura IA-32 que poseen mas de un núcleo de CPU, en estos casos todo el sistema se ejecutará en el procesador principal o BSP ("*Bootstrap Processor*").
- No habrá protección de memoria entre procesos ni entre el código de usuario y de núcleo. Esto significa que todo el sistema, incluyendo las tareas de usuario, se ejecutará en el nivel de privilegio 0 del procesador (el mas privilegiado).
- La versión de 32 bits utilizará un modelo de memoria plano (que es similar al modelo de memoria usado en PARTEMOS16[1]), donde todas las direcciones que se accedan coincidirán con direcciones físicas.
- Siguiendo la implementación de PARTEMOS16, la nueva versión PARTEMOS no realizará salvas de los registros de punto flotante (que coinciden con los registros MMX). Por tanto las aplicaciones PARTEMOS no podrán hacer uso de estos registros, a menos que sean usados por una única tarea.
- No se utilizará paginación (y por tanto no se creará ningún tipo de tabla de página). El mecanismo de paginación es útil para crear esquemas de protección de memoria o implementar memoria virtual, funcionalidades que no están presentes en PARTEMOS16, y que por tanto no se incluirán en PARTEMOS32.

Aunque en la arquitectura IA-32 es obligatorio el uso de registros selectores de segmento, todos los selectores usados serán inicializados a una dirección base 0 y como tamaño máximo toda la memoria direccionable (FFFFFFFFH). De esta forma toda la memoria se verá como un gran segmento que contiene código, datos, espacios de entrada/salida, etc.).

Para permitir el modelo de memoria plano, la tabla de descriptores globales (GDT) contendrá únicamente 3 descriptores. La entrada 0 de la GDT será configurada como descriptor de segmento nulo, mientras las entradas 1 y 2 contendrán los descriptores de segmento para código y datos respectivamente, configurados para acceder a toda la memoria.

Como no se utilizará segmentación ni protección de memoria entre procesos, no es necesario mantener tablas de descriptores locales (LDT). Tampoco se usarán segmentos de estado de tarea (TSS) ya que su uso para conmutar entre tareas generalmente es más costoso que las técnicas de conmutación "manuales", ajustadas a las necesidades de nuestro sistema[2].

Se creará una tabla de descriptores de interrupción (IDT) completa, es decir, con 256 descriptores. Los primeras 32 entradas de la IDT se usarán como vectores para las excepciones de CPU. Las restantes entradas quedarán disponibles para que el módulo INTHAL[3] las utilice como desee, de acuerdo con el

[1] Aunque la compilación con modelo "small" de PARTEMOS16 utiliza dos segmentos de memoria disjuntos (uno para código y otro para datos), este modelo es similar al modelo plano en el sentido de que el código del núcleo no necesita tener en cuenta la presencia de segmentos. La variante de 16 bits de PARTEMOS32 continuará siendo compilada con un modelo de memoria "small".

[2] En caso de que en un futuro se quieran utilizar diferentes niveles de privilegio es obligatorio definir al menos un TSS, aun cuando no se utilice conmutación de tareas por hardware. Esto se debe a que los cambios de privilegio provocan cambio de pila, y el selector a los segmentos de pila se obtiene del TSS.

[3] Para más información sobre este y otros módulos consultar la sección 2.5.1, "Arquitectura de PARTEMOS".

controlador de interrupciones que use y sus peculiaridades de implementación internas. Las entradas de la IDT no usadas serán marcadas como no válidas.

En una primera etapa se realizará una implementación del INTHAL sobre el PIC 8259A, por ser el mismo controlador usado en PARTEMOS16. Los 8259 se configurarán para generar interrupciones por encima de 31, para no interferir con las excepciones de CPU. El BIOS deshabilita los APICs o los deja en un estado transparente luego de reiniciar la PC, por lo que no necesitamos preocuparnos de deshabilitar estos para usar los 8259.

En una segunda etapa se intentará realizar el control de interrupciones de PARTEMOS con controladores APIC, para lo que se deshabilitarán los 8259.

Capítulo 4
Entorno para el desarrollo de sistemas operativos de 32 bits

Para el desarrollo de la versión de 32 bits de PARTEMOS, es necesario hacerse de un conjunto de herramientas, que posibiliten la traducción del código fuente del sistema en un único archivo que contenga el código ejecutable del mismo, y que permitan la carga y ejecución de este archivo. Este conjunto de herramientas y las técnicas de uso de las mismas constituyen lo que denominamos entorno de desarrollo para sistemas operativos. La creación de este entorno de desarrollo es una tarea necesaria no solo para el desarrollo del micronúcleo PARTEMOS, sino que puede servir para el desarrollo de otros sistemas operativos de 32 bits.

El presente capítulo aborda la creación de un entorno para el desarrollo de sistemas operativos de 32 bits, incluyendo el proceso de selección de las herramientas del entorno, y las técnicas para utilizar las mismas en el desarrollo de sistemas. El entorno creado tiene como plataforma destino las computadoras personales (PC) con arquitectura IA-32, que es la misma elegida para el desarrollo de la primera versión de 32bits de PARTEMOS; aunque se trató de que las herramientas y técnicas elegidas puedan ser utilizadas o adaptadas fácilmente a otras plataformas. Aunque el objetivo buscado fue crear un entorno para el desarrollo de sistemas en general, también se mencionan algunas decisiones tomadas sobre el entorno para el desarrollo específico de PARTEMOS.

4.1 Requerimientos de un entorno para el desarrollo de sistemas operativos

Al igual que otros sistemas operativos, el código fuente de PARTEMOS está escrito mayoritariamente en lenguaje “C”, con algunas porciones escritas en ensamblador. En algunos sistemas (como PARTEMOS) el código de sistema se enlaza junto con el código de la aplicación a ejecutar, para conformar un único archivo denominado **imagen del sistema**, que es el resultado final en el proceso de desarrollo. En otros sistemas más acabados, las aplicaciones se compilan independientemente del núcleo y no forman parte de la imagen del sistema.

En el resto del capítulo asumiremos que las herramientas del entorno de desarrollo serán utilizadas para generar un núcleo de sistema operativo que, al igual que PARTEMOS, se debe enlazar en un único archivo imagen junto con la aplicación a ejecutar. Esto no afecta la generalidad de las herramientas seleccionadas, ya que el entorno de desarrollo de sistemas es el mismo aunque las aplicaciones se enlacen o no junto con el núcleo[1]. La figura 22 muestra el flujo de trabajo necesario para convertir el código fuente disponible (tanto de sistema como de aplicación) en la imagen del sistema.

Como paso intermedio a la generación del archivo imagen, todos los archivos fuente deben ser traducidos a archivos objeto binario, tarea que realiza un **compilador** en el caso de los archivos “C” y un programa **ensamblador** para el caso del código escrito en lenguaje ensamblador. Todos los archivos

[1] En caso de que las aplicaciones se compilen en archivos independientes al núcleo, es posible que además se requiera construir un entorno para el desarrollo de aplicaciones.

objetos se combinan en el archivo imagen del sistema, utilizando para ello un programa enlazador (*linker*).

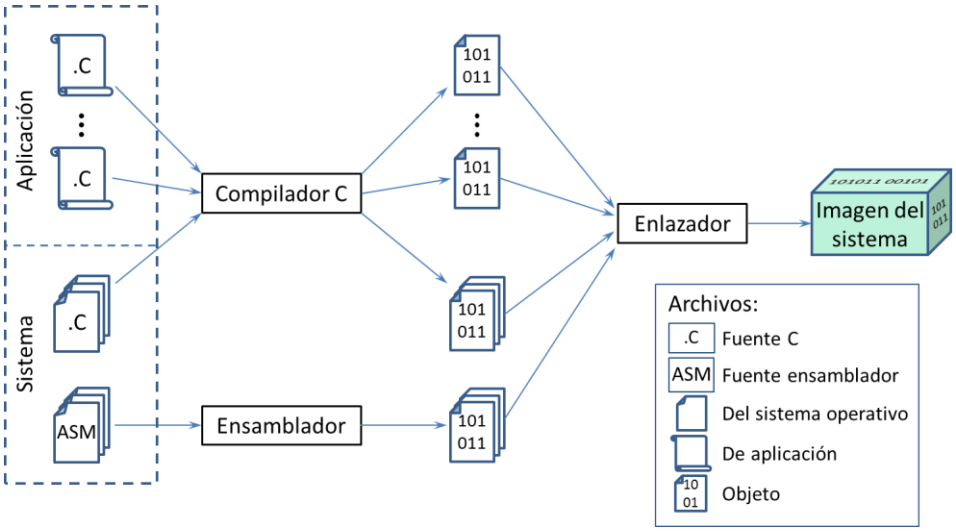


Figura 22: Flujo de generación de una imagen de sistema operativo.

Alternativamente, la generación de la imagen del sistema se puede hacer en dos fases (ver figura 23). Durante la primera fase todo el código del sistema es traducido a archivos objeto, que son encapsulados en un único archivo, denominado **biblioteca de enlazador**. En una segunda fase se compila el código de la aplicación, y se enlazan los archivos objeto resultantes junto con la biblioteca de enlazador, para generar la imagen del sistema.

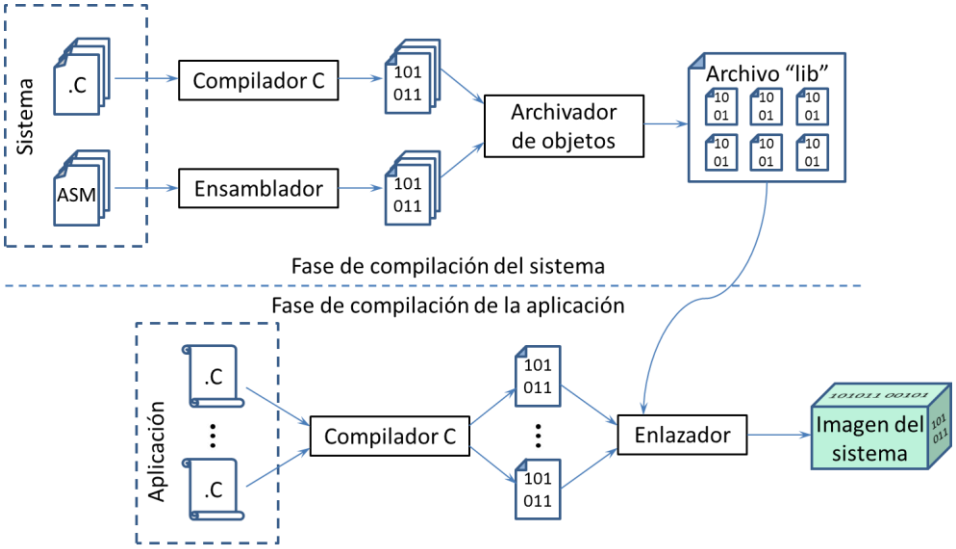


Figura 23: Flujo de trabajo de desarrollo en dos fases.

De lo visto anteriormente se concluye que **el entorno para el desarrollo de sistemas operativos debe contar con tres herramientas fundamentales: un ensamblador, un compilador, y un enlazador**; y opcionalmente con un programa para trabajo con bibliotecas de enlazador (que denominaremos archivador de objetos). Las herramientas deben ser elegidas de tal forma que el enlazador pueda "comprender" y enlazar mutuamente los archivos generados por las restantes herramientas.

Típicamente el compilador, el enlazador, y el archivador de objetos (si existe) forman parte de un paquete de software, que a veces también incluye un ensamblador.

Para ejecutar el sistema (y la aplicación) generado, se debe copiar el archivo imagen a un dispositivo de almacenamiento, que pueda servir como medio de arranque en la máquina destino (ver figura 24). Este medio de arranque debe contener un **software cargador**, encargado de copiar el archivo imagen (cargarlo) a la memoria de la máquina, y transferir el control al punto de inicio del sistema operativo.

Figura 24: Carga y ejecución del sistema.

El software cargador debe ser capaz de reconocer el formato del archivo imagen generado, por lo que también es un factor a tener en cuenta a la hora de elegir las herramientas de desarrollo.

La máquina destino puede ser una computadora física, o una **máquina virtual**. Una máquina virtual tiene todas las características de la plataforma destino, pero está simulada por un software dentro de la máquina de desarrollo. El uso de máquinas virtuales acelera el ciclo de desarrollo, ya que es más rápido copiar el sistema a la máquina virtual que a una física, y generalmente brindan algún tipo de soporte a la depuración del sistema.

4.2 Alternativas para los componentes del entorno

A continuación se analizan las alternativas disponibles para los diferentes componentes de un entorno para el desarrollo de sistemas operativos de 32 bits, asumiendo como plataforma destino una PC con procesador de la arquitectura IA-32.

4.2.1 Plataforma de desarrollo

Por ser la opción más común y económica, utilizamos computadoras personales PC para el desarrollo de sistema. Debido a que nuestro equipo de trabajo se encuentra familiarizado con los ambientes Windows, se ha elegido a esta familia de sistemas operativos como plataforma de ejecución del entorno de desarrollo creado. Sin embargo, las herramientas de desarrollo seleccionadas en su mayoría están disponibles para Linux, o en su defecto existen herramientas similares.

4.2.2 Compilador/Enlazador de C para 32 bits

Existen varias opciones posibles a elegir como compilador para el desarrollo de sistemas operativos. A continuación mencionaremos algunas de las herramientas de compilación que se encontraron entre las más factibles.

Watcom: Es una poderosa suite de herramientas de desarrollo. El compilador de C incluido tuvo la reputación de producir uno de los códigos más optimizados, aunque actualmente ya no tiene una comunidad de colaboradores tan activa. Una desventaja de este compilador es que sólo genera código para la plataforma x86.

Compiladores de Intel[1]: Famosos por realizar una fuerte optimización, los productos de desarrollo de Intel actualmente se encuentran disponibles para Windows, Linux, and Mac OS X. Según Intel sus productos son compatibles con Visual C++ y GCC. Entre sus desventajas resalta que se requiere comprar una licencia comercial para su uso, y como es de esperar sólo generan código para procesadores x86.

Visual Studio (Visual C++): El nombre Visual Studio se refiere a un paquete de herramientas de desarrollo de Microsoft, incluyendo compiladores y ambientes de desarrollo integrados (IDE). Aunque el Visual Studio como tal no es gratis, los compiladores de Microsoft si lo son, y existe una versión gratis del ambiente para desarrollo en C/C++ denominada **Visual C++ Express.** Visual Studio y las herramientas similares de Microsoft sólo se ejecutan sobre sistemas Windows de escritorio, y sólo generan código para plataformas que incluyan algún sistema operativo de Microsoft.

GCC: el *GNU Compiler Collection* (GCC) es un conjunto de herramientas de desarrollo, disponible para una amplia variedad de plataformas. Es una de las utilidades principales producidas por el proyecto GNU[2] de la fundación de software libre, y se encuentra en desarrollo constante. Aunque GCC es fuertemente soportado por las plataformas compatibles con POSIX, también ha sido portado a otros sistemas operativos como Windows. Entre las muchas arquitecturas de CPU soportadas por GCC se encuentran la x86, x86-64, IA-64, Alpha, SPARC, MIPS, PowerPC y ARM. GCC también es capaz de realizar compilación cruzada, lo que significa que con él se puede generar código para una plataforma diferente a la utilizada para el desarrollo. Debido a las ventajas antes mencionadas, GCC se ha convertido actualmente en una las opciones más recomendadas para el desarrollo de sistemas, y por lo mismo **fue elegido como el compilador a usar en el entorno de desarrollo de sistemas operativos de 32 bits**. GCC también ha sido adoptado como el compilador estándar por la mayoría de los sistemas operativos basados en UNIX, incluyendo Linux y Mac OS X.

El compilador[3] GCC se apoya en la colección de herramientas **binutils** (*GNU Binary Utilities*), que incluye herramientas para trabajo con archivos binarios como un enlazador ("ld"), y un archivador ("ar", para editar bibliotecas de enlazador). La colección binutils también incluye un ensamblador, denominado GAS (***G**nu **AS**sembler*).

Cuando se utiliza Windows como sistema operativo de la máquina de desarrollo, posiblemente la mejor opción es utilizar el compilador GCC sobre la plataforma **Cygwin[4]**. El paquete de software Cygwin proporciona una DLL que implementa la mayor parte del API POSIX encima de Windows, lo que permite portar fácilmente herramientas de GNU/Linux a Windows. Cygwin también incluye un software "gestor de paquetes", que permite descargar de internet e instalar muchas aplicaciones GNU/Linux que han sido portadas a Cygwin, incluyendo el compilador GCC.

[1] http://software.intel.com/en-us/articles/intel-compilers/
[2] http://www.gnu.org/
[3] Aunque GCC actualmente se refiere a una familia de compiladores que soportan muchos lenguajes, el software originalmente solo constaba de un compilador "C". En este documento llamaremos compilador GCC solamente al compilador "C", ya que los restantes lenguajes soportados no son utilizados.
[4] http://cygwin.com/

4.2.3 Ensambladores para la arquitectura IA-32

Prácticamente todos los sistemas operativos requieren al menos una pequeña porción de código escrita en lenguaje ensamblador. Este código no es portable y debe rescribirse para cada nueva plataforma soportada, por lo que no es requisito que el programa ensamblador elegido para una plataforma soporte una sintaxis específica. El programa ensamblador solo debe generar un código objeto compatible con las restantes herramientas de desarrollo. El ensamblador GNU Assembler (GAS) forma parte del paquete **binutils** (también usado por GCC), y por tanto es compatible con las herramientas de desarrollo seleccionadas. Sin embargo, el GAS presenta el inconveniente de que utiliza la sintaxis de AT&T para las instrucciones en ensamblador[1]. La sintaxis AT&T no solo es incómoda para los programadores acostumbrados a la sintaxis de Intel (por usar argumento en orden inverso a la sintaxis Intel) sino también es más oscura debido al uso de caracteres como "$" para valores inmediatos, o "%" para denotar registros.

Dado la experiencia previa con ensambladores con sintaxis de Intel para el desarrollo de PARTEMOS, se prefirió utilizar como programa ensamblador del entorno de desarrollo al **Netwide Assembler**[2] (NASM). NASM utiliza la notación de Intel para los opcodes y puede generar archivos de salida compatibles con binutils (en particular compatibles con el enlazador **GNU LD**).

4.2.4 Cargador

Debido a las facilidades que brinda, hemos decidido utilizar GRUB[3] como cargador del sistema operativo desarrollado. **GNU GRUB** *(GNU GRand Unified Bootloader)* es un cargador o gestor de arranque múltiple desarrollado por el proyecto GNU. Se utiliza para iniciar un sistema operativo instalado en la máquina, soportando más de un sistema instalado en el mismo equipo. **GRUB** es la implementación de referencia de la **especificación multiboot**[4] (*multiboot specification*), y posiblemente es el cargador más versátil y flexible que existe actualmente. Otros cargadores que siguen la especificación multiboot son LILO y Syslinux.

GRUB es utilizado principalmente en sistemas operativos GNU/Linux, y otros de código abierto. Adicionalmente, GRUB es capaz de reconocer y cargar sistemas de código cerrado que no cumplen con la especificación multiboot, como DOS y Windows. El sistema operativo Solaris ha usado GRUB como gestor de arranque en sistemas x86 desde la versión 10.

La especificación multiboot sigue la filosofía de que las imágenes de los sistemas deberían ser fáciles de crear. Idealmente la imagen de un sistema operativo debe ser un archivo ejecutable de 32 bits ordinario, en cualquiera que sea el formato que use el sistema operativo; sin necesidad de utilizar herramientas especializadas para crear imágenes en un formato especial. Adicionalmente, la filosofía de la especificación multiboot se basa en que generalmente lo más apropiado es liberar al sistema operativo de tareas que puede hacer el cargador, y que se deben realizar sólo al inicio; esto no sólo hace al sistema más sencillo de programar, sino que también ahorra memoria, debido a que toda la memoria consumida por el cargador típicamente queda disponible luego de que el sistema operativo tome el control. Cuando

[1] Las versiones más recientes de GAS soportan la sintaxis de Intel aunque con ciertas limitaciones.
[2] http://www.nasm.us/
[3] http://www.gnu.org/software/grub/ ó http://wiki.osdev.org/GRUB
[4] http://www.gnu.org/software/grub/manual/multiboot/multiboot.html

se usa un cargador compatible con la especificación multiboot, el sistema operativo no tiene que preocuparse por cosas como poner el sistema en modo de 32 bits o activar la línea A20[1].

Para evitar que los cargadores (*bootloaders*) tengan que interpretar los diferentes formatos ejecutables que existen (pues de otro modo se volverían específicos para algunos sistemas operativos), las imágenes compatibles con la especificación multiboot deben contener un encabezado multiboot. Este encabezado consta de un número mágico seguido de otros campos, y permite cargar la imagen del sistema sin necesidad de entender los diferentes formatos ejecutables. El encabezado mágico no necesita estar al inicio del archivo ejecutable, por lo que las imágenes de los núcleos aún pueden seguir cumpliendo con su formato binario en que se compiló.

En la próxima sección 4.3.1 se brinda más información sobre el formato de encabezado multiboot, y de como crear una imagen de núcleo compatible con esta especificación. En la documentación en línea de la especificación multiboot es posible encontrar información más detallada de la misma.

4.2.5 Simuladores de la plataforma de ejecución

Existen varias aplicaciones que pueden proporcionar una plataforma de ejecución simulada, denominada máquina virtual, con las mismas características que una PC. Estas máquinas virtuales permiten la ejecución de un sistema operativo sin necesidad de copiarlo a una máquina física de prueba, y a veces incluyen características útiles para la depuración del sistema.

Para ejecutar el software dentro de la máquina virtual, los simuladores de plataformas pueden utilizar dos técnicas fundamentales: la **emulación** y la **virtualización**. Cuando se utiliza emulación, la aplicación simuladora lee las instrucciones a ejecutar dentro de la máquina virtual, y las interpreta según su función. Durante la virtualización, las instrucciones de la máquina virtual son ejecutadas directamente por la CPU física, lo cual es mucho mas rápido pero requiere soporte de virtualización por parte del procesador. Existe una tercera alternativa conocida como **recompilación dinámica**, la cual traduce bloques de código máquina en instrucciones de la CPU física, para luego ejecutarlos directamente. Esta técnica se utiliza fundamentalmente cuando el procesador virtual es de una arquitectura diferente al procesador físico, por lo que este último no puede ejecutar directamente las instrucciones de la máquina virtual.

Se denomina hospedero (*host*) al sistema donde se ejecuta el software de máquina virtual, y huésped (*guest*) al sistema que se ejecuta dentro de la máquina virtual. Entre las aplicaciones de máquina virtual más comunes tenemos:

VMWare Workstation: Es un simulador de máquina virtual muy conocido, que utiliza virtualización para ejecutar la mayoría de las instrucciones de la máquina virtual. VMWare se caracteriza por tener una interfaz de usuario muy amigable y fácil de utilizar, y brindar facilidades para la utilización conjunta del sistema hospedero y uno o más sistemas huésped. El software de virtualización está pensado para el uso de los sistemas operativos más populares en entornos de producción, para lo cual brinda una buena velocidad de ejecución. Aunque es un software comercial, existe una variante gratuita conocida como "VMware Player", la cual está destinada a la ejecución de máquinas virtuales previamente creadas.

[1] Para ser compatibles con programas de 16 bits en modo real antiguos las placas madres tienen la opción de inhabilitar la línea 20 del bus de direcciones.

VMWare Workstation no brinda grandes beneficios para el desarrollador de nuevos sistemas operativos, y no siempre realiza una emulación perfecta del hardware de la PC. A partir de la versión Workstation 6.0, incluyó ciertas características útiles para desarrolladores de sistemas como la posibilidad de conectarse a la máquina virtual con un depurador GDB, o la posibilidad de registrar la ejecución de la máquina virtual, para reproducirla posteriormente; lo cual puede ser útil para reproducir condiciones de carrera.

Bochs: Es un simulador de código libre para la plataforma IA32, que se encuentra en continua mejora. Bochs brinda una emulación completa de una PC x86, incluyendo el procesador, dispositivos de hardware y la memoria. Debido a esta emulación, es uno de los programas de máquina virtual más lentos que existen, pero también uno de los que más exactamente emula la arquitectura de una PC. El uso de este software es poco intuitivo, y requiere de herramientas independientes y de línea de comandos para realizar acciones como la creación de discos virtuales. Para crear una nueva máquina virtual, se debe crear un archivo texto que contenga las opciones de configuración de la máquina.

Bochs incluye muchas características que pueden ser útiles a los desarrolladores de sistemas operativos, muchas de las cuales deben habilitarse mediante una opción en el archivo de configuración. El software incluye un depurador interno basado en línea de comandos, que puede conectarse con otros programas de interfaz para permitir depuración gráfica. Bochs brinda la posibilidad de escribir a un puerto de trazas (puerto 0xE9[1]), de forma que todas las escrituras desde la máquina virtual a este puerto se muestren en la consola de Bochs. Entre otras opciones de Bochs que facilitan la depuración de sistemas tenemos la posibilidad de redirigir la salida de los puertos serie y de impresora hacia un archivo texto, lo cual puede ser útil para crear trazas de ejecución.

A menos que se configure de otra forma, el temporizador de Bochs no funciona en tiempo real. Una espera por un tiempo determinado puede durar más o menos de ese tiempo, pero si un dispositivo virtual requiere de ese tiempo para estar listo, el emulador garantiza que estará listo al culminar la espera. Aunque esto no es bueno para la ejecución de sistemas en un entorno productivo, lo hace bueno para desarrollo, ya que la máquina virtual se comportará como una real (aunque con tiempos de respuestas diferentes).

Virtual Box: Es un software de virtualización gratuito, mayormente de código libre, aunque cuenta con algunos paquetes con código fuente cerrado. Es muy fácil de usar y tiene buen soporte para muchas plataformas, tanto de hardware hospedero como simuladas. Virtual Box utiliza una combinación de técnicas de virtualización asistida por hardware y virtualización basada en software (incluyendo recompilación dinámica) para lograr un rendimiento comparable a VMWare. El programa incluye un depurador integrado por línea de comandos, que permite examinar y controlar el estado de la máquina virtual.

QEMU: Es un emulador y virtualizador de máquinas genérico de código libre. Utiliza una técnica de traducción dinámica de binarios para obtener un buen rendimiento mientras su código se mantiene fácilmente portable entre plataformas. También incluye módulos que permiten utilizar la virtualización asistida por hardware cuando las plataformas lo permiten (como es el caso de la emulación de una PC

[1] En lo que resta de documento se utilizará el prefijo 0x (notación de C) para denotar números en hexadecimal.

sobre una PC). Al igual que Bochs, QEMU tampoco tiene una interfaz muy amigable, y su documentación técnica es escasa, lo que ha provocado que este emulador no goce de tanta popularidad.

QEMU brinda una consola a través de la cual se puede monitorizar el estado de la máquina virtual, y proporciona soporte nativo para el depurador GDB. Se ejecuta principalmente sobre plataformas tipo UNIX (Linux, FreeBSD y Mac OS X). La versión para Windows todavía se encuentra en fase de desarrollo alfa.

A diferencia de otras herramientas (como el compilador) no es obligatorio elegir una entre las muchas opciones de simuladores de plataforma disponibles. Es posible, y de hecho recomendable, elegir más de un emulador para probar el sistema, ya que la simulación del hardware virtual no es del todo fiable, y lo que funciona en una máquina virtual puede que no funcione en otra.

Para la ejecución de prueba de PARTEMOS, se utilizó fundamentalmente el emulador Bochs, auxiliado de ser necesario del software Peter Bochs, como herramienta de depuración visual. También se utilizó Virtual Box como segunda plataforma de prueba, y en menor medida por sus capacidades de depuración y por su capacidad de emular los registros de depuración de la arquitectura IA-32. PARTEMOS también ha sido comprobado en VMWare, aunque en menor medida. Algunas pruebas al módulo INTHAL de PARTEMOS sobre VMWare fallaron debido a que este software no emula correctamente la relación entre contador 0 del temporizador 8253 y la generación de la IRQ0.

El anexo B1 brinda más información sobre la instalación y uso del emulador Bochs y el depurador Peter Bochs.

4.2.6 Otras Herramientas

La mayor parte de las herramientas mencionadas hasta ahora constituyen elementos esenciales e indispensables para un entorno de desarrollo en 32 bits. Existen otras herramientas que pueden formar parte de este entorno de desarrollo, pero que no son indispensables, o que su selección depende mucho de los gustos personales del desarrollador, sin condicionar de ninguna forma el código del sistema operativo. La selección de estas herramientas permite a los desarrolladores crear su propio entorno personalizado, mientras mantienen las herramientas “núcleo”, que en este caso son el compilador GCC (junto con las herramientas del paquete binutils) y el ensamblador NASM.

Una herramienta necesaria para el desarrollo, pero que su elección depende mucho de los gustos personales son los **editores de texto**. Aunque cualquier editor de texto sin formato puede servir para editar el código del sistema y sus aplicaciones, se recomienda utilizar algunos que incluyan facilidades como el resaltado de sintaxis, integración con herramientas externas, búsqueda y remplazo avanzadas, edición de múltiples archivos simultáneamente, edición de texto en columna, comparación y combinación de archivos, automatización de acciones, entre otras. Entre los diversos editores avanzados que existen podemos mencionar Ultraedit, NotePad++ y PSPad.

Otro grupo de herramientas, necesarias para la ejecución de sistemas operativos en un hardware simulado (máquina virtual) son las herramientas para trabajo con **discos virtuales**, particularmente con archivos imágenes de discos flexibles. Una **imagen de un disco flexible** es un archivo binario que contiene una copia de los sectores en un disco flexible físico (*floppy disk*). Aunque los discos flexibles físicos están en desuso, sus imágenes siguen siendo útiles para usarlas como discos virtuales.

Prácticamente todas las aplicaciones de máquina virtual soportan el uso de archivos imágenes como discos virtuales, y estos constituyen un medio efectivo para arrancar la máquina, lo que unido a su pequeño tamaño los hace ideales como dispositivos de arranque.

Existen muchos archivos imágenes con alguna variante del cargador GRUB instalado, descargables desde internet. A estos archivos solo es necesario copiarles la imagen del núcleo generada para utilizarlos como medio de arranque del sistema. Esto se puede realizar con alguna **herramienta de edición de imágenes**, aunque posiblemente la mejor alternativa es utilizar una **herramienta de montaje de imágenes**. Las herramientas de montaje de imágenes permiten utilizar las imágenes como discos físicos en la máquina de desarrollo, lo que facilita la manipulación de archivos en la misma. Las herramientas de edición de imágenes también permiten la creación de nuevas imágenes, y la escritura de las mismas a discos físicos. El anexo B2 describe algunas de las herramientas de trabajo con imágenes de disco utilizadas en este trabajo.

En caso de que se quiera ejecutar el sistema generado en una máquina física, los programas para trabajo con imágenes de disco flexible no son suficientes, ya que no todas las computadoras disponibles hoy en día cuentan con unidades para leer estos discos. En ese caso es necesario instalar GRUB en algún medio de arranque físico, como un disco duro, o una memoria USB, para lo cual se requiere un **programa de instalación de GRUB**. El anexo B3 describe uno de los programas usados con este objetivo.

Otro conjunto de herramientas que también puede ser considerado a la hora de crear un entorno para el desarrollo de sistemas operativos es **MinGW**[1], el cual es un pequeño paquete de desarrollo que incluye versiones para Windows de herramientas GNU como GCC, Binutils, y GDB. Aunque la opción recomendada para el desarrollo de sistemas es utilizar versiones específicas para la plataforma Cygwin, contar con versiones específicas para Windows de estas herramientas permite invocarlas fácilmente desde editores de texto avanzados, los cuales capturan la salida y permiten localizar fácilmente las líneas de error. De forma similar, es posible que se desee descargar una versión nativa para Windows de NASM, con el objetivo de invocar al ensamblador desde el editor de texto usado. Alternativamente, se puede instalar el entorno de desarrollo DevC++, el cual incluye a MinGW y NASM, junto con otras herramientas gráficas de desarrollo.

4.2.7 Herramientas para desarrollo en 16 bits

Aunque este trabajo se enfoca en obtener una versión de 32 bits de PARTEMOS, uno de los objetivos del mismo es que esta versión sea fácilmente portable entre plataformas. Una forma de probar esta portabilidad es compilando PARTEMOS32 en 16 bits. Mantener una versión de 16 bits de PARTEMOS también tiene el beneficio de que se pueden usar herramientas de desarrollo y depuración ya conocidas, que pueden ser útiles para encontrar errores en la parte común (portable) del código del sistema.

El desarrollo de PARTEMOS en 16 bits se realiza utilizando las herramientas ya conocidas desde PARTEMOS16, que son fundamentalmente aplicaciones para MS-DOS como el entorno de desarrollo integrado “Borland C”, o el depurador “Turbo Debugger”, entre otras. Estas herramientas se ejecutan relativamente bien en entornos Windows de 32 bits, pero no son soportadas por versiones de 64 bits de este sistema operativo.

[1] http://mingw.org/

En caso de que se desee realizar desarrollo para 16 bits sobre versiones de Windows de 64 bits, es necesario utilizar un software auxiliar que permita la ejecución de programas MS-DOS sobre este entorno. En nuestro caso se probaron soluciones como instalar las herramientas de desarrollo necesarias sobre una máquina virtual con MS-DOS, pero la alternativa más sencilla encontrada fue utilizar el emulador DOSBox[1], que recrea un entorno similar a DOS para permitir la ejecución de programas escritos para este sistema operativo.

4.3 Uso del entorno de desarrollo

Hemos visto que las herramientas que conforman el entorno para el desarrollo de sistemas operativos de 32 bits se pueden dividir en dos grupos: las herramientas esenciales, y las herramientas opcionales. Las herramientas esenciales, también denominadas de núcleo, influyen en como se debe codificar el sistema, y en el formato de la imagen del núcleo, por lo que una vez elegidas no pueden ser cambiadas (a menos que se usen otras herramientas compatibles), y deben ser utilizadas por todos los desarrolladores del núcleo o de aplicaciones. Las herramientas opcionales, como los editores de texto o aplicaciones para trabajo con máquinas y discos virtuales, pueden elegirse según los gustos personales, o usarse más de una si se desea.

De la sección anterior podemos resumir que las herramientas núcleo elegidas para el desarrollo de PARTEMOS para la arquitectura IA-32 son las siguientes:

- Compilador GCC
- Ensamblador NASM
- Enlazador LD
- Cargador GRUB, o cualquiera que soporte la especificación multiboot.

Para evitar conflictos con los formatos de salida de las herramientas específicas de Windows, se decidió utilizar la plataforma Cygwin para ejecutar las herramientas de compilación, ensamblado y enlace. Adicionalmente, se creó una nueva versión personalizada del compilador GCC y el enlazador LD para Cygwin, a partir del código fuente de estas. Estas versiones constituyen lo que denominamos “compilador GCC cruzado”, y se describe mas adelante.

El Anexo A brinda detalles sobre la instalación de las herramientas de desarrollo núcleo, incluyendo la plataforma Cygwin, mientras que el Anexo B describe la instalación y uso de otras herramientas opcionales. Solo se mencionan las herramientas opcionales que fueron importantes para el desarrollo del presente trabajo, y que constituyen aplicaciones de uso poco frecuente. Entre las aplicaciones mencionadas tenemos al emulador Bochs, el depurador Peter Bochs, el cargador GRUB, y algunas herramientas para trabajo con discos virtuales.

[1] http://www.dosbox.com/

4.3.1 Compilador GCC cruzado

En lugar de utilizarse una versión precompilada de GCC, como las disponibles con los sistemas Linux y la versión de GCC para Cygwin, lo recomendado para el desarrollo de sistemas es recompilar una nueva versión de GCC, incluyendo el enlazador LD y otras herramientas del paquete **binutils**. Esta nueva versión proporciona un ambiente de compilación limpio que tiene ciertas ventajas como:

- Se eliminan referencias a rutinas dependientes del sistema como **`malloc`**, y otras bibliotecas.
- Se eliminan errores de enlace como "*PE operation on non-PE file*", frecuentes al utilizar versiones de GCC para Windows.
- Se eliminan otros posibles errores como que no se reconocen los símbolos creados en C y usados en ensamblador o viceversa, o que el cargador no puede leer la imagen del núcleo resultante.
- Se evita el uso accidental de archivos disponibles al compilador GCC común, y que no son aplicables al desarrollo de sistemas operativos, como las bibliotecas estándares de C[1].

A estas herramientas específicamente creadas para el desarrollo de sistemas operativos le llamaremos **herramientas de compilación cruzada**, o simplemente **compilador cruzado**. El término compilador cruzado se utiliza para denominar a una herramienta de compilación que se ejecuta en una plataforma, pero genera código para otra. En este caso ambas plataformas de hardware son las mismas, pero aun así utilizamos el término compilador cruzado, ya que existen diferencias entre los entornos de software de las plataformas de desarrollo y de ejecución, y además la técnica para generar las herramientas es la misma utilizada para generar otros compiladores cruzados.

En el Anexo A se brindan las instrucciones para generar las herramientas de compilación cruzada, ejecutables desde la plataforma Cygwin sobre Windows[2]. Como alternativa también se brindan las instrucciones para instalar el compilador cruzado utilizado en el presente trabajo, con lo que se evita tener que generar uno nuevo, y se garantiza que los resultados obtenidos sean los mismos.

Para evitar conflictos, las herramientas de compilación cruzada obtenidas de los procedimientos mencionados anteriormente se denominan de forma diferente a sus versiones "precompiladas". Para invocar a alguna de estas herramientas se debe utilizar el prefijo "`i586-elf-`", por ejemplo para invocar al compilador GCC se debe utilizar el nombre "`i586-elf-gcc`".

4.3.2 Creación y compilación de un núcleo con arranque a través de GRUB

A continuación se muestra, ayudados por un ejemplo, cómo se pueden utilizar las herramientas seleccionadas previamente para crear un núcleo de sistema operativo que pueda iniciar mediante GRUB. El código del sistema podrá contener cuantos archivos en ensamblador o en C se deseen; pero es obligatorio que contenga un archivo en ensamblador, que denominaremos "`c0hal.asm`", el cual contendrá el punto de entrada de nuestro sistema y el encabezado multiboot. En la figura 25 se lista el

[1] Muchas funciones de las bibliotecas de C utilizan funcionalidades específicas del sistema operativo, por lo que no deben utilizarse en el desarrollo de sistemas. Los desarrolladores de sistemas deben crear sus propias versiones de las bibliotecas estándares, ya sea para uso del sistema operativo o versiones para uso de las aplicaciones.

[2] En http://wiki.osdev.org/GCC_Cross-Compiler se pueden encontrar las instrucciones generales para crear un compilador GCC cruzado, aplicables a diferentes sistemas operativos de desarrollo.

código ensamblador del archivo ejemplo "`c0hal.asm`", en el que se pueden resaltar dos elementos importantes:

- **Punto de entrada al código**: definido por el símbolo "**`startx`**" (línea 4), es lo primero que se encuentra en el archivo. El código a partir de **`startx`** (líneas 5-8) realiza operaciones muy sencillas para configurar la pila e invocar a la función **`main`**, presumiblemente en un archivo C externo[1].
- **Encabezado multiboot**: esta sección (líneas 13-35) define el encabezado multiboot[2], necesario para que el cargador (GRUB) sepa como debe cargar el núcleo en memoria. Está dividida en dos partes: una primera parte que contiene valores conocidos como "campos mágicos" (líneas 24-26), y una segunda parte con campos de dirección (líneas 31-35).

El encabezado multiboot debe estar contenido completamente dentro de los primeros 8 KB de la imagen del sistema operativo, y alineado en una palabra de 32 bits. Se sugiere que este encabezado aparezca lo más cerca posible del inicio del archivo imagen del núcleo, por lo que la sección de código colocada en el punto de entrada debe ser pequeña. La directiva ALIGN en la línea 12 asegura la alineación del encabezado.

Dentro de la sección de campos mágicos, el primer campo (línea 24) se conoce como "*magic*" y contiene un número fijo (0x1BADB002) utilizado para identificar el encabezado. El segundo campo (línea 25) es conocido como "*flags*", y contiene varias banderas que indican características solicitadas o requeridas del cargador. En nuestro código ejemplo las banderas son definidas por las siguientes constantes:

`MBOOT_PAGE_ALIGN` (bit 0) —
: Indica que todos los módulos cargados junto con el sistema operativo deben estar alineados en fronteras de página (4 KB).

`MBOOT_MEMORY_INFO` (bit 1) —
: Hace que el cargador provea al sistema operativo con información de la memoria disponible.

`MBOOT_AOUT_KLUDGE` (bit 16) —
: Indica que la segunda parte del encabezado (campos de dirección) es válida y debe utilizarse para cargar el sistema. Este bit puede dejarse en cero si la imagen del núcleo está en formato ELF.

El último campo mágico (*"checksum"*, línea 26) contiene un valor de verificación, calculado de forma que al sumarse con los dos campos anteriores el resultado sea cero.

[1] Nótese que la función C externa se llama "**`main`**" en lugar de "**`_main`**". El compilador cruzado por defecto utiliza las convenciones de Linux, y por tanto no agregará el prefijo "**`_`**" a los símbolos definidos en C, como es común en los compiladores sobre DOS/Windows.

[2] En http://www.gnu.org/software/grub/manual/multiboot/multiboot.html es posible acceder a la especificación multiboot completa.

```
1  ; c0hal.asm
2  [BITS 32]
3  global startx
4  startx: ; Punto de entrada, se llega con interrupciones deshabilitadas
5    mov esp, main_stack     ; Fija esp a nuestra area de pila
6    extern main
7    call main
8    jmp $    ;Por seguridad, ciclo infinito al retorno de main
9
10 ; A continuacion viene el encabezado "multiboot"
11 ; debe estar alineado en una palabra de 32 bits
12 ALIGN 4
13 mboot:
14   ; macros para hacer las banderas mas legibles
15   MBOOT_PAGE_ALIGN  equ 1<<0
16   MBOOT_MEMORY_INFO equ 1<<1
17   MBOOT_AOUT_KLUDGE equ 1<<16
18   MBOOT_HEADER_MAGIC    equ 0x1BADB002
19   MBOOT_HEADER_FLAGS  equ MBOOT_PAGE_ALIGN | MBOOT_MEMORY_INFO | MBOOT_AOUT_KLUDGE
20   MBOOT_CHECKSUM    equ -(MBOOT_HEADER_MAGIC + MBOOT_HEADER_FLAGS)
21   EXTERN code, bss, end
22   ; la parte del encabezado
23   ; Contiene numero magico, banderas y suma de verificación
24   dd MBOOT_HEADER_MAGIC
25   dd MBOOT_HEADER_FLAGS
26   dd MBOOT_CHECKSUM
27   ; 2a parte del encabezado
28   ; contiene direcciones físicas para que el cargador
29   ; sepa donde y que cargar en memoria, y a donde ceder el control.
30   ; estos datos son llenados por el enlazador (guiado por el script)
31   dd mboot
32   dd code
33   dd bss
34   dd end
35   dd startx
36
37 ; ... AQUI PONER CUALQUIER CODIGO AUXILIAR DESEADO ...
38 ; Seccion BSS. Solo usada para almacenar la pila inicial
39 SECTION .bss
40 global main_stack_limit
41 main_stack_limit:
42     resb 8192               ; reservamos 8KB de memoria
43 main_stack:                 ; tope de pila inicial (la pila crece hacia abajo)
```

Figura 25: Código ensamblador con encabezado multiboot.

La segunda parte del encabezado multiboot contiene los siguientes campos:

header_addr (línea 31) —

Contiene la dirección física en la cual se supone que debe quedar el inicio del encabezado multiboot cuando se cargue el sistema en memoria. Gracias a este campo el cargador conoce donde poner la imagen del núcleo y le permite asociar direcciones dentro de la imagen con direcciones de memoria física.

load_addr (línea 32) —

Contiene la dirección física del inicio del segmento de código (sección ".text").

load_end_addr (línea 33) —

Contiene la dirección física del final del segmento de datos. La diferencia entre este campo y el anterior (load_end_addr - load_addr) define cuantos bytes se cargarán. Esto implica que los segmentos de código y datos deben estar consecutivos en la imagen del núcleo. Si este campo es cero, el cargador asume que los segmentos de código y datos ocupan todo el archivo con la imagen del núcleo.

bss_end_addr (línea 34) —

Contiene la dirección física del final del segmento bss. El cargador reserva esta área de memoria para evitar colocar otras estructuras en ella, y la inicializa en cero. Si este campo es cero se asume que no existe segmento bss.

entry_addr (línea 35) —

Contiene la dirección física a la cual el cargador debe saltar para iniciar la ejecución del sistema operativo.

Guiándose por los datos leídos del encabezado, GRUB carga la imagen del núcleo en memoria, y le cede el control de la CPU (saltando a **`startx`**). Está garantizado que el sistema obtendrá el control con las interrupciones deshabilitadas, y es obligatorio que el sistema reconfigure las tablas de descriptores IDT y GDT. La tarea de reconfigurar las tablas de descriptores queda relegada a la función main, y no será mostrada en este ejemplo. El capítulo 5 explica como se realiza esto dentro de la capa de abstracción de hardware de PARTEMOS.

Para **ensamblar** el código de "`c0hal.asm`" con NASM usaremos la siguiente línea de comandos, ejecutada sobre Cygwin:

```
nasm -f elf -o c0hal.o c0hal.asm
```

Esta línea invoca a NASM indicándole que utilice el formato de salida ELF, necesario para poder enlazar luego. La misma línea se usará para cualquier otro archivo en ensamblador que tengamos, utilizando el nombre adecuado para el archivo de entrada y de salida. Aunque no se muestra en la figura, si el archivo ensamblador no se encuentra en la carpeta actual también debe especificarse la ruta hasta él.

Los archivos en lenguaje C **deberán ser compilados** con el compilador GCC cruzado, como se muestra en la siguiente línea de comandos:

```
i586-elf-gcc -Wall –O -nostdinc -fno-builtin –I../include -c -o FILE.o FILE.c
```

En la línea anterior, "`i586-elf-gcc`" es el nombre del compilador GCC cruzado previamente generado. Se asume que se ha configurado el entorno Cygwin para que las herramientas de compilación cruzada estén en la ruta de búsqueda de comandos, como se explica en el Anexo A. La línea de comandos se puede incluir opciones de compilación válidas para GCC, las cuales están disponibles en la documentación en línea de este compilador. El ejemplo anterior utiliza dos opciones de compilación que mencionamos por ser importantes en el desarrollo de sistemas, estas son:

`-nostdinc`

impide que GCC busque en las carpetas "`include`" estándares del sistema.

`-fno-builtin`

impide que GCC use versiones internas de funciones estándares de C.

En la línea de comandos previamente mostrada, "`FILE.c`" representa el nombre del archivo a compilar. Y se asume que la carpeta "`include`" del proyecto se encuentra en la ruta relativa "`../include`".

Una vez creados los archivos objeto (con extensión ".o") para todos los archivos del núcleo, estos se deben **enlazar en una imagen de núcleo**, para lo cual debemos pasarle una serie de opciones al enlazador LD. Una forma conveniente de hacer esto es crear un archivo de configuración para el enlazador, también llamado guion de enlace ("*LD script*"). La figura 26 muestra un archivo de configuración, que funciona con el ejemplo de código dado anteriormente, y que llamaremos "`link.ld`".

```
 1 OUTPUT_FORMAT("binary")
 2 ENTRY(startx)
 3 phys = 0x00100000;
 4 SECTIONS
 5 {
 6   .text phys : AT(phys) {
 7     code = .;
 8     *(.text)
 9     *(.rodata)
10     . = ALIGN(4096);
11   }
12   .data : AT(phys + (data - code))
13   {
14     data = .;
15     *(.data)
16     . = ALIGN(4096);
17   }
18   .bss : AT(phys + (bss - code))
19   {
20     bss = .;
21     *(.bss)
22     . = ALIGN(4096);
23   }
24   end = .;
25 }
```

Figura 26: Ejemplo de guion de enlace.

La línea 1 del archivo "`link.ld`" le indica al enlazador que el archivo de salida sea un "binario plano", para mantenerlo sencillo. También se puede generar una imagen de núcleo en formato ELF, para lo cual se debe sustituir la opción "**`binary`**" por "**`elf32-i386`**". El formato ELF puede ser útil si deseamos incluir información simbólica dentro de la imagen del núcleo. En la línea 2, la palabra clave **`ENTRY`** indica cual archivo objeto será enlazado de primero en la imagen. Queremos que el archivo "`c0hal.o`", correspondiente al código en "`c0hal.asm`", sea el primer objeto enlazado; ya que en este se encuentra el encabezado multiboot. Por tanto debemos poner como argumento de ENTRY el símbolo **`startx`**, localizado al inicio del módulo.

La siguiente línea (3) define la variable **`phys`**, utilizada más adelante en el guion de enlace, y que indica la posición en memoria donde se cargará el núcleo. En este ejemplo estamos indicando que la imagen del núcleo se debe cargar a partir del primer megabyte de memoria. El cargador GRUB no soporta la

carga del núcleo en una dirección por debajo de 1Mb, posiblemente para evitar conflictos con zonas de memoria usadas por el BIOS.

La palabra clave **SECTIONS** (línea 4) indica que a continuación se definirán las secciones del archivo imagen. Si estudiamos el guion del enlazador podemos observar que se definen tres secciones: ".text", ".data", y ".bss", que deben ir obligatoriamente en ese orden (requisito de la especificación multiboot). Tambien se definen las variables **code**, **data**, **bss**, y **end**, que son utilizadas en el archivo "`c0hal.asm`". Las líneas 10, 16, y 22 al final de cada sección aseguraran que la sección siguiente comienze en una página separada de memoria.

Para enlazar el núcleo con el guión de enlace mostrado, se puede utilizar un comando como el siguiente:

```
i586-elf-ld -T link.ld -Map kernel.map -o kernel.bin c0hal.o file1.o …
```

Esta línea causa que se cree la imagen de núcleo en el archivo "`kernel.bin`", utilizando el guion de enlace "`link.ld`" definido previamente. Nótese como al final se listan todos los archivos objetos que hayamos creado para nuestro núcleo. La opción "`–Map`" provoca que se genere un archivo con el mapa de símbolos del núcleo, conteniendo todos los símbolos públicos y su posición final en memoria.

Otras opciones del compilador o enlazador que pueden ser de interés para la creación de sistemas operativos son "`-nostdlib`", "`-nostartfiles`", y "`-nodefaultlibs`", que impiden el enlace del núcleo con archivos ajenos a este (como archivos de la bibliotecas estándares), en nuestro caso no se utilizaron estas opciones porque el compilador cruzado no incluye estos archivos.

4.3.3 Creación y uso de bibliotecas de enlazador

Una biblioteca de enlazador es un archivo único que contiene una colección de otros archivos, normalmente archivos objeto resultantes de una compilación previa. La biblioteca de enlazador contiene toda la información (nombre, fecha, permisos, etc.) de los archivos que incluye para permitir extraerlos a su forma original. Las bibliotecas de enlazador pueden utilizarse directamente durante el enlace de objetos para generar una aplicación; lo que permite que el enlazador seleccione automáticamente que archivos objeto debe utilizar.

Durante la generación de un sistema en dos fases (ver figura 23), todo el código compilado del núcleo se encapsula en una biblioteca de enlazador. Este método tiene varias ventajas como que se puede distribuir el código de sistema operativo compilado en un único archivo en lugar de usar muchos archivos objeto, y sin necesidad de proporcionar el código fuente. La compilación de la aplicación final también resulta más sencilla ya que se enlaza contra el archivo biblioteca, sin necesidad de especificar cada archivo objeto necesario. Si la biblioteca de enlazador contiene módulos que no son necesarios para una aplicación específica, el enlazador simplemente los ignora.

Formando parte del paquete binutils se encuentra una herramienta para trabajo con bibliotecas de enlazador, denominada "**AR**". Esta herramienta permite crear bibliotecas, o modificar las existentes.

Para crear una biblioteca de enlazador que incluya los archivos objeto de un núcleo de sistema, usaremos la siguiente línea de comando:

```
i586-elf-ar rcs libkernel.a obj1.o obj2.o …
```

El comando anterior crea una biblioteca de enlazador llamada "`libkernel.a`", donde objx.o representan los archivos objetos que se quieren incluir. Nótese como el nombre del archivo comienza con el prefijo "lib" y termina con ".a", lo que es necesario para que el enlazador pueda posteriormente localizar el archivo. El nombre del comando usado es **`i586-elf-ar`** en lugar de **`ar`**, ya que estamos usando la versión cruzada de las herramientas binutils[1].

Una alternativa a las bibliotecas de enlazador comunes es la creación de "bibliotecas de enlazador delgadas", que contienen un índice de símbolos y una referencia a los archivos miembros originales. Las bibliotecas delgadas son útiles para realizar compilaciones locales en las que se espera que todos los archivos objetos originales estén disponibles, ahorrándose tiempo y espacio en disco. Evidentemente no son recomendadas para la distribución del núcleo compilado, ya que no incluyen los archivos objeto en su interior. Para crear una biblioteca de enlazador delgada, se debe agregar la letra "T" a las opciones del comando AR, así el ejemplo anterior quedaría:

```
i586-elf-ar rcsT libkernel.a obj1.o obj2.o …
```

Para utilizar la biblioteca de enlazador del núcleo previamente creada, se tienen que pasar al enlazador las opciones "`-L`" para indicar la ruta de la biblioteca, y "`-l`" para indicar su nombre. Por ejemplo, para enlazar el archivo del núcleo "`libkernel.a`" previamente generado con un archivo de aplicación llamado "`ejemplo.o`", localizados en la misma carpeta, se puede utilizar el siguiente comando:

```
i586-elf-ld -static -T link.ld  ejetst10.o -L. -lkernel -o kernel.bin
```

Nótese como con la opción "`-l`" se ha indicado "kernel" como nombre de archivo, el enlazador completa este nombre con el prefijo "lib" y el sufijo ".a" para construir el nombre de archivo "`libkernel.a`", que es precisamente la biblioteca de enlazador del núcleo. Es importante que el nombre de la biblioteca se especifique luego del archivo objeto (o los archivos, si es que hay más de uno), ya que el enlazador sólo incluye los archivos de la biblioteca necesarios para resolver los símbolos no definidos hasta el punto de su inclusión en la línea de comandos. La opción `-static` sólo está por seguridad y le indica al enlazador que no use bibliotecas compartidas (sólo estáticas). Al igual que la línea de enlace mostrada en la sección anterior, esta línea causa que se cree la imagen de núcleo en el archivo "`kernel.bin`", utilizando el guion de enlace "`link.ld`".

4.3.4 Scripts de compilación

Recompilar un proyecto con más de un archivo es una tarea repetitiva y tediosa, por lo que generalmente se busca alguna forma de automatizarla. Entre las alternativas existentes para automatizar esta tarea están los **scripts de shell** (o guiones del intérprete de comandos), y la herramienta **make**.

La herramienta **make** es una aplicación capaz de interpretar archivos scripts especiales con instrucciones sobre como generar un proyecto. Estos scripts de make (también conocidos como *makefiles*) permiten definir para cada archivo generado en el proyecto un listado de archivos de los que depende. De esta forma la herramienta make puede decidir cuales archivos se deben volver a generar, basado en la fecha de modificación de los mismos y sus dependencias.

[1] De todas formas la versión original de ar debe ser compatible y funcionar de forma idéntica.

Como el micronúcleo PARTEMOS se encuentra en constante evolución y desarrollo, las dependencias de cada módulo tienden a cambiar con el tiempo, lo que dificulta la creación y mantenimiento de archivos makefiles. Para evitar conflictos de dependencias entre archivos, actualmente siempre se recompilan todos los archivos del micronúcleo, sin importar cuales archivos fueron modificados desde la última compilación. Debido a lo antes mencionado, se prefirió utilizar scripts de shell para automatizar la tarea de regenerar todo el micrónucleo.

Durante el presente trabajo se crearon varios scripts de compilación que facilitan la compilación del núcleo, o sirven de ejemplo para compilación de aplicaciones PARTEMOS. Los capítulos 6 y 7 tienen secciones dedicadas a mostrar el uso de estos scripts de compilación.

4.3.5 Ejecución del núcleo

Una vez configurado el entorno de desarrollo, y generado el archivo con la imagen del núcleo, es posible utilizar esta imagen para ejecutar el sistema junto con la aplicación compilada, para lo cual se debe copiar a un dispositivo de arranque en la máquina de ejecución. En el capítulo 7 se muestra como ejecutar una aplicación PARTEMOS, ya sea en una máquina virtual o física.

Capítulo 5
Desarrollo de la capa de abstracción de hardware

Como se vio en el capítulo 2, la Capa de Abstracción de Hardware (HAL) de PARTEMOS brinda una interfaz que permite programar las capas superiores independientemente del hardware subyacente, facilitando así la portabilidad del micronúcleo a otras plataformas de hardware. Al ser dependiente del hardware subyacente, esta capa es la que más trabajo de programación implica a la hora de portar PARTEMOS a otras plataformas.

La capa HAL (especialmente los módulos INTHAL y CPUHAL) es esencial para poder compilar el resto del sistema operativo. Por otro lado estos módulos son relativamente independientes, no dependen del sistema e incluso pueden utilizarse como base para la construcción de otros sistemas. Es por eso que la programación de esta capa debe ser una de las primeras tareas a realizar durante el proceso de portar PARTEMOS a otras plataformas.

El presente capítulo describe el trabajo de desarrollo realizado sobre los diferentes módulos de esta capa. Gran parte de este trabajo estuvo dirigida a portar el HAL a la arquitectura Intel de 32 bits, pero no se limitó solamente a esto. Algunos módulos del HAL fueron rediseñados para hacerlos mas eficientes, o sufrieron cambios para simplificar su interfaz. Al final del capítulo se propone una metodología a seguir para portar esta capa a otras plataformas de hardware.

5.1 Compilación condicional

La **compilación condicional** es una característica de los compiladores de C, concretamente del preprocesador, que permite que ciertas secciones de código sean procesadas o ignoradas por el compilador. Con la introducción de la posibilidad de compilar el núcleo PARTEMOS para diferentes plataformas, el uso de la compilación condicional se volvió una característica indispensable. Los archivos de encabezado ".h" de la capa abstracción de hardware (HAL) son unos de los que mayor uso hacen de la compilación condicional, ya que en ellos se definen muchas macros cuya definición varía de una plataforma a otra.

Para realizar compilación condicional, pueden utilizarse las directivas de preprocesador **`#if`**, **`#ifdef`**, **`#ifndef`**, **`#else`**, **`#elif`** y **`#endif`**. Por ejemplo la siguiente sección de código

```
#ifdef __MSDOS__          /* Borland C 16 bits */
  #define BREAK asm int 3
#elif defined(__i386__)  /* GCC para IA-32    */
  #define BREAK __asm__("xchgw %bx, %bx")
#else
  #error Plataforma desconocida
#endif
```

define la macro **BREAK** (usada para ceder el control al depurador, si existe) dependiendo de si están definidos los símbolos **__MSDOS__** ó **__i386__**, y en caso de que ninguno de dichos símbolos esté definido aborta la compilación con un mesaje de error, para lo cual usa la directiva **#error**.

La mayoría de las veces en la condición de compilación condicional se verifica si está definido un símbolo (macro). Algunos símbolos predefinidos por el compilador permiten identificar el compilador usado o la plataforma destino, lo cual es muy útil para la compilación de diferentes versiones de PARTEMOS.

La versión de PARTEMOS resultante del presente trabajo es compilable para PCs en modo real de 16 bits, y para la arquitectura IA-32 en modo protegido, por lo que fue necesario investigar que símbolos podían identificar a cada una de estas plataformas.

La versión de 16 bits de PARTEMOS sólo compila con compiladores Borland/Turbo C para MS-DOS, los cuales definen símbolos como **__BORLANDC__** ó **__TURBOC__**. Estos símbolos podrían utilizarse para identificar esta plataforma, sin embargo también son definidos por otros compiladores de Borland para aplicaciones no MS-DOS (por ejemplo Windows), por lo que se decidió utilizar el símbolo **__MSDOS__** con este propósito. También se podría verificar la existencia simultánea de dos símbolos, como **__MSDOS__** y **__TURBOC__**, pero esto es innecesario ya que entre los compiladores para MS-DOS importantes sólo los de Borland definen el símbolo **__MSDOS__**.

Para detectar que estamos compilando código para la arquitectura IA-32, se podría utilizar el símbolo **__GNUC__**, definido únicamente por el compilador GCC. Sin embargo este símbolo es definido por GCC sin importar la plataforma destino, por lo que se prefirió utilizar el símbolo **__i386__**, definido por GCC solamente para la plataforma IA-32.

En general el uso de la compilación condicional debe limitarse a algunos archivos de encabezados ".h" específicos, **no debe escribirse código de compilación condicional dentro del código C del sistema**. Esto facilita el proceso de portado del núcleo a otras plataformas, y hace mas claro el código del núcleo.

En el archivo "`global.h`" se definen varias macros utilizando compilación condicional, listadas en la tabla 3. Algunos de las macros definidas en este archivo sirven como símbolos para posterior compilación condicional, por ejemplo el símbolo **ARCH32** define que la arquitectura es de 32 bits, y por tanto sirve para separar el código de cualquier plataforma de 32 bits. Otras macros realizan funciones específicas como abortar el sistema o activar un depurador específico de la plataforma, por lo que son definidas condicionalmente según la plataforma.

Aunque no se considera parte del HAL, este archivo "`global.h`" es de uso general en todas las capas del núcleo PARTEMOS, por lo que debe ser uno de los primeros objetos de trabajo al portar el HAL a otras plataformas. Para más detalles sobre el proceso de portar el HAL y su relación con "`global.h`" ver la sección 5.8 más adelante.

Tabla 3: Macros de "`global.h`".

Macro	Descripción
ARCH_PC	Indica que la computadora es una PC (16 o 32 bits).
ARCH_AT16	Definido si la plataforma es una PC-AT en modo real 16 bits.
ARCH16	Definido para cualquier plataforma de 16 bits.
ARCH_IA32	Definido para PCs con arquitectura IA-32 (en modo protegido).
ARCH32	Definido para cualquier plataforma de 32 bits.
ABORT	Usada para terminar el sistema de forma anormal.
BREAK	Detiene la ejecución y activa el depurador (si existe).

5.2 Descripción del marco de prueba

Las versiones de 32 bits de los principales módulos del HAL (el HAL de CPU y el Interrupciones) fueron creadas antes de portar el resto del sistema, y comprobadas individualmente. Para poder realizar pruebas a los diferentes módulos, se hizo necesario el desarrollo de otros módulos auxiliares, encargados de actividades como la carga y ejecución del código de la prueba, configuración de los descriptores de memoria y visualización de los resultados de las pruebas.

Para poder cargar el código de la prueba, y que este tome el control al encender la máquina, se utilizó la técnica descrita previamente en la sección 4.3.1. Un archivo escrito en ensamblador ("`SRC/HAL4IA32/c0hal.asm`"), muy similar al que se muestra en la sección mencionada, brinda el encabezado multiboot necesario para que la imagen compilada sea reconocida y cargada por GRUB. El cargador **GRUB cede el control a este archivo, el cual a su vez invoca la función externa `main`,** que es el punto de entrada inicial del núcleo, o en nuestro caso del código de la prueba a realizar. El archivo "`c0hal.asm`" también brinda funciones de bajo nivel utilizadas para cargar las tablas GDT e IDT.

El próximo paso a realizar antes de proceder con el resto de la prueba es inicializar las tablas de descriptores[1] GDT e IDT. Para hacer el código independiente del hardware, hemos llamado a este paso **inicialización primaria**, y lo hemos encapsulado en la macro "**`initPlatform()`**", definida dentro del archivo "`include/HAL/sysHAL.h`". Esta macro[2] debe definirse condicionalmente para cada plataforma que soporte el sistema, y su función es realizar cualquier tipo de inicialización requerida por la plataforma antes de poder ejecutar otro código, por tanto debe ser lo primero que se encuentre en la función **`main`.**

En el caso de la arquitectura IA-32, la inicialización primaria consiste en inicializar las tablas de descriptores GDT e IDT. Para ello la macro "**`initPlatform()`**" invoca a la función **`initDescTables`**, definida en el archivo "`SRC/HPC/IA32/descript.c`". La función

[1] Necesariamente GRUB debe configurar una GDT para poder ceder el control al núcleo en modo protegido, sin embargo, esta GDT debe considerarse temporal y remplazarse por una propia del núcleo lo antes posible.

[2] Nótese que el nombre de esta macro contiene paréntesis, esto fue realizado intencionalmente para permitir que el símbolo **`initPlatform`** sea definido como función en lugar de como macro (por ejemplo para otras plataformas), sin requerir cambios en el código que la utiliza.

initDescTables carga las tablas del sistema GDT e IDT, inicializando en la GDT los selectores de segmento de código y datos usados por PARTEMOS, para crear un modelo de memoria plano como se explicó en la sección 3.3. La IDT es creada con 256 entradas marcadas como "No presentes", por tanto la ocurrencia de cualquier interrupción o excepción en este estado provocará una doble falla y finalmente el sistema quedará congelado[1]. Las entradas de la IDT pueden ser posteriormente asignadas mediante el servicio **setInterruptGate,** brindado por el módulo "descript.c". La rutina **setInterruptGate** es utilizada por el HAL de interrupciones y el HAL de excepciones para establecer los puntos de entradas de las IRQs y excepciones de CPU.

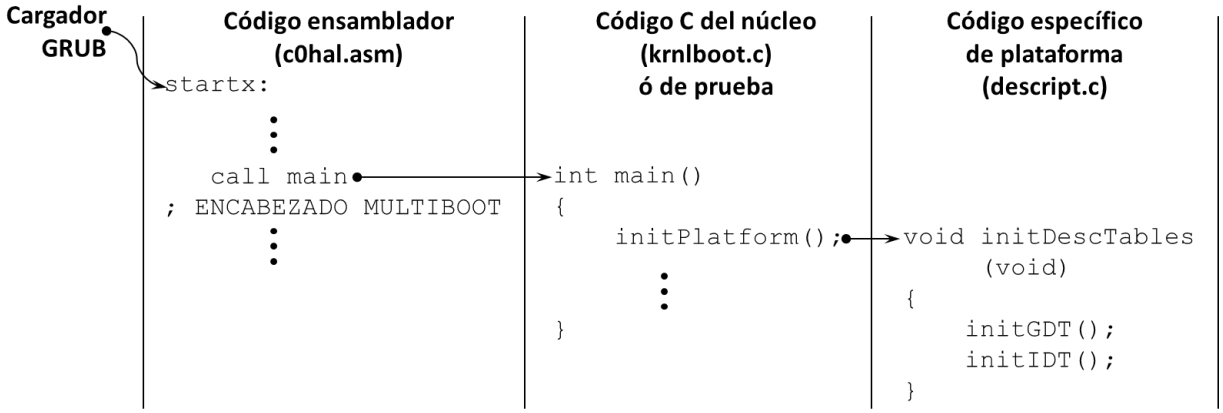


Figura 27: Secuencia de arranque del sistema.

La figura 27 muestra la secuencia de arranque explicada previamente. Nótese como la macro "**initPlatform()**" realmente provoca una invocación a la función **initDescTables**, ya que así fue definida en "sysHAL.h". Los archivos "c0hal.asm" y "syshal.h" conforman una sección de la capa HAL denominada "HAL de sistema".

Luego de la inicialización primaria, la función **main** puede proseguir con el código específico de la prueba. Este código depende del módulo a comprobar y de la funcionalidad a probar, pero algo en común que deben tener las pruebas es alguna manera de informar al programador acerca de la ejecución correcta o no de la misma.

Hemos desarrollado varias alternativas para obtener retroalimentación acerca de la ejecución de una prueba, las cuales se basan en salida a pantalla o en la escritura al puerto de trazas del emulador BOCHS. Estas no sólo informan acerca del resultado de la ejecución, sino que muchas veces son esenciales en la depuración del código.

Las técnicas de salida varían en complejidad, y van desde sencillas escrituras de caracteres individuales hasta salida de texto formateada a diferentes regiones de la pantalla (en modo texto). En las primeras etapas de desarrollo del HAL se utilizaron técnicas de salida sencillas, que posteriormente fueron

[1] Una excepción de doble falla (*Double Fault Exception*) ocurre cuando el procesador detecta una segunda excepción al intentar invocar un manejador para una excepción previa. Si ocurre una tercera excepción al intentar invocarse el manejador de doble falla la CPU entra en modo "congelado", similar al estado luego de ejecutar una instrucción HLT.

remplazadas por otros métodos de salida de mayor complejidad. En el próximo capítulo se dedica una sección a describir las diferentes técnicas de salida disponibles actualmente.

5.3 Desarrollo del HAL de interrupciones

La **Capa de Abstracción del Hardware de Interrupciones**, INTHAL (*Interrupt Hardware Abstraction Layer*) es la responsable del tratamiento de las interrupciones a más bajo nivel. Esta se ocupa de los aspectos dependientes del hardware de interrupción de la máquina, de forma que el resto del sistema sea lo más independiente posible de su arquitectura.

En esta sección se describe el trabajo realizado sobre el INTHAL de PARTEMOS, que incluye modificaciones a la interfaz del mismo, la adaptación del INTHAL sobre PIC8259 existente a la plataforma IA-32, y el uso de controladores APIC para la implementación de este módulo.

5.3.1 Papel del INTHAL en el modelo integrado de prioridades

El modelo integrado de prioridades de PARTEMOS (ver sección 2.5.2) está compuesto por 256 niveles de prioridad en un espacio común para interrupciones de hardware y tareas, donde 0 es la menor prioridad y 255 la máxima. Este modelo define **un nivel de prioridad de interrupción**, asociado con la prioridad de la tarea actualmente en ejecución. Ninguna IRQ con prioridad menor o igual a este nivel puede interrumpir la ejecución de la tarea actual.

El HAL de interrupciones juega un papel primordial en la implementación del modelo integrado de prioridades. El INTHAL suministra la capacidad de establecer el nivel de prioridad de interrupción del sistema. Adicionalmente, brinda la posibilidad de establecer dinámicamente la prioridad de cada IRQ. Esta prioridad se puede fijar a cualquier valor dentro del espacio de prioridades unificado, sin importar el esquema de prioridades del hardware de interrupciones. Una IRQ cuya prioridad en espacio unificado es cero queda deshabilitada y nunca puede interrumpir al procesador.

Dos servicios de la interfaz del INTHAL son esenciales para permitir el modelo integrado de prioridades de PARTEMOS: **`setIrqPriority`** y **`setIrqLevel`**. El servicio **`setIrqPriority`** puede ser utilizado en cualquier momento para modificar el nivel de prioridad de una IRQ. Por su parte **`setIrqLevel`** establece el nivel de interrupción actual del sistema, de forma que las IRQs con prioridad menor o igual a este nivel queden deshabilitadas.

Cada vez que una IRQ interrumpe la CPU (lo que normalmente sólo puede suceder si su prioridad es mayor que el nivel de interrupción actual del sistema) el INTHAL transfiere el control a la rutina **`IRQHandler`** del núcleo, pasándole como parámetro la IRQ correspondiente. La invocación a **`IRQHandler`** se realiza con interrupciones deshabilitadas a nivel del procesador. La ocurrencia de la interrupción también provoca que el INTHAL eleve automáticamente el nivel de interrupción a la prioridad de la IRQ ocurrida.

Generalmente, la funcionalidad del INTHAL no es soportada directamente por el hardware de control de interrupciones del sistema. El INTHAL debe encargarse de los detalles técnicos necesarios para implementar su modelo de interrupciones sobre el hardware disponible. Podemos considerar que el INTHAL brinda el acceso a un Controlador de Interrupciones Programable a la Medida Virtual o VCPIC

(*Virtual Custom Programmable Interrupt Controller*) [Leyva-del-Foyo 2008]. El VCPIC mantiene la prioridad actual para cada IRQ y el nivel de prioridad actual del sistema.

Se ha estudiado la posibilidad de implementar un controlador de interrupciones a la medida utilizando FPGAs, que implemente en hardware el modelo de prioridades de interrupción del VCPIC [Leyva-del-Foyo y Mejia-Alvarez 2004]. Con un controlador de este tipo, las rutinas del INTHAL únicamente realizarían una operación de acceso al hardware, sin necesidad de algoritmos para emular el modelo del INTHAL. Sin embargo, esta opción podría añadir costos económicos al sistema o incluso pudiera no ser posible en todas las arquitecturas.

El núcleo PARTEMOS soporta un modo de trabajo con **enmascaramiento de enmascaramiento de interrupciones virtual** [Leyva-del-Foyo *et al.* 2006b] dentro del INTHAL. En este modo se evita acceder al hardware de interrupciones para enmascarar interrupciones hasta tanto no ocurra una **interrupción indebida** (con prioridad menor o igual que el nivel de interrupción actual). Este modo de trabajo asume que no va a ocurrir ninguna interrupción indebida, lo cual es cierto en la mayoría de los casos y por tanto permite ahorrarse un gran número de costosos accesos al hardware. Sólo cuando ocurre una interrupción indebida el INTHAL accede al hardware para establecer físicamente el nivel de interrupción, e invoca a **`IRQHandler`** para notificar al núcleo pero sin modificar el nivel de interrupción actual. Es responsabilidad del núcleo recordar la ocurrencia de la IRQ.

5.3.2 Simplificación a la interfaz del INTHAL

Como parte de este trabajo se realizaron varios cambios a la interfaz brindada por el INTHAL, que la hicieron más sencilla y ajustada al modelo de prioridades de PARTEMOS. Anteriormente, el INTHAL brindaba las rutinas **`enableIrq`** y **`disableIrq`** para habilitar y deshabilitar la captura de IRQs. Se propuso una nueva filosofía donde todas las IRQs se consideran capturadas, aunque inicializadas con prioridad 0, lo que impide su ocurrencia. Según esta filosofía, las rutinas **`enableIrq`** y **`disableIrq`** ya no son necesarias, y sólo se necesita fijarle la prioridad a una IRQ (mediante **`setIRQPriority`**) para comenzar a recibirlas.

Otra rutina que también fue eliminada de la interfaz del INTHAL fue **`endIRQ`**, que tenía como función restaurar el nivel de interrupción existente previo a la ocurrencia de la última IRQ. Esta rutina puede ser útil en sistemas que permiten interrupciones anidadas, pero no es utilizada por el núcleo PARTEMOS, el cual siempre utiliza **`setIrqLevel`** para fijar el nivel del sistema. En caso de que se desee utilizar el INTHAL en otro núcleo que requiera la funcionalidad de **`endIRQ`**, queda como responsabilidad del núcleo recordar los niveles de prioridad previos y restaurarlos utilizando **`setIrqLevel`**.

El sistema PARTEMOS16 depende de MS-DOS para cargarse en memoria, y opcionalmente brinda la posibilidad de mantener la ejecución de los manejadores de interrupción brindados por DOS, gracias al uso de lo que denominamos interrupciones heredadas. Como la nueva versión de PARTEMOS no depende de otro sistema operativo para cargarse en memoria, no tiene sentido la existencia de interrupciones heredadas, las cuales fueron eliminadas del código (tanto para la versión de 16 bits como para la de 32 bits).

La nueva versión del INTHAL no soporta el concepto de prioridades por defecto para las IRQ (las prioridades por defecto eran utilizadas por **`enableIrq`**). Por tanto el servicio **`initIrqHardware`**,

que sirve para inicializar el INTHAL y se invoca como parte del arranque del núcleo del sistema, ya no recibe argumentos para las prioridades por defecto.

Anteriormente se mencionó la posibilidad de implementar el modelo de prioridades del INTHAL en un hardware controlador de interrupciones (sobre FPGA), el cual brindaría la misma funcionalidad que la interfaz del INTHAL. Con las simplificaciones al INTHAL antes mencionadas, la tarea de implementar este hardware sería mucho más sencilla. En particular, con la eliminación de la rutina **`endIRQ`** el HAL de interrupciones ya no necesita recordar el orden de ocurrencia de interrupciones anidadas, lo que hace su implementación más sencilla, tanto en software como en hardware.

Las simplificaciones al HAL de interrupciones mencionadas **no provocaron muchos cambios al código existente de PARTEMOS**. Se eliminaron las invocaciones a la rutina **`enableIrq`**, las cuales estaban seguidas de invocaciones a **`setIrqPriority`** (que ahora hace el trabajo necesario para habilitarla). Las llamadas a la rutina a **`disableIrq`** fueron remplazadas por invocaciones a **`setIrqPriority`** con prioridad 0. Como ya no existen prioridades por defecto para las IRQ ahora la función de inicialización del INTHAL (**`initIrqHardware`**) ya no recibe estas prioridades como parámetro.

5.3.3 Implementación del INTHAL de 32 bits sobre PIC 8259

El HAL de interrupciones para PARTEMOS16, cuyo detalles de implementación pueden encontrarse en [Leyva-del-Foyo *et al.* 2006a], fue escrito completamente en ensamblador. Sobre este código ensamblador se realizaron los cambios (simplificaciones) descritos en la sección anterior, creándose así un HAL de interrupciones útil para compilar la nueva versión de 16 bits de la de PARTEMOS. Sin embargo, para compilar la variante de 32 bits de PARTEMOS, se decidió rescribir la mayor parte del HAL de interrupciones en lenguaje C, en aras de mejorar la mantenibilidad y portabilidad del mismo. Aunque se utilizó un nuevo lenguaje de programación para su implementación, el INTHAL para 32 bits sobre controladores PIC8259 mantiene la misma técnica de anular el manejo de prioridades de los 8259s, y administrar las prioridades por software [Leyva-del-Foyo *et al.* 2006a] [Leyva-del-Foyo *et al.* 2006b].

Sólo una pequeña parte del INTHAL para 32 bits sobre el PIC8259 está escrita en lenguaje ensamblador, y se encuentra localizada en el archivo "`irqhal32.asm`", ubicado en la subcarpeta "`SRC/HAL4IA32`". Este archivo contiene los puntos de entrada de cada IRQ, y un código común de salva y restauración (en pila) de los registros de la CPU. Luego de la salva de los registros, el código ensamblador invoca a la rutina **`irqEntryC`** escrita en C (archivo "`ihal32.c`"), la cual se encarga del resto del procesamiento de la IRQ. Todos los datos salvados en la pila por el código ensamblador conforman una estructura de tipo **`IntContext`**, de la cual **`irqEntryC`** recibe un apuntador como parámetro. En teoría **`irqEntryC`** podría acceder a la estructura **`IntContext`** para conocer más del contexto interrumpido, o modificarlo antes de retornar, pero actualmente el único campo utilizado de la misma es el que contiene el número de la interrupción ocurrida.

La figura 28 muestra el flujo de control de las interrupciones a través del INTHAL, separando las secciones escritas en ensamblador y en C. Puede verse como los puntos de entrada de cada IRQ son etiquetas de la forma **`irqX`**, y lo único que hacen es poner en la pila el número de la IRQ ocurrida, antes de saltar a **`commonIrqEntry`**. Posteriormente se salvan todos los registros de la CPU en el mismo orden que define la estructura **`IntContext`**, y se pasa como parámetro (se apila) el valor actual del

tope de pila ESP, que constituye un apuntador a la estructura salvada. Cuando ocurre una interrupción la CPU automáticamente limpia la bandera de interrupción, por lo que todo el código descrito se ejecuta con interrupciones deshabilitadas. El registro de banderas (EFLAGS) es apilado automáticamente por la CPU, junto con los valores de retorno CS:EIP, estos campos también forman parte de la estructura **`IntContext`.**

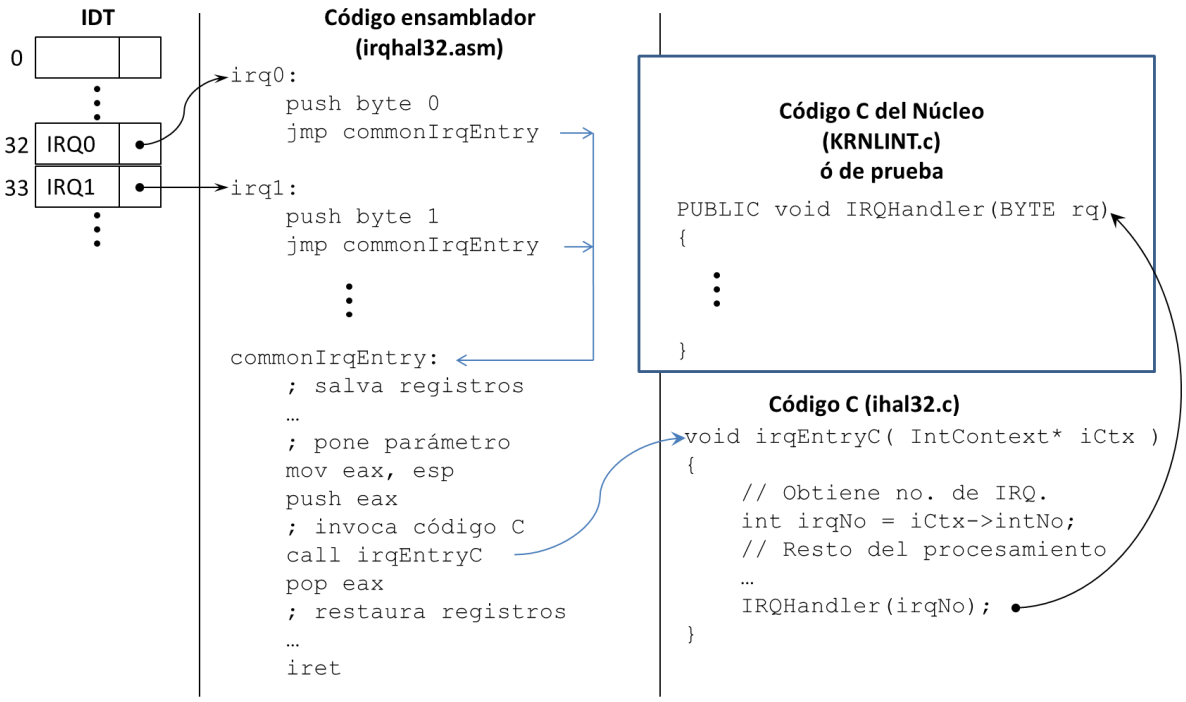


Figura 28: Flujo de control para el manejo de una IRQ.

Una de las primeras acciones que realiza la rutina de inicialización del INTHAL (**`initIrqHardware`**) es configurar los controladores 8259 del sistema, tarea que realiza invocando a la rutina interna **`Init8259`**. La rutina **`Init8259`** configura los controladores 8259 para que generen interrupciones en el rango 32-47, lo cual es necesario debido a que los vectores 0-31 están reservados para excepciones de la CPU. Como parte de la inicialización también se enmascaran todas las IRQs en el 8259.

Para capturar todas las IRQs, el servicio **`initIrqHardware`** se apoya en la rutina interna **`initIrqVectors`**, la cual a su vez utiliza la rutina **`setInterruptGate`** para configurar las entradas 32-47 de la IDT de forma que apunten a los puntos de entrada **`irqX`** antes mencionados, como se muestra en la figura 28. Las entradas en la IDT se configuran como compuertas de interrupción para indicar a la CPU que limpie la bandera de interrupción cada vez que ocurra una IRQ.

Una diferencia notable de esta implementación del INTHAL respecto a la implementación anterior usada en PARTEMOS16 es que ahora no existen los estados de IRQ (capturada, ignorada, etc., ver [Leyva-del-Foyo 2008] para una descripción de los diferentes estados), y por tanto sólo existe una única ruta de ejecución para todas las IRQs. Ahora es responsabilidad de **`irqEntryC`** verificar si una IRQ ocurrió de forma indebida y realizar el tratamiento de error si es necesario.

Al igual que su predecesora, la versión de 32 bits del INTHAL sobre 8259 permite activar o desactivar los modos de "EOI automático" y "enmascaramiento virtual".

Las estructuras de datos para mantener el estado del INTHAL se mantienen fundamentalmente iguales que en la versión para PARTEMOS16 [Leyva-del-Foyo 2008], aunque ahora están implementadas en lenguaje C. Las variables **`OLD_LEVEL`** y **`OLD_Mask`**, asociadas al antiguo servicio **`endIrq`**, no fueron incluidas en esta versión.

Los servicios del INTHAL mantienen la misma lógica que la versión para PARTEMOS16, sólo que rescrita en lenguaje C. El único segmento de código que tiene modificaciones importantes es el manejador de interrupciones **`irqEntryC`**, ahora esta función combina la lógica de manejo de interrupciones que anteriormente estaba dividida entre manejadores de interrupciones capturadas y no capturadas. La figura 29 muestra el código C de la función **`irqEntryC.`**

```
1  void irqEntryC( IntContext* iCtx )
2  {
3      int irqNo = iCtx->intNo;
4      if( IRQ_Priority[irqNo] <= IRQ_Level ){ /*¿Es una IRQ indebida?      */
5          if( OptimisticIMR ) {               /*¿Modo de máscara Optimista?*/
6              /* hacemos efectiva la máscara lógica de forma
7               * que no vuelvan a ocurrir IRQs no deseadas */
8              Set8259FisicIMR(Logical_Mask);
9              undesired++;
10         }
11         else {
12             SendEOI( irqNo );
13             panic(irqNo);           /* Pánico, esta IRQ no debió ocurrir */
14             return;
15         }
16     } else {                       /* La IRQ es válida segun su prioridad */
17         if( OptimisticIMR ) {               /*¿Modo de máscara Optimista?*/
18             Logical_Mask = IRQ_Mask[irqNo]; /*Actualiza la máscara lógica*/
19         }
20         else {
21            /* Modo de Mascara Física. Enmascara todas las IRQs con
22             * prioridad menor o igual a la prioridad de la IRQ activa */
23             Set8259FisicIMR( IRQ_Mask[irqNo] );
24         }
25         IRQ_Level = IRQ_Priority[irqNo];  /*Fija el nivel de interrupción*/
26     }
27     SendEOI( irqNo );                  /*Envía EOI si el modo lo necesita*/
28     /* Invoca al manejador de interrupción del núcleo IRQHandler(IRQ)
29      * La entrada IRQHandler se produce con interrupciones INHABILITADAS */
30     IRQHandler(irqNo);
31 }
```

Figura 29: Listado del código de la función "irqEntryC".

Como se mencionó previamente, la función **`irqEntryC`** recibe el control de un pequeño código escrito en ensamblador, que se encarga de la salva y restauración de registros en pila, y de pasar sus argumentos**.** Lo primero que realiza **`irqEntryC`** es obtener (de la estructura **`IntContext`** que recibe) el número de la IRQ ocurrida (ver figura 29, línea 3). Posteriormente verifica si la IRQ podía ocurrir según su prioridad (línea 4). Una IRQ sólo puede ocurrir si su prioridad es mayor que el nivel de interrupción actualmente definido en el INTHAL, en caso contrario se considera una ocurrencia indebida. Una

interrupción indebida puede provocar que el INTHAL fije físicamente la máscara de interrupción (línea 8) si está en modo de enmascaramiento virtual, o notifique el error al núcleo en caso contrario (línea 13).

Si una IRQ ocurrida es válida (su prioridad es mayor que el nivel de interrupción), entonces puede ser tratada de dos formas, dependiendo del modo de enmascaramiento activo. Si el modo de enmascaramiento virtual está habilitado simplemente se actualiza una máscara lógica (línea 18), en caso contrario se fija esta máscara físicamente en los 8259 (línea 23). La máscara lógica sólo se fija físicamente en el caso (raro) de que posteriormente ocurra una IRQ indebida, lo que ya vimos es realizado por la línea 8 del código de **`irqEntryC`** listado. Sin importar el modo de enmascaramiento usado una IRQ válida siempre provocará que se actualice el nivel de interrupción del sistema (línea 25).

En las líneas 12 y 27 del código listado se puede notar la invocación a la rutina **`SendEOI`**, esta encapsula la lógica necesaria para enviar el reconocimiento de interrupción a los 8259, teniendo en cuenta si tenemos activo o no el modo de EOI automático. Por último, si todo ocurrió sin error, se invoca al manejador de la capa superior **`IRQHandler`** (línea 30).

5.3.4 Implementación del INTHAL sobre controladores APIC

Los controladores APIC brindan características similares a las del modelo de prioridades de PARTEMOS, por ejemplo las prioridades de las IRQ no dependen de su número de línea física, y pueden ser cambiadas de forma independiente. Cada LAPIC (puede haber mas de uno si el sistema es multiprocesador) tiene un nivel de prioridad (TSKPRI) que corresponde a la prioridad de la tarea en ejecución dentro del núcleo de CPU asociado. Todas las interrupciones con prioridad menor o igual a TSKPRI quedarán bloqueadas.

Sin embargo existen diferencias importantes entre los modelos de prioridad del APIC y el INTHAL que dificultan el uso del APIC en PARTEMOS. La prioridad de cada interrupción está determinada por su vector de interrupción asociado, específicamente por los 4 bits más significativos del número de vector, lo que nos limita a 16 valores de prioridad, a diferencia de los 256 valores soportados por PARTEMOS.

El APIC no permite usar los números del 0 al 15 como vectores de interrupción, y peor aún, los primeros 32 vectores de interrupción están reservados para excepciones de la CPU (aunque el APIC no trata los vectores 16-31 como ilegales). Por tanto sólo podemos asignar a cada IRQ vectores entre 32 y 255, lo que nos deja con sólo 14 prioridades de interrupción diferentes. A cada bloque de 16 vectores de interrupción con la misma prioridad se le denomina **clase de interrupción**.

Como se ha visto no hay una traducción directa entre el modelo de prioridades de PARTEMOS y el brindado por el APIC. Para implementar el modelo de PARTEMOS sobre el APIC hemos identificado dos alternativas:

1. Utilizar el mecanismo de prioridades brindado por el APIC, convirtiendo las prioridades de PARTEMOS en números de vector y valores de prioridad del APIC. Esta traducción no es directa y requiere algún nivel de cómputo.

2. Desactivar el mecanismo de prioridades del APIC (mediante el envío de EOIs), y usar las máscaras de interrupción para forzar las prioridades de interrupción por software. Esta alternativa es similar a la implementación sobre el 8259, con la diferencia de que en el APIC cada IRQ debe ser activada o desactivada individualmente. No existe una máscara general que controle todas las interrupciones.

Para facilitar la implementación se establecerán las siguientes restricciones:

- No se hará detección de controladores APIC durante la inicialización del INTHAL, por lo que debe asegurarse que la PC sobre la cual se ejecute está versión del INTHAL **disponga de un IOAPIC y un APIC local** en el chip del procesador[1].
- Aunque en teoría en un sistema pueden existir mas de un IOAPIC que manejan diferentes conjuntos de IRQs globales, asumiremos la configuración típica de una **PC con un solo IOAPIC**, que maneja todas las IRQs globales (un controlador IOAPIC típico como el 82093AA [Intel 1996] puede manejar hasta 24 fuentes de interrupción).
- Asumimos que **las direcciones de memoria para acceso a los APICs son las direcciones estándares**, y que han sido configuradas previamente como fuertemente no cacheables (posiblemente por el BIOS).
- **No se hará manejo de errores en LAPIC**. Se asumirá que siempre se usarán vectores y registros legales (lo cual debería suceder siempre, a menos que exista algún error de programación), y que los mensajes provenientes del IOAPIC siempre serán correctos. La interrupción de error quedará enmascarada en la entrada correspondiente de la LVT.
- **Se permitirán identificar 25 IRQs diferentes**, numeradas del 0 al 24, las IRQs 0-23 serán manejadas por el IOAPIC, mientras el número de IRQ 24 lo usaremos para identificar la interrupción del temporizador del APIC local (LAPIC). Como estamos asumiendo un sistema monoprocesador sólo habrá una de estas interrupciones locales.
- **No se hará detección de la configuración de las líneas IRQ conectadas al IOAPIC**, típicamente las líneas IRQ conectadas a los PIC 8259 se conectan en el mismo orden al IOAPIC. Una excepción notable a la regla anterior es la línea de IRQ0 (temporizador 8254), que típicamente se conecta a la línea 2 del IOAPIC (y por tanto el software la ve como IRQ2). La constante **`TIMERIRQ`** definida en la interfaz del INTHAL identifica la IRQ del reloj del sistema, para su uso dentro del núcleo.

A continuación se describen diferentes propuestas de implementación del INTHAL sobre controladores APIC, basadas en las alternativas de implementación mencionadas previamente. Debemos mencionar que el objetivo principal buscado con estas implementaciones no fue lograr un HAL de interrupciones más eficiente, sino que sirvieron de ejercicio para practicar la portabilidad del mismo, y evaluar la factibilidad de uso de los APIC como controladores de interrupción en PARTEMOS.

Alternativa 1: Uso del mecanismo de prioridades del LAPIC

A continuación describimos un intento de implementación de la primera alternativa del INTHAL sobre controladores APIC. Como se explica al final de esta sección este intento de implementación fracasó;

[1] Algunas pruebas creadas para APIC (como "`IOrecTst.c`") permiten comprobar si existen o no controladores APIC, ya que de no existir muestran información de estado del APIC con valores no válidos.

pero como veremos más adelante, la idea planteada sirve de base a otras propuestas de implementación que intentan corregir los problemas que se presentaron.

La idea de esta propuesta es mantener un orden en los vectores de interrupción que coincida con el orden de prioridades asignadas a las diferentes IRQs en PARTEMOS, y fijar un nivel de prioridad de tarea (TSKPRI) en el LAPIC de forma que habilite y deshabilite las IRQ tal y como lo esperaría el sistema (ver figura 30). Se impusieron las siguientes restricciones al diseño:

- A pesar de poderse identificar 25 IRQs diferentes, sólo podrán estar activas (con prioridad mayor que cero) al mismo tiempo 14 de ellas. Esta restricción se debe a que sólo tenemos 14 niveles de prioridad disponibles, y facilita la programación del INTHAL.
- Cada nivel de interrupción del APIC sólo podrá tener como máximo una IRQ asociada. Con esta restricción se impide que dos IRQs tengan la misma prioridad en el LAPIC, lo que facilita algunos algoritmos del INTHAL. La excepción de esta regla sería el nivel de interrupción 1 del APIC, que tendrá asociadas todas las IRQs inactivas (con prioridad 0 en PARTEMOS), como explica mas adelante.
- Cada IRQ activa (con prioridad > 0) tendrá asignado el primer vector de interrupción dentro de alguna de las 14 clases de interrupción disponibles. Esto significa que el vector de interrupción asociado a cada IRQ tendrá un valor en hexadecimal de la forma “X0”, donde “X” representa la prioridad de la IRQ en el APIC.
- Las IRQs inactivas (con prioridad 0), tendrán asignado el primer vector de la clase 1 (o sea, el vector 10H). Aunque este vector está reservado para excepciones de CPU (falla de coprocesador) nunca habrá confusión entre la excepción y las IRQs debido a que el nivel prioridad en el LAPIC (registro TSKPRI) siempre será mayor o igual a 1, con lo que evitaremos la ocurrencia de las IRQs inactivas.

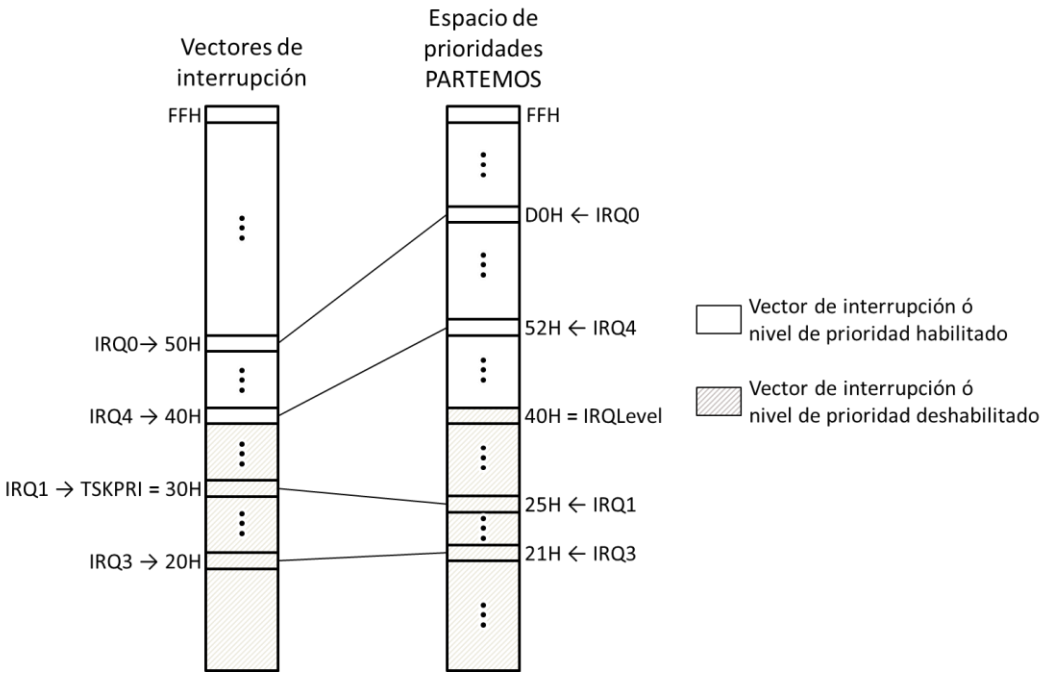


Figura 30: Ejemplo de asociación entre prioridades establecidas en el INTHAL y en el IOAPIC.

La figura 30 muestra con un ejemplo como esta propuesta implementación del INTHAL mantiene una asociación entre las prioridades de IRQ en PARTEMOS y los vectores de interrupción. Se puede notar como las prioridades de INTHAL para las IRQs activas (entre 1 y 255) coincide con el orden de sus vectores, pero no necesariamente con su valor. En la figura "IRQLevel" representa nivel de interrupción en el INTHAL, el cual tampoco coincide con TSKPRI pero ambos enmascaran las mismas IRQs en sus respectivos espacios. Para fijar el nivel de interrupción, la rutina **setIRQLevel** debe realizar una búsqueda del valor de TSKPRI a establecer en el LAPIC, de forma que se habiliten y deshabiliten las IRQs deseadas. Cambiar la prioridad de una IRQ (mediante la rutina **setIRQPriority**) puede implicar **cambios en el vector de interrupción de la IRQ afectada y de otras IRQs**.

La rutina de atención a interrupción dentro del INTHAL podría enviar el comando EOI al LAPIC y fijar el nivel TSKPRI correcto antes de invocar al manejador del núcleo. Alternativamente, el INTHAL podría mantener un registro de todas las interrupciones en servicio, y enviar comandos EOI sólo cuando sea necesario bajar la **prioridad de procesador** (ver sección 3.2.3) para habilitar IRQs.

Como se mencionó previamente, **este intento de implementación del INTHAL fracasó**, precisamente debido a que se basa en el cambio de vectores de interrupción en el IOAPIC para cambiar su prioridad, y el uso de la prioridad de tarea en el LAPIC (TSKPRI) para enmascararlas. El IOAPIC despacha las interrupciones apenas las detecta, no importa si están o no enmascaradas en el LAPIC. Un cambio de prioridad de una interrupción puede causar el cambio de su vector, pero las IRQs ocurridas previamente seguirán registradas en el LAPIC con el vector anterior.

Para ejemplificar este problema, supóngase que tenemos capturada una IRQ con vector de interrupción 20H, y la prioridad de tarea en el LAPIC es 4 (TSKPRI es 40H), y por tanto la IRQ está enmascarada, como muestra la figura 31 a), si en ese momento ocurre la interrupción en cuestión esta quedará registrada en el registro IRR del LAPIC con el vector 20H. Supóngase que luego queremos elevar la prioridad de la IRQ y fijamos su vector a 60H (ver figura 31 b), ahora la IRQ está habilitada para interrumpir al núcleo, pero la interrupción anterior no ocurrirá porque quedó registrada con un vector inferior.

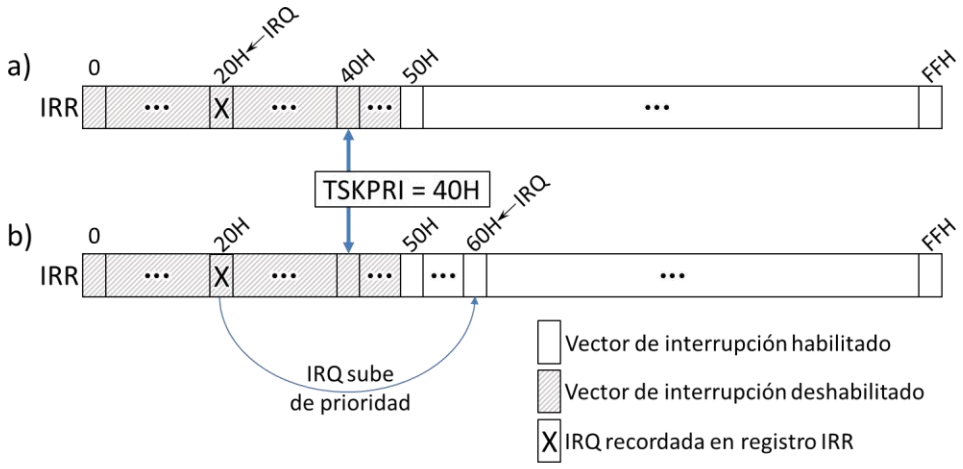


Figura 31: Problema del uso del mecanismo de prioridades del LAPIC.

El problema descrito sólo puede suceder cuando se realizan cambios en los valores de los vectores de interrupción (y por tanto en su prioridad). Por tanto esta solución sería útil en sistemas que capturen

todas las IRQs al inicio y nunca cambien su prioridad, pero no se ajusta al modelo proporcionado por el módulo INTHAL, el cual permite modificar la prioridad de una interrupción en cualquier momento.

Se podría pensar en simular por software las interrupciones previamente registradas, pero esto conduce a más problemas, ya que no hay manera de limpiar un bit en el registro IRR del LAPIC sin dejar que la interrupción ocurra físicamente. Más adelante se describe una variante de esta implementación que podría solucionar el problema planteado.

La propuesta de INTHAL mostrada en esta sección fue implementada parcialmente, pero su implementación fue descontinuada cuando se descubrió el problema descrito. No obstante, esta implementación parcial sirvió para refactorizar secciones de código, mejorándose la implementación del INTHAL ya existente, y posiblemente haciendo más fácil la implementación de otras versiones del INTHAL. El código implementado quedó disponible para su revisión o para usarlo de base en otras implementaciones.

Alternativa 2: Desactivación del mecanismo de prioridades del LAPIC

Esta alternativa de implementación del INTHAL sobre APIC propone el envío de comandos EOI inmediatos para desactivar el mecanismo de prioridades del LAPIC, y el uso de enmascaramiento de interrupciones para forzar las prioridades de las mismas. Las IRQs se enmascaran fijando un bit específico dentro de su entrada correspondiente de la tabla IOREDTBL ó LVT (dependiendo de si proviene de una fuente de interrupción externa o local).

Como no existe un registro que permita enmascarar todas las interrupciones simultáneamente (a diferencia del PIC8259 que si lo brinda), la implementación del modelo del INTHAL con enmascaramiento de interrupciones físico (no virtual) sería demasiado costosa, ya que un cambio de nivel de prioridad podría implicar muchos accesos al dispositivo controlador externo (IOAPIC), para enmascarar y desenmascarar IRQs.

Otra razón en contra de la implementación del INTHAL con enmascaramiento físico es que la desincronización entre el IOAPIC y el APIC local puede permitir la ocurrencia de interrupciones indebidas. Si el IOAPIC notifica una interrupción al APIC local, y antes de ser despachada al núcleo dicha interrupción es enmascarada en el IOAPIC (porque bajó de prioridad o subió el nivel de prioridad del sistema), de todas formas la interrupción será enviada al núcleo por el LAPIC. En este caso el núcleo debe tratar la interrupción como una interrupción indebida del modelo con enmascaramiento virtual.

El problema detectado en esta propuesta de implementación es que el IOAPIC (a diferencia de otros controladores de interrupción) ignora todas las líneas de interrupción activadas por flanco que estén enmascaradas, sin recordar la ocurrencia de ninguna petición IRQ en estas líneas.

Una solución al problema anterior es nunca enmascarar las IRQs. Un hecho a favor de esta solución consiste en que la mayoría de los dispositivos de interrupción no vuelven a generar más interrupciones hasta que son atendidos, haciendo innecesario (y más costoso) su enmascaramiento.

Las líneas de IRQ activadas por nivel no padecen del problema descrito para las líneas activadas por flanco, ya que aunque el IOAPIC no mantiene un registro de las IRQs ocurridas, la condición que provoca la interrupción (línea en alto) se mantiene hasta que se le de tratamiento a la misma. Si una línea de IRQ

activada por nivel se encuentra enmascarada, la interrupción se notificará a la CPU apenas se desenmascare la misma.

En el caso de las interrupciones activadas por nivel si es obligatorio el enmascaramiento de la misma, ya que como el manejador de interrupciones dentro del INTHAL la reconoce (envía EOI) apenas ocurre, si no se enmascara la interrupción ocurriría infinitamente. Más adelante en la sección 5.3.5 se realiza un análisis del tratamiento de interrupciones activadas por nivel.

Nota: se podría pensar en no enviar EOI inmediatamente como solución al problema antes mencionado, de forma que la interrupción quede bloqueada en el LAPIC (y usar EOI cuando desee desbloquearse), pero esto implicaría que las interrupciones deberían priorizarse cambiando su vector como se describe en la primera alternativa de implementación, con los mismos problemas que ya se describieron.

Variante de la alternativa 1: uso del registro de petición de interrupciones

Esta variante se diseñó como solución a los problemas presentados en la primera alternativa de implementación del INTHAL. Esta no fue implementada por su complejidad y por el posible costo excesivo que implicaría la misma. Con la descripción de esta alternativa sólo nos proponemos mostrar que se puede implementar un INTHAL sobre APIC que se comporte según la especificación sin usar enmascaramiento de interrupciones virtual.

Esta propuesta de implementación es similar a la descrita en la primera alternativa, por lo que sólo se señalarán las diferencias con esta.

- Sólo se utilizarán las clases de prioridad entre 2 y 15. La clase 1 no se utilizará ni siquiera para las IRQs inactivas. Esto se debe a que en esta propuesta se permite la ocurrencia de las interrupciones ocurridas en su vector original, pero la clase 1 está reservada para excepciones de CPU.
- Cada IRQ tendrá reservado un vector dentro de cada clase de prioridad. Esto implica que se deberían capturar 196 vectores de interrupción (14 niveles de prioridad X 14 posibles IRQs), con número de vector mayor a 20H. Una primera implementación podría usar una asignación directa (o sea, IRQ0 asociada a vectores de la forma X0H, IRQ1 con los vectores X1H, etc), posteriormente podrían utilizarse tablas para mantener la asociación con cualquier IRQ.

  Aunque ahora cada clase tendrá 14 vectores reservados (uno por IRQ), se mantiene el hecho de que no pueden existir dos IRQs cuyos vectores asignados estén dentro de la misma clase. Dicho de otra forma, dos IRQs no pueden tener la misma prioridad en el APIC.
- Se mantendrá una copia en memoria del registro IRR del LAPIC, aunque la estructura de datos usada no necesariamente tiene que ser un mapa de bits. Esta copia, que llamaremos IRRC, no siempre será igual al IRR físico, pero debe mantener los valores de IRR para las IRQs problemáticas, es decir aquellas que ocurrieron, fueron recordadas en el IRR y luego se movieron de vector.

  Nuestra estructura de datos permitirá mantener 4 estados para cada vector de interrupción:

  - 00B: limpio (no está en IRRC, ni esta pendiente de notificarse su ocurrencia al núcleo)
  - 01B: En IRR (Esta registrado en el IRR, pero su IRQ asociada ya fue notificada al núcleo)
  - 10B: Pendiente limpio (no está en IRRC, pero debe notificarse al núcleo la ocurrencia de la IRQ asociada)

- 11B: Pendiente (Esta registrado en el IRR y debe notificarse su ocurrencia al núcleo)

- Cada vez que una IRQ sea movida de vector, posteriormente debe leerse la porción del IRR asociada al vector previo de la IRQ[1], a menos que este vector ya este registrado en IRRC. Si la IRQ se encuentra registrada en IRR, el vector previo al movimiento será marcado en estado pendiente (11B). Opcionalmente se puede aprovechar la lectura al IRR para hacer la misma operación con otro vector, aun cuando no vaya a ser movido.
- Cada vez que ocurra una IRQ con número de vector V, el manejador del HAL enviará un EOI al LAPIC, y limpiará el vector V en IRRC (el estado pasa de xyB a x0B). Alternativamente el INTHAL podría no enviar el EOI inmediatamente, y mantener un registro de las interrupciones en servicio para controlar la prioridad de procesador en el LAPIC.
- Cuando ocurra una IRQ X asociada al vector V con estado diferente de 01B, no necesariamente se notificará esta IRQ al núcleo. Si algún vector W pendiente tiene asociada una IRQ Y con mas prioridad que IRQX, se notificará en su lugar la IRQ Y. El vector V se marcará como pendiente limpio (estado 10b), y el vector W se pasará a un estado no pendiente (pasa de 1xB a 0xB). Si no existe el vector W antes mencionado, entonces se notificara la IRQ X ocurrida, y si su vector V tiene estado pendiente (11B, no 10B) se cambiará a limpio (00B).
- Si el vector de una interrupción ocurrida tiene estado 01B nunca se notificará al núcleo su IRQX asociada, aunque no haya IRQ pendiente de más prioridad (porque ya fue notificada previamente). En su lugar se buscará un vector W pendiente asociado a una IRQY de la mayor prioridad posible, y se notificará la IRQY al núcleo, cambiando el estado del vector W como se hizo en el caso anterior (de 1xB a 0xB).

  El presente diseño garantiza que siempre exista el vector W, y que la IRQY tenga prioridad para ocurrir.
- Si existe al menos una interrupción pendiente (estado 1xB) con prioridad suficiente para ocurrir, el nivel de prioridad del LAPIC se fijará de forma que al menos un vector en IRRC quede habilitado; así garantizamos que ocurra una nueva interrupción que el INTHAL podrá utilizar para notificar al núcleo la interrupción pendiente y repetir el proceso si es necesario. Si no existen interrupciones pendientes o no tienen suficiente prioridad, el nivel de prioridad del LAPIC se fijará de la forma descrita en para la primera alternativa de implementación[2].

En raros casos puede suceder que una IRQ se active varias veces, pero que sólo se recuerde una única ocurrencia. Esto sucede si una IRQ con vector en estado pendiente limpio (10B) ocurre nuevamente y existe una IRQ pendiente con más prioridad que ella. Esta pérdida de notificaciones no debería ser un problema ya que sólo puede ser causada por fuentes de interrupción “estresantes”, que no esperan a ser

[1] El registro IRR del LAPIC tiene un tamaño de 256 bits (un bit por vector). Para brindar acceso a todos sus bits, el IRR se divide en 8 registros de 32 bits, accesibles de forma independiente.

[2] Existe la posibilidad (muy rara) de que una IRQ no tenga suficiente prioridad pero haya sido recordada en un vector suficientemente grande para ocurrir. En este caso posiblemente la mejor solución es permitir que ocurra la interrupción y tratarla como interrupción indebida. Esta condición solo puede suceder si la IRQ tenía una prioridad alta, que luego es reducida sin esperar a su ocurrencia.

atendidas antes de generar la siguiente IRQ. En ocasiones este tipo de interrupciones también son ignoradas por el hardware controlador de interrupciones.

Las interrupciones activadas por nivel deberían funcionar bien con el esquema propuesto, siempre que cumplan con las restricciones que se mencionan en la sección 5.3.5. En este caso las pérdidas de notificaciones mencionadas en el párrafo anterior son un comportamiento deseado que impide notificar mas de una vez (o sea incorrectamente) la misma IRQ.

La técnica propuesta en esta sección permite implementar sobre APIC el modelo clásico de interrupciones del INTHAL, que no permite la notificación de interrupciones indebidas al núcleo. Aunque el esquema propuesto permite la ocurrencia de IRQs que teóricamente no deberían ocurrir de acuerdo con su prioridad, estas sólo ocurren en "reemplazo" de una IRQ de más prioridad recordada, que será notificada al núcleo en su lugar, por lo que no existe sobrecarga adicional por concepto de interrupciones indebidas.

Se podría implementar una variante del esquema propuesto que permita la ocurrencia de interrupciones indebidas, con un posible ahorro en operaciones de acceso al APIC, como subir el nivel de prioridad físico en el controlador.

El esquema propuesto es costoso debido al número de acceso a los dispositivos APIC que implica. Los cambios de vectores de interrupción (causados por cambios de prioridad de IRQ) no sólo implican accesos al IOAPIC como en el primer intento, sino también accesos al IRR del LAPIC para verificar si la interrupción modificada ocurrió. Cualquier intento de implementación de esta técnica debería ser muy cuidadoso en los algoritmos de asignación de vectores utilizados, buscando minimizar el cambio de los mismos.

La implementación de esta técnica también es más compleja desde el punto de vista de las estructuras de datos y los algoritmos necesarios. Se sugiere mantener dos arreglos binarios que indiquen si los vectores están en IRRC y pendientes, y además mantener contadores que almacenen el número de interrupciones pendientes para cada IRQ.

Conclusión sobre el uso de APICs para implementación del INTHAL

Las diferencias entre el esquema de manejo de interrupciones de los controladores APIC y el modelo de prioridades de PARTEMOS dificultan el uso de estos controladores para la implementación del INTHAL de PARTEMOS. La separación existente entre el elemento de hardware que recibe y enmascara las líneas de IRQ y el que les da prioridad, unido al hecho de que para cambiar las prioridades de las IRQ sea necesario cambiar su número de vector, impiden el uso directo de los APIC para implementar la funcionalidad del INTHAL. La incapacidad de los controladores APIC de recordar las interrupciones ocurridas cuando una línea está enmascarada impide el uso de enmascaramiento para simular el mecanismo de prioridades de PARTEMOS.

La alternativa implementada para utilizar los APIC como controladores de interrupción de PARTEMOS, consiste en utilizar una variante de la alternativa 2, que no enmascara las interrupciones. En este caso el sistema queda susceptible a interrupciones de baja prioridad, aunque mantiene ciertas ventajas sobre el modelo tradicional de manejo de interrupciones. Ante una interrupción, PARTEMOS ejecuta una

pequeña sección de código, cuyo objetivo final es poner lista una tarea que atenderá la interrupción. Una tarea de alta prioridad en ejecución sólo será interrumpida por la pequeña sección de código, no por la tarea (de menor prioridad) que da servicio a la interrupción. De esta forma se minimiza la interferencia de las IRQs de baja prioridad, siendo este modelo similar al utilizado por sistemas de tiempo real como LynxOS.

Debemos señalar que lo mencionado anteriormente aplica al caso frecuente en que las IRQs son activadas por flanco. Si una IRQ es activada por nivel debe enmascararse apenas ocurra, para evitar que ocurra infinitamente, como se explica en la sección 5.3.5.

Notas de implementación

Con la implementación de versiones del INTHAL sobre controladores APIC, se hizo necesario realizar algunas modificaciones al archivo de interfaz "`iINTHAL.h`". Siempre que se desee utilizar una versión del INTHAL que utilice controladores APIC, se debe definir el símbolo **`USE_APIC`** al inicio de "`iINTHAL.h`", el que será utilizado posteriormente para realizar compilación condicional. El uso de controladores APIC permite un mayor número de fuentes de interrupción (25 IRQs en nuestro caso), por lo que el archivo "`iINTHAL.h`" ahora define la constante **`IRQNUMBERS`** y nuevas constantes de la forma **`IRQx`** basándose en la existencia o no del símbolo **`USE_APIC`**. Este archivo también define condicionalmente el símbolo **`TIMERIRQ`**, que identifica la IRQ asociada a la interrupción de reloj.

Con la posibilidad de compilar el núcleo PARTEMOS para PC utilizando diferentes controladores de interrupción, y por tanto diferentes versiones del INTHAL, también existe la posibilidad de compilar erróneamente una versión del INTHAL para 8259 con el símbolo **`USE_APIC`** definido, o una versión para APIC sin este símbolo. Para detectar este tipo de errores rápidamente, al código C de las versiones del INTHAL se le introdujo una verificación de existencia (o ausencia) del símbolo **`USE_APIC`**. Por ejemplo las implementaciones sobre APIC utilizan una verificación como la siguiente:

```
#ifndef USE_APIC
#error "El simbolo USE_APIC debe definirse para compilar INTHAL APIC"
#endif
```

Cualquier implementación del INTHAL sobre APIC debería hacer uso del módulo "HPC/IA32/APIC.C", y su archivo de interfaz "APIC.H". Estos archivos definen varias funciones y macros que brindan acceso a los controladores APIC.

Sólo la alternativa 2 sin enmascaramiento de interrupciones del INTHAL sobre APIC fue implementada completamente, comprobada e integrada al núcleo de PARTEMOS. Al igual que la versión del INTHAL sobre 8259, esta implementación está dividida en dos archivos: una porción en ensamblador ("`HAL4IA32/irqAPICm.asm`") que contiene los puntos de entrada de todas las IRQs (ahora 25), y un archivo C ("`HAL4IA32/APICHalm.c`") que contiene el resto del INTHAL. Esta implementación del INTHAL no realiza enmascaramiento de interrupciones, ni siquiera de interrupciones activadas por nivel (las cuales no son soportadas), por lo que la implementación de los servicios del INTHAL es muy sencilla. Como se explicó en la sección anterior, esta implementación a pesar de ser funcional elimina ciertas ventajas del modelo de prioridades de PARTEMOS.

5.3.5 Soporte de interrupciones activadas por nivel

Como se mencionó en la sección 3.2.1, las líneas interrupción activadas por nivel se mantienen en nivel alto (1 lógico) mientras el software no elimine la condición que la provocó (es decir, que atienda al dispositivo que causó la interrupción). Para evitar que este tipo de interrupciones ocurra infinitamente, sólo se le debe enviar reconocimiento (EOI) al controlador de interrupciones una vez que haya sido atendida o, alternativamente, debería enmascararse. Las implementaciones realizadas del INTHAL envían el comando EOI apenas ocurre una interrupción para anular el mecanismo de prioridades del hardware, lo cual puede causar problemas en algunas circunstancias.

El HAL de interrupciones en modo de trabajo virtual no enmascara una fuente de interrupción si esta ocurre debidamente (o sea, si tiene suficiente prioridad para ocurrir). Como lo primero que hace el INTHAL (en todas sus versiones actuales) ante la ocurrencia de una interrupción es enviar el EOI al controlador, una interrupción activada por nivel que ocurra debidamente se va a generar nuevamente y será vista como una IRQ indebida, por lo que el INTHAL procederá a enmascararla físicamente. En otras palabras, una única ocurrencia de una interrupción activada por nivel puede interrumpir dos veces a la CPU. Este problema fue descrito en [Leyva-del-Foyo 2008] y la solución propuesta en ese mismo documento es siempre enmascarar la IRQ activada por nivel cuando esta ocurra (aun estando en modo de enmascaramiento virtual).

En el caso de la implementación del INTHAL sobre APIC sin enmascaramiento el problema antes descrito se manifiesta con peores consecuencias, ya que de ocurrir una IRQ activada por nivel esta se mantendrá ocurriendo de forma infinita (ya que se envía EOI pero nunca se enmascara). La solución en este caso también es enmascarar físicamente la línea de interrupción en el IOAPIC, y desenmascararla cuando el nivel de interrupción baje nuevamente (de forma que quede activa). Como se mencionó previamente, enmascarar una línea de IRQ activada por nivel no provoca los problemas descritos para las IRQs activadas por flanco.

Otro problema relacionado que no ha sido descrito previamente es que, sin importar el modo de trabajo del INTHAL, si la tarea que da servicio a una interrupción (IST) activada por nivel tiene menos prioridad que la interrupción, entonces esta ocurrirá de forma infinita. Esto se debe a que la IRQ queda habilitada antes de cederle el control al manejador, por lo que ocurre nuevamente antes de que se le pueda dar tratamiento. Normalmente las tareas manejadoras de interrupción tienen el mismo nivel de prioridad (o incluso mayor) que la IRQ que atienden, por lo que este problema no debería suceder. Pero PARTEMOS no obliga de ninguna forma a que esto sea así, quedando como responsabilidad del usuario el uso correcto de las prioridades de interrupción. Una posible solución a este problema es que PARTEMOS prohíba explícitamente que una tarea espere por una IRQ de más prioridad. Alternativamente, la interfaz del INTHAL podría brindar una forma de declarar una IRQ como “activada por nivel”, y proporcionar un servicio para que la IST notifique el fin del tratamiento (o sea, un servicio de EOI).

5.3.6 Pruebas al INTHAL

Las diferentes implementaciones del HAL de interrupciones fueron sometidas a varias pruebas de unidad, la mayoría de las cuales fueron variantes de pruebas ya existentes [Martínez-Cortés 2005]. Las

pruebas fueron modificadas para adaptarlas a los cambios de interfaz en el INTHAL, y además realizan como primera operación la invocación a la rutina **`initPlatform`**.

Algunas de las pruebas requieren que el software automáticamente espere por la ocurrencia de la interrupción de reloj, lo que se logra mediante lecturas del contador 0 del temporizador 8254. La mayor parte de las pruebas realizadas se pueden compilar tanto en 16 como en 32 bits, aunque también se realizaron pruebas específicas para versiones de 32 bits del INTHAL.

Inicialmente las pruebas realizadas utilizaban módulos de salida muy sencillo para mostrar sus resultados en pantalla. Posteriormente el administrador de ventanas de PARTEMOS fue modificado para hacerlo portable, y las diferentes pruebas al HAL fueron modificadas para utilizar este módulo (el cual no necesita de PARTEMOS para funcionar). En el próximo capítulo se dedica una sección al administrador de ventanas de PARTEMOS y otros métodos de salida.

5.4 Desarrollo HAL de excepciones

Las excepciones de CPU son una forma de notificación de condiciones de error detectadas mientras se ejecuta una instrucción. En la mayoría de las computadoras, el mecanismo usado por la CPU para notificar excepciones es el mismo que para las interrupciones de hardware. Por eso la implementación del HAL de excepciones para la arquitectura IA-32 guarda muchas semejanzas con la implementación realizada para el INTHAL.

El HAL de excepciones (EXCHAL) para la versión de 16 bits de PARTEMOS está realizado completamente en ensamblador, y sólo captura y pasa al núcleo la excepción de división por cero, ignorando las restantes posibles excepciones. La versión de 32 bits del EXCHAL esta realizada casi por completo en lenguaje C, y se mantiene notificando al núcleo únicamente la excepción de división por cero. Adicionalmente, el EXCHAL de 32 bits captura las restantes excepciones en un manejador interno que imprime un error y aborta el sistema. Para el caso de la excepción de depuración, se usa otro manejador interno, que imprime información relativa a esta excepción antes de abortar el sistema. En el próximo capítulo se brinda más información sobre la depuración del núcleo.

La figura 32 muestra diferentes flujos de ejecución posibles para el manejo de excepciones. De forma similar al HAL de interrupciones, una pequeña porción del EXCHAL está contenida en un módulo en lenguaje ensamblador (“`HAL4IA32/ExcHal0.asm`”), el cual contiene los puntos de entrada de todas las posibles excepciones (vectores 0-31). Al igual que se hace dentro del INTHAL, el código ensamblador del EXCHAL salva el contexto interrumpido en una estructura tipo **`IntContext`** (que contiene registros salvados, el número de la excepción ocurrida y el parámetro de excepción si aplica), cuya dirección es pasada como argumento al manejador de interrupciones general **`excEntryC`**, contenido dentro de la porción C del EXCHAL (archivo “`HAL4IA32/ExcHal.c`”).

El EXCHAL permite la instalación de manejadores internos para excepciones específicas, lo que realiza invocando a la rutina interna “**`excInstallHandler`**”, durante su inicialización. Actualmente sólo se instalan manejadores específicos para las excepciones 0 (división por cero) y 1 (excepción de depuración), aunque es muy fácil extender el EXCHAL para manejar otras excepciones. El código de **`excEntryC`** verifica si la excepción ocurrida tiene instalado un manejador interno específico, y de existir este lo invoca. En caso contrario imprime un mensaje de error por defecto y aborta el sistema.

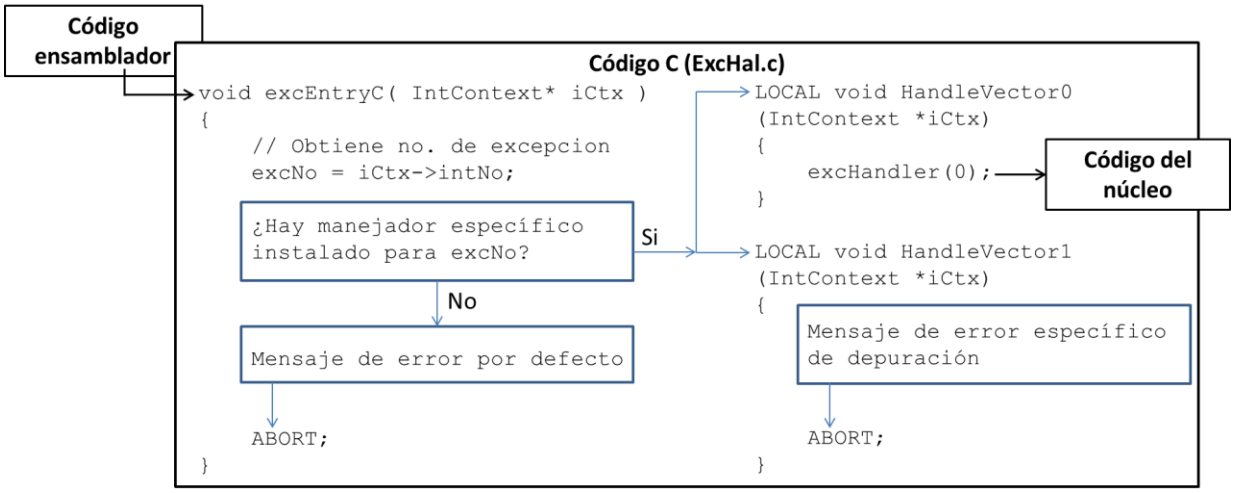


Figura 32: Secuencia de manejo de excepciones.

5.5 Desarrollo del HAL de CPU

El HAL de CPU (CPUHAL) contiene fundamentalmente rutinas para trabajar con contextos de CPU, que permiten salvar y restaurar estados de ejecución del procesador.

Un servicio esencial del CPUHAL es la conmutación de contexto, utilizada por el núcleo para cambiar la tarea actual en ejecución (conmutación de tarea). Es deseable que la conmutación de contexto sea una operación muy rápida, por lo que una parte de este trabajo estuvo enfocada a mejorar la misma, y no simplemente portarla a la plataforma IA-32. En esta sección describimos los trabajos realizados sobre el CPUHAL, incluyendo la optimización de la conmutación de contexto, cambios en la interfaz del módulo, y su adaptación a la plataforma de 32 bits.

5.5.1 Análisis del modelo de conmutación de contexto previo

PARTEMOS16 brindaba una conmutación de contexto relativamente pesada, en la cual se salvan todos los registros de la CPU en una estructura denominada CPUCONTEXT, ubicada dentro de los bloques de control de cada tarea. La conmutación de contexto en PARTEMOS16 es realizada por la rutina **`contextSwitch`** ubicada en el HAL de CPU, esta rutina salva todos los registros en el contexto interrumpido y los restaura del nuevo contexto a ejecutar.

La conmutación de contexto es una operación programada dentro del núcleo de los SO, por lo que es necesario que el núcleo gane el control (se active) antes de que ocurra una conmutación de contexto[1]. Existen dos formas en que el núcleo de un sistema operativo (incluyendo PARTEMOS) puede activarse: por medio de una interrupción (forma asíncrona) y por medio de llamadas al sistema (forma síncrona). Como PARTEMOS es un sistema con expropiación, cualquiera de las dos formas de activación puede provocar una conmutación de contexto.

En el caso de una conmutación de contexto provocada por una interrupción se realiza dos veces la salva de todos los registros: una en pila por el manejador de interrupción, y otra en el bloque de control de la

[1] Con la conmutación de contexto por hardware existe la posibilidad de realizar conmutación directa sin involucrar al núcleo, pero esto no es normalmente utilizado por los sistemas operativos.

tarea interrumpida. A esta sobrecarga debemos sumar que se restauran todos los registros de la nueva tarea a ejecutar. Cuando se reanuda la ejecución de una tarea que fue pausada como consecuencia de una interrupción, se realiza dos veces la restauración de todos los registros: una dentro del núcleo (realizada por **`contextSwitch`**), y otra por el código de salida del manejador de interrupción (restauración desde la pila).

Para la conmutación provocada por una llamada a un servicio del sistema (forma síncrona) sólo se salva el contexto interrumpido una sola vez, lo que es nuevamente realizado por código del núcleo. Actualmente el código de usuario de PARTEMOS se enlaza en tiempo de compilación junto con el núcleo, por lo que este no espera que las rutinas del sistema salven más registros que los especificados por la convención de llamada del compilador C, lo cual es garantizado por el propio compilador. Sin embargo la conmutación de contexto de PARTEMOS16 salva (innecesariamente) todos registros de la CPU, por lo que se decidió implementar un nuevo tipo de conmutación de contexto más eficiente.

5.5.2 Nueva conmutación de contexto ligera

El núcleo PARTEMOS realiza las conmutaciones de contexto invocando a la rutina **`contextSwitch`**, y la invocación a esta rutina la realiza código implementado en C, que sólo requiere que **`contextSwitch`** retorne dejando intactos los registros especificados por la convención de llamada del compilador C[1]. En base a esto se implementó una nueva forma de conmutación de contexto, denominada "**conmutación de contexto ligera**", la cual se limita a salvar los registros especificados por la convención de llamada de C.

Con la conmutación de contexto ligera sólo se salvan los registros mínimos necesarios, o sea aquellos que el compilador asume que no son modificados al invocarse una función. Para la versión de PARTEMOS de 16 bits (compilada con Borland/Turbo C) se salvan los registros BP, SI y DI. Los registros DS y SS también son de tipo no-tocables para el compilador, sin embargo estos no se salvan porque se supone que son iguales para todas las tareas y no serán modificados por el programador; es responsabilidad del programador que estos registros estén correctos cuando invocan servicios del sistema. Si el código de usuario es interrumpido por una IRQ, los puntos de entrada de las interrupciones corrigen el valor de DS, pero no el de SS, en otras palabras una tarea de usuario pudiera cambiar temporalmente el valor de DS, pero nunca debe tocar SS.

En el caso de la implementación de la conmutación de contexto ligera sobre 32 bits, la idea es similar a la de la versión de 16 bits, sólo que ahora debemos ajustarnos a la convención de llamada del compilador GCC, por lo que debemos salvar en la pila los registros EBX, ESI, EDI y EBP. Se garantiza que los registros de segmento DS, ES, y SS no serán modificados porque sólo existe un selector válido para estos en la GDT.

[1] Para mas información sobre las convenciones de llamada de C revisar:
http://alexfru.narod.ru/os/c16/c16.html
http://wiki.osdev.org/Calling_Conventions
http://www.agner.org/optimize/calling_conventions.pdf

Con la nueva conmutación, los registros salvados no se almacenan en estructuras de datos reservadas para cada tarea, sino en el tope de la pila interrumpida, junto con la dirección de ejecución interrumpida. Esto hace que la rutina de conmutación ahora sea muy sencilla y ligera.

La figura 33 muestra el proceso de conmutación de contexto ligera en su versión de 16 bits, desde una Tarea A en ejecución a una Tarea B. A la entrada de la rutina de conmutación, el tope de pila de la tarea A contiene la dirección de retorno de la rutina (representada por la frase "ret dir"), mientras que la pila de la tarea B contiene además de la dirección de retorno los registros salvados BP, SI, y DI. Lo primero (paso 1 en la figura) que realiza la rutina de conmutación es apilar (en la pila de la tarea A) los valores de los registros BP, SI, y DI, quedando los topes de pila de ambas tareas con la misma estructura. El siguiente paso (2 en la figura) es realizar una conmutación de pila, que consiste en asignarle al registro tope de pila SP el valor tope de pila de la tarea B, que estaba salvado en su bloque de control de tarea (TCB). Por supuesto antes de conmutar de pila se debe salvar el valor de SP como tope de pila en el TCB de la tarea A, para permitir reanudarla posteriormente. El tercer paso mostrado en la figura consiste en restaurar (desapilar) los registros DI, SI, y BP del tope de pila actual (ahora de la tarea B), lo que deja la pila en el mismo estado que tenía a la entrada de la rutina de conmutación, es decir con una dirección de retorno en el tope. Al retornarse de la rutina de conmutación de contexto la ejecución continuará donde la tarea B fue previamente interrumpida. La versión de 32 bits de la conmutación de contexto ligera es muy similar a la descrita, con la diferencia que salva los registros extendidos EBP, ESI, EDI, y EBX.

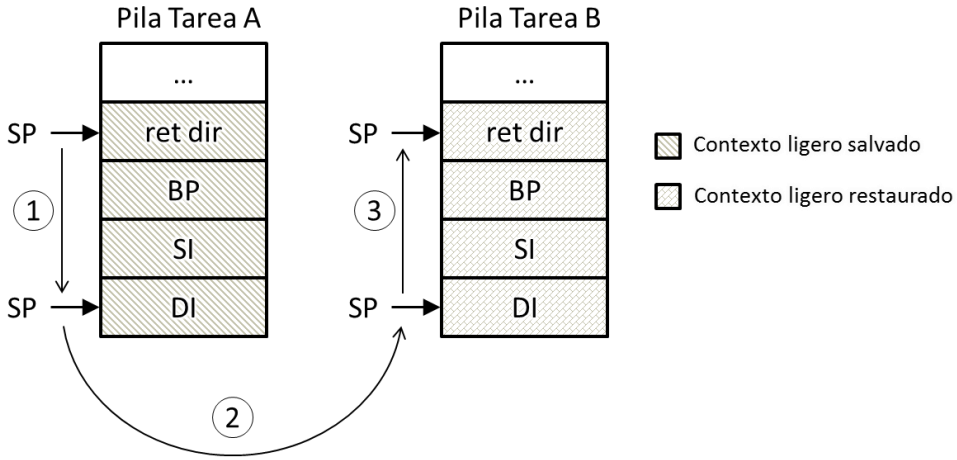


Figura 33: Conmutación de contexto ligera.

Toda tarea que no se encuentre en ejecución tendrá en el tope de su pila los valores salvados necesarios para reanudar su ejecución posteriormente. A estos valores salvados en el tope de pila los llamaremos en conjunto **contexto ligero**. En la figura 33 los segmentos sombreados representan contextos ligeros, donde el contexto de la tarea A fue creado en el proceso de conmutación descrito, mientras el de la tarea B ya existía y fue usado para restaurar su ejecución. Los bloques de control de cada tarea ahora almacenarán el valor del tope de la pila (evidentemente este valor es válido sólo si la tarea no se encuentra en ejecución), o lo que es lo mismo, un apuntador al contexto ligero de la tarea interrumpida.

Además de ser más rápida, la conmutación de contexto ligera tiene otras ventajas. Ahora los bloques de control de cada tarea no tienen que almacenar espacio para las estructuras de tipo CPUCONTEXT, y en su lugar almacenan un apuntador al tope de pila de la tarea. De esta forma se ahorra algo de memoria,

aunque a cuesta de una pequeña cantidad extra de pila (más pequeña de todas formas que la usada por CPUCONTEXT).

Otra ventaja de la nueva forma de conmutación de contexto es que el código C del núcleo se hace algo más portable. Ahora el HAL de CPU salva los registros necesarios en la pila, y no es necesario que el núcleo conozca la estructura o el tamaño del contexto ligero salvado. En otras palabras la estructura del contexto ligero queda oculta al núcleo.

5.5.3 Cambios al CPUHAL

Con la introducción de la conmutación de contexto ligera se eliminó el uso de estructuras de tipo **`CPUCONTEXT`** dentro de los bloques de control de tarea (TCB), ya que esta estructura no está involucrada en la conmutación de tarea. Sin embargo la declaración de la estructura **`CPUCONTEXT`** se mantuvo porque es utilizada en el mecanismo de excepciones de PARTEMOS. La conmutación de contexto ligera no fue el único cambio realizado al CPUHAL, también se introdujeron nuevas funciones con el objetivo de eliminar dependencias del hardware dentro del código del núcleo, y por tanto hacerlo mas portable.

Las rutinas **`initContext`**, **`contextSwitch`**, **`contextPush`**, y **`changeContextCP`** trabajaban con la estructura **`CPUCONTEXT`** y fueron eliminadas del CPUHAL ó del código del núcleo, debido a que ya no eran necesarias. De las funciones que operan sobre la estructura **`CPUCONTEXT`**, sólo las funciones **`getContext`** y **`jmpContext`** permanecieron definidas dentro del HAL de CPU, porque son usadas dentro el mecanismo de excepciones del núcleo.

Las funciones de trabajo sobre contextos ligeros **`lightContextSwitch`**, **`lightJmpContext`**, y **`lightInitContext`** fueron agregadas al CPUHAL, remplazando a sus equivalentes sin prefijo “light”.

La rutina **`initContext`** recibía apuntadores de tipo FAR (o sea, apuntadores formados por un segmento y un desplazamiento) para inicializar la pila y el punto de ejecución de un contexto, y otro parámetro con el valor del segmento de datos para el contexto. Para hacer la interfaz del CPUHAL compatible con el modelo de memoria plano, la nueva rutina **`lightInitContext`** no recibe apuntadores FAR, que fueron remplazados por apuntadores simples (o sea formados sólo por el desplazamiento). El parámetro usado para definir el segmento de datos se eliminó del prototipo de la función.

El núcleo PARTEMOS soporta el uso de una abstracción denominada “**procedimiento asíncrono**”. Un procedimiento asíncrono es una rutina que no es invocada desde el contexto que la ejecuta, sino que interrumpe asíncronamente la ejecución de este contexto, de forma similar a una interrupción. La interrupción del procedimiento asíncrono es simulada haciendo cambios en el contexto donde se debe ejecutar; este proceso lo denominamos “**inyección de procedimiento asíncrono**”, y es ejecutado desde otra tarea.

Para eliminar dependencias de la plataforma, existentes dentro del núcleo PARTEMOS y relacionadas con el uso de procedimientos asíncronos, se introdujo una nueva función dentro del CPUHAL, con el prototipo:

```
PUBLIC void InjectAsyncProcedure(
        LIGHTCONTEXT lctx,     /* apuntador al contexto     */
        void (*injectFunct)(),/* Punto de entrada          */
        WORD_ arg              /* Argumento a pasar         */
        )
```

Este servicio recibe un apuntador al contexto ligero (**lctx**) de una tarea interrumpida, y lo modifica de forma que al conmutarse hacia el mismo la ejecución de la tarea prosiga en la función **injectFunct**, a la cual se le colocan sus parámetros según el prototipo:

```
void asyncProcedure(WORD arg, WORD_ * interruptContext);
```

Donde **arg** es el mismo argumento que recibe **InjectAsyncProcedure**, y **interruptContext** es un apuntador a un contexto que la función invocada (procedimiento asíncrono) utilizará para restaurar el contexto original de la tarea (pasado en **lctx) .**

Auxiliándose de la función **InjectAsyncProcedure**, ahora el núcleo puede "inyectar" llamadas a procedimientos asíncronos a otros contextos, sin necesidad de conocer detalles como la forma de pasar parámetros a funciones, o la estructura interna de los contextos de CPU. En el próximo capítulo se muestra como esta función es usada dentro del núcleo.

Como los cambios descritos en esta sección afectan la interfaz entre el módulo CPUHAL y el núcleo del sistema, los mismos fueron realizados en tanto en la versión de 16 bits como en la versión de 32 bits del módulo CPUHAL.

5.5.4 Notas de implementación

El HAL de CPU (tanto en su versión de 16 bits como en la de 32 bits) fue implementado en dos archivos. Las rutinas **lightContextSwitch**, **lightJmpContext**, **getContext**, y **jmpContext** fueron implementadas en lenguaje ensamblador, debido a que acceden directamente a los registros de la CPU. Las restantes rutinas modifican contextos ligeros salvados en pila, por lo que fueron fácilmente implementadas en lenguaje C. Los archivos específicos usados en la capa HAL se listan mas adelante en la sección 5.7.

El código ensamblador para la versión de 32 bits del CPUHAL contiene varias diferencias notables con su equivalente de 16 bits. Todos los registros de la estructura **CPUCONTEXT** fueron extendidos a 32 bits (incluso los campos destinados a registros selectores de segmento, a pesar de que estos sólo son de 16 bits), y se agregaron dos campos para los registros GS y FS. Hay que notar que el ensamblador usado (NASM) no soporta el concepto de estructuras de la forma en que lo hacen los ensambladores típicos de 16 bits (como MASM o TASM), por lo que la declaración y uso de la estructura también cambió. Los parámetros en la pila de las diferentes funciones del HAL de CPU no se encuentran en la misma posición de antes, debido a su aumento de tamaño. Otras diferencias entre el código ensamblador de 16 y 32 bits vienen dadas por restricciones impuestas por el propio ensamblador NASM.

5.5.5 Pruebas al CPUHAL

La prueba de unidad existente previamente para el CPUHAL estaba diseñada para trabajar con las rutinas anteriores, que usaban estructuras de tipo **CPUCONTEXT** y fueron eliminadas en su mayoría. Como

algunas de estas rutinas (**`getContext`** y **`jmpContext`**) todavía son utilizadas dentro del mecanismo de excepciones del núcleo, la prueba existente fue simplificada para comprobarlas únicamente a ellas. El archivo script "`tstcpuhw.sh`" ("`TSTCPUHW.PRJ`" para 16 bits) puede ser utilizado para compilar esta prueba. Adicionalmente, la prueba del mecanismo de excepciones del núcleo (script "kexctst1.sh") también hace uso de estas rutinas, comprobándolas de paso.

Para las rutinas de trabajo con contextos ligeros en pila se creó una nueva prueba, compilable con el script "`tstcpuh.sh`" ("`TSTCPUH.PRJ`" en 16 bits). La figura 34 muestra el flujo de ejecución de la prueba en seudocódigo, cuyo código C está disponible en el archivo "`SRC/HAL4PC/test/CPUHTEST.c`". La idea de la prueba consiste en utilizar una variable entera global "**`estado`**", la cual es incrementada y verificada a medida que el flujo de ejecución se mueve entre las diferentes zonas del código. Para mayor claridad, el seudocódigo no muestra las acciones tomadas ante un error, en su lugar, se muestran entre corchetes las condiciones verificadas en las pruebas. Por ejemplo, la línea

```
[estado==0]
```

indica que la prueba verifica que el valor de la variable **`estado`** sea cero, mostrando un error en caso contrario.

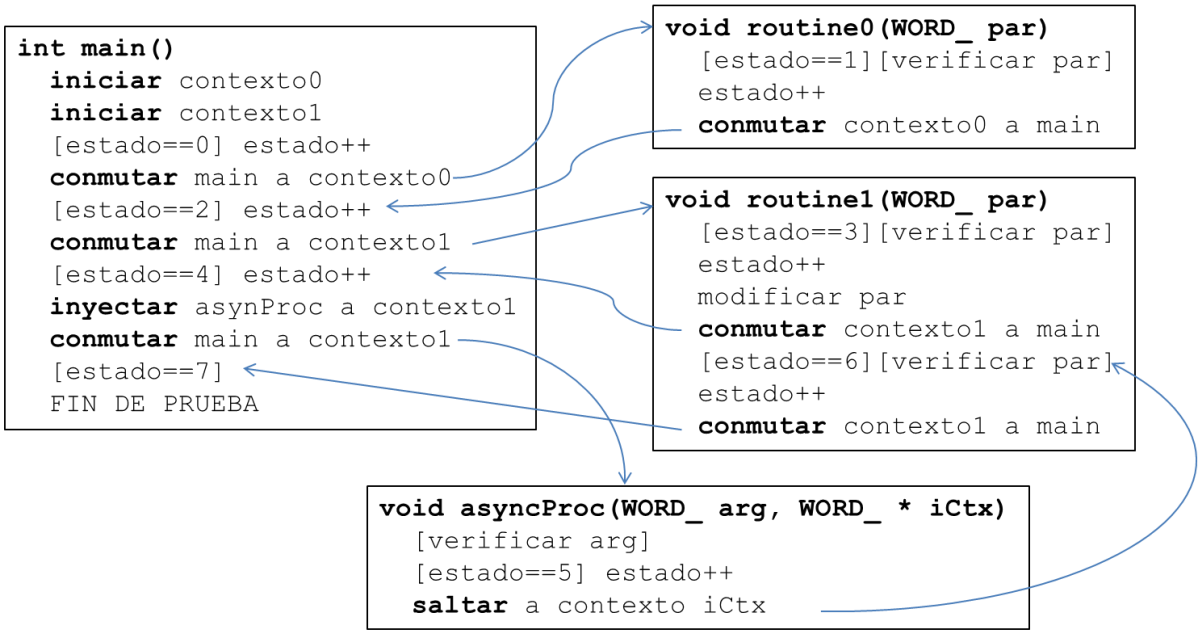


Figura 34: Prueba a rutinas del CPUHAL.

Aunque para mantener la claridad de la figura no se muestra el detalle de la invocación de las diferentes rutinas del CPUHAL, cada palabra escrita en **negrita** representa una invocación alguna de estas rutinas. La palabra "**`iniciar`**" representa una invocación a **`lightInitContext`**, "**`conmutar`**" a **`lightContextSwitch`**, "**`inyectar`**" a **`InjectAsyncProcedure`**, y "**`saltar`**" a **`lightJmpContext`**.

5.6 HAL de memoria

El HAL de memoria (MEMHAL) tiene como única función reservar para uso del núcleo un bloque de memoria libre, idealmente el mas grande posible. Esta función es realizada por la rutina **`getSysMem`**, la cual es invocada por el administrador de memoria del núcleo durante su inicialización.

La detección de toda la memoria RAM disponible en un sistema no es una tarea sencilla[1], por lo que para la versión de 32 bits de PARTEMOS se realizó una implementación temporal y muy simple del CPUHAL, que devuelve un bloque de memoria fijo, conocido y seguro (desde la dirección 2MB hasta 3MB). Esto permitió enfocarse en el desarrollo del resto del sistema en lugar de dedicarle tiempo a la tarea de detección de memoria.

Una limitación actual del administrador de memoria del núcleo es que este se inicializa con un bloque contiguo de memoria libre. Sin embargo, la memoria física disponible en un sistema generalmente está fragmentada en varios bloques de memoria libre, lo cual impide al administrador de memoria hacer uso de toda la memoria disponible en el sistema. La solución de este problema en el futuro implicaría un cambio en la interfaz de MEMHAL.

Una alternativa sencilla para conocer la memoria disponible en el sistema es dejar la tarea de detección al cargador GRUB. El cargador es capaz de detectar los bloques de memoria disponibles en el sistema, y puede colocar la información en una estructura de datos accesible al sistema operativo. Con la información de memoria recopilada por GRUB, el HAL de memoria podría localizar el bloque de memoria más grande disponible, y devolverlo para su uso dentro del núcleo.

5.7 Archivos de la capa HAL

La tabla 4 resume los diferentes archivos que forman parte de la capa de abstracción de hardware de PARTEMOS, en sus versiones de 16 y 32 bits. Algunos archivos de encabezado, como "`intCPU.h`", no están asociados con ningún archivo de código fuente, mientras otros módulos (como el HAL de excepciones) no tienen definido ningún archivo de encabezados ".h".

Para cada archivo fuente se muestra la subcarpeta donde se puede localizar el mismo, la cual a su vez se encuentra dentro de la carpeta de fuentes de PARTEMOS ("`SRC`"). Todos los archivos de encabezado listados se encuentran en la subcarpeta "`include/HAL`" dentro de la carpeta de desarrollo de PARTEMOS.

[1] En http://wiki.osdev.org/Detecting_Memory_(x86) es posible encontrar más información al respecto.

Tabla 4: Listado de archivos de la capa HAL.

encabezado	carpeta	Archivo fuente	descripción
HAL de sistema			
sysHAL.h	-		inicialización primaria
-	HAL4IA32	c0hal.asm	Arranque del sistema
HAL de interrupciones			
iINTHAL.h	HAL4AT16	ATINTHAL.ASM	INTHAL para AT 16 bits
		IHAL16.C	
	HAL4IA32	iHal32.c	INTHAL para IA-32 con 8259
		irqhal32.asm	
		APICHalm.c	INTHAL para IA-32 sobre APIC
		irqAPICm.asm	
HAL de CPU			
ICPUHAL.H	HAL4AT16	CPUHAL16.C	CPUHAL para AT 16 bits
		cpuhal_.ASM	
	HAL4IA32	cpuhal32.c	CPUHAL para IA-32
		cpuhal.asm	
HAL de Excepciones			
-	HAL4AT16	ExcHal.asm	EXCHAL para AT 16 bits
	HAL4IA32	ExcHal.c	EXCHAL para IA-32
		ExcHal0.asm	
Otros módulos de la capa HAL			
intCPU.h	-		Activar/Desactivar interrupciones
iCLKHAL.H	HAL4PC\CLK	8253HAL .C	HAL de reloj para PC (16 y 32 bits)
iMemHal.h	HAL4AT16\MEM	MemHal.c	HAL de memoria para AT 16 bits
	HAL4IA32\MEM	tmpMemHal.c	HAL de memoria para IA-32

5.8 Procedimiento para portar el HAL a otras arquitecturas

En esta sección se brinda una metodología para portar la capa de abstracción de hardware (HAL) a otras plataformas. Para hacer más comprensible el texto, en algunas partes de este se supondrá que estamos haciendo la traducción del HAL a una plataforma de ejemplo, denominada “XYZ”.

5.8.1 Trabajo preliminar

No existe un orden a seguir para realizar las tareas mostradas en esta sección, por el contrario estas tareas están muy interrelacionadas y deben realizarse casi al unísono, constituyendo todo un trabajo de investigación preliminar, y evaluación de diferentes opciones.

La primera tarea preliminar e indispensable es la **selección de las herramientas de desarrollo** a utilizar. Estas herramientas incluyen (pero no están limitadas a) compilador, ensamblador, y preferiblemente algún depurador.

El compilador elegido debe **soportar la escritura de código ensamblador integrado**, ó alguna forma de emisión de código máquina dentro del código "C" [1]. También es necesario que el ensamblador y el compilador generen códigos objeto compatibles, de forma que puedan enlazarse juntos.

Otra tarea preliminar consiste en **investigar y elegir un método de arranque** (carga inicial) del sistema, lo cual puede implicar el uso de algún software cargador como GRUB. También se debe investigar como **realizar la configuración básica del sistema**, incluyendo como inicializar las tablas de memoria y de vectores de interrupción.

Por último se deben investigar los diferentes **métodos de salida de texto** disponibles, y elegir los que se consideren adecuados. La salida de texto es útil para realizar depuración de errores, y comprobar la ejecución del código. Entre los métodos de salida podemos encontrar alguna forma de escritura a pantalla, o el envío de caracteres a algún medio de registro. Algunos depuradores como Bochs brindan un método de salida de caracteres básico, que en el caso de Bochs se muestran en una consola especial, constituyendo una técnica muy sencilla y útil para depuración en etapas tempranas.

La versión actual del administrador de ventanas puede ejecutar salidas sin depender del núcleo del sistema y salidas a la ventana por defecto sin necesidad de inicialización. Si la nueva plataforma soporta salida de video en modo texto entonces es posible que en esta fase se desee **adaptar y probar el administrador de ventanas**, para utilizarlo como medio de salida durante el desarrollo subsiguiente.

Como resultado del trabajo hasta este punto, se debe contar **con código de ejemplo compilable y ejecutable en la nueva plataforma**, ya que este código sirve para comprobar las herramientas y métodos elegidos en esta sección.

5.8.2 Técnicas y carpetas de trabajo

El código del HAL, específicamente los archivos de encabezados ".h", está lleno de secciones compiladas condicionalmente de acuerdo con la plataforma. A la hora de modificar estos archivos se necesitará crear nuevas secciones para la nueva plataforma, lo cual puede hacer un poco engorroso el desarrollo. Alternativamente, se puede crear una nueva versión del HAL para la plataforma deseada, sin utilizar compilación condicional. Esto define dos técnicas generales de trabajo a la hora de portar el HAL a la nueva plataforma:

Técnica de desarrollo fusionado: Utiliza desde un inicio la compilación condicional, para crear una nueva versión de los archivos de encabezado que soporten la nueva plataforma además de las soportadas previamente. El desarrollo se hace "encima" de los archivos existentes, por lo que una vez portado el HAL a la nueva plataforma los archivos de encabezado estarán listos para ser utilizados en cualquier plataforma previamente soportada.

Técnica de desarrollo independiente: Consiste en crear una versión específica de la capa HAL para la nueva plataforma, usando como guía los archivos existentes. Una vez portado el HAL a la nueva plataforma, se requiere unificar las versiones de los archivos de encabezado del HAL para soportar todas las plataformas, mediante uso de compilación condicional.

[1] El ensamblador integrado se utiliza en la definición de algunas macros en encabezados ".h", teóricamente se podría redefinir estas macros para invocar funciones escritas en ensamblador, con lo cual se puede eliminar esta restricción, aunque a costa de un peor desempeño.

La alternativa a utilizar depende de factores como la habilidad del programador, y cuan diferente es la nueva arquitectura de las anteriores. Sin importar la alternativa utilizada, el resultado final debería ser el mismo. La alternativa de desarrollo independiente tiene ciertas ventajas como que nos permite olvidarnos en principio de la compilación condicional, y trabajar sobre un código mas limpio. Esta segunda alternativa fue la utilizada para portar el HAL a la arquitectura IA-32, lo que se justifica porque al ser la primera vez que se portaba este código todavía no existían secciones compiladas condicionalmente.

No importa la alternativa seleccionada, en la carpeta de desarrollo se deben **crear las subcarpetas adecuadas** para ubicar el nuevo código. El nombre de estas carpetas contendrá una palabra que identifique a la plataforma destino, preferiblemente de 4 letras o menos. Para nuestra plataforma destino de ejemplo, usaremos como nombre la palabra XYZ. En el caso del código específico de la capa HAL, este debe ubicarse en **una carpeta con el prefijo "`HAL4`",** por ejemplo para nuestra plataforma XYZ crearemos la carpeta "`HAL4XYZ`" para colocar el código del HAL, anidada dentro de la carpeta de fuentes "`SRC`".

También es muy probable que se necesite crear una **carpeta para archivos que brindan servicios específicos de la plataforma**[1], la cual debe comenzar con una letra "H", por lo que siguiendo nuestro ejemplo crearíamos la carpeta "SRC/HXYZ" con este objetivo.

Adicionalmente, es común que cada plataforma tenga su propia **carpeta de proyecto**, la cual incluye archivos como scripts de compilación, o archivos de proyecto para herramientas de desarrollo específicas. La carpeta de proyecto se ubica directamente dentro de la carpeta de desarrollo del núcleo, y generalmente se le coloca la terminación "prj". Para nuestra plataforma de ejemplo el nombre de la carpeta de proyecto podría ser "`XYZprj`".

La figura 35 muestra como quedaría la estructura de carpetas del núcleo luego de portarlo a la plataforma de ejemplo XYZ. Sólo se muestran algunas de las carpetas actuales del núcleo, ya que de mostrarse todas, la imagen quedaría muy grande y poco entendible. En el caso de la alternativa de desarrollo independiente, se debe copiar el código del HAL (todos los encabezados ".h", y los archivos fuente que se estimen convenientes) a una carpeta de trabajo temporal. En esta nueva carpeta se creará la versión independiente del HAL para la nueva plataforma, que una vez probada se puede combinar con la versión existente.

Para combinar las versiones del HAL se deben unificar los archivos de encabezados de la carpeta "`include`", utilizando compilación condicional donde sea necesario. En este paso es posible auxiliarse de herramientas de comparación de texto. Típicamente, el resto de las carpetas del HAL desarrollado independientemente se deben copiar sin cambio a la carpeta principal del núcleo, manteniendo la estructura de directorio.

[1] A diferencia del HAL, estos archivos no brindan una interfaz definida, pero generalmente serán utilizados por el propio HAL para implementar su interfaz estandarizada.

Figura 35: Estrategias de desarrollo fusionada e independiente.

5.8.3 Pasos para portar el código del HAL

A continuación mostramos una serie de pasos sugeridos para portar el código de la capa HAL. Para facilitar la explicación de estos seguiremos asumiendo que se realizará la traducción del HAL a la plataforma de ejemplo "XYZ".

En algunos lugares se hablará de la creación de código condicional como si se estuviera usando una técnica de desarrollo fusionado. En caso de utilizarse la técnica de desarrollo independiente, se debe crear el código indicado **omitiendo el uso de secciones compiladas condicionalmente**. Posteriormente en la etapa de combinación de versiones se deben **retomar las indicaciones para crear las secciones condicionales** necesarias.

Paso 1: Portar encabezados preliminares

Antes de continuar con el resto del código, se deben escribir las versiones para la nueva plataforma de los archivos de encabezado "`GLOBAL.H`", "`CPUTYPE.h`", "`intCPU.h`" y "`sysHAL.h`", los dos primeros localizados en la carpeta "`include`", y los restantes en "`include/HAL`". Estas versiones pueden ser independientes o combinadas con las ya existentes dependiendo de la técnica de trabajo elegida previamente.

En el archivo "GLOBAL.H" se debe definir condicionalmente un **símbolo que identifique la nueva plataforma**. Este símbolo típicamente comenzará con la cadena "**ARCH_**", por ejemplo para la plataforma XYZ se debe definir el símbolo "**ARCH_XYZ**". También se deben definir otros símbolos que permitan identificar el tipo de arquitectura (16 bits, 32 bits, u otro, ver sección 5.1 "Compilación condicional"). Para realizar lo anterior se necesita investigar que símbolo o conjunto de símbolos **definidos por el compilador permiten identificar unívocamente la nueva arquitectura** de las ya existentes. En caso de usarse una técnica de desarrollo independiente, estos pasos pueden postergarse hasta el momento de la combinación de versiones.

También es importante que en el archivo "GLOBAL.H" se **definan las macros ABORT** y **BREAK** para la nueva arquitectura. La macro **ABORT** termina el sistema de forma anormal, y una implementación muy sencilla sería deshabilitar las interrupciones y entrar en un ciclo infinito, como muestra la siguiente línea:

```
#define ABORT {LOCK;for(;;);}
```

Aunque esta es una implementación válida de **ABORT**, se recomienda que posteriormente se modifique para imprimir un mensaje con el archivo y número de línea donde se invocó. En el caso de la macro BREAK, su función es activar el depurador, y es posible dejar su implementación vacía.

El archivo "CPUTYPE.H" define los tipos de datos utilizados en el sistema. Si la nueva plataforma es de 16 o 32 bits, es posible que la versión actual de este archivo pueda utilizarse con poco o ningún cambio. En el próximo capítulo se brindan mas detalles de los tipos de datos básicos usados en PARTEMOS.

El archivo "intCPU.h" define rutinas para habilitar y deshabilitar interrupciones, que deben ser definidas para la nueva arquitectura, posiblemente utilizando código ensamblador integrado.

Por último, en el archivo "sysHAL.h" se debe **definir condicionalmente la macro** "**initPlatform()**", que debe contener el código necesario para realizar la configuración básica del sistema, definido entre las tareas de la sección 5.8.1 "Trabajo preliminar". Esta configuración constituye la **inicialización primaria** explicada en la sección 5.2. Adicionalmente, en el archivo "sysHAL.h" se debe definir la macro "**hardwareClean()**", que realiza acciones necesarias para dejar el hardware consistente antes de cerrar el sistema; y en caso de no ser necesaria podría definirse esta macro vacía.

Paso 2: Portar el HAL de interrupciones

El segundo paso consiste en crear una versión del HAL de interrupciones para la nueva plataforma. Este nuevo INTHAL debe brindar la abstracción del controlador virtual VPIC de PARTEMOS sobre el hardware de interrupciones disponible, implementando todos los servicios definidos en "iINTHAL.h". Los archivos fuente que implementen los servicios del INTHAL deben ser colocados en la carpeta fuente del HAL ("HAL4XYZ" para nuestra plataforma de ejemplo), mientras que los archivos implementados únicamente para brindar servicios de acceso al hardware se colocarán en la carpeta de la plataforma ("HXYZ").

En el caso del archivo "iINTHAL.h", si la nueva plataforma define un número de IRQs diferentes a la actual, o define constantes de IRQ especiales (como IRQ_A - IRQ_D para las IRQs del bus PCI), entonces se debe usar compilación condicional para **definir las constantes adecuadas** dentro de este archivo. También se debe **definir el símbolo TIMERIRQ**, para identificar el número de IRQ de reloj.

Si la nueva versión del INTHAL va a implementar la rutina **`restoreIrqHardware`** (para permitir restaurar el estado original del hardware al cerrar el sistema), entonces debe **definir el símbolo HW_RESTORE** para la nueva plataforma en el archivo "global.h".

En el archivo de interfaz del INTHAL también se deben **definir condicionalmente las macros "`DISABLE_NMI()`" y "`ENABLE_NMI()`"**. Si la nueva plataforma no soporta interrupciones no enmascarables estas macros pueden definirse vacías.

Finalmente, se debe **compilar y probar el INTHAL**. Las pruebas de unidad del INTHAL existentes para PC deben poder ejecutarse con poco o ningún cambio en la nueva plataforma. El uso de herramientas de depuración puede ayudar a localizar y corregir los errores que puedan existir.

Paso 3: Portar el HAL de CPU

El próximo paso consiste en crear una versión del HAL de CPU para la nueva plataforma, que implemente todos los servicios definidos en "`iCPUHAL.H`". Al igual que con el INTHAL, los archivos fuente del CPUHAL deben colocarse en la carpeta fuente del HAL (ej. "`HAL4XYZ`").

Actualmente el CPUHAL implementa dos grupos de rutinas para trabajos con contextos de CPU. Las rutinas que trabajan con contextos "pesados" son usadas dentro del mecanismo de excepciones del núcleo, mientras las rutinas de contexto ligeras se usan por el planificador del sistema para la conmutación de tarea. De seguirse con esta tendencia en un futuro, el CPUHAL debe implementar ambos conjuntos de rutinas.

Para utilizar las rutinas de trabajo con contextos de CPU "pesados", el archivo "`iCPUHAL.H`" debe **definir condicionalmente la estructura CPUCONTEXT**, tal y como se usa dentro del HAL de CPU. Esta estructura debe ser capaz de almacenar todo un contexto de ejecución para poder restaurar un punto de ejecución de una tarea, y es utilizada dentro del sistema de excepciones del núcleo. No se necesita redefinir la estructura LIGHTCONTEXT ya que esta es un apuntador a pila, y debería ser igual para diferentes plataformas (la estructura real del contexto ligero se salva en pila y queda oculta al núcleo).

Aunque la versión actual de CPUHAL se apoya en contextos ligeros en pila para la conmutación de tarea, en caso de requerirse una versión de conmutación de contexto diferente esta se puede realizar mediante cambios a los archivos del CPUHAL, sin necesidad de cambios en el núcleo. En este caso posiblemente sea necesario redefinir el tipo LIGHTCONTEXT dentro de "`iCPUHAL.H`", utilizando compilación condicional. Los servicios actuales del CPUHAL contienen la palabra "light" en su nombre (por ejemplo **`lightContextSwitch`**), en caso de implementarse rutinas que no se consideren "ligeras" es posible que se desee renombrarlos, lo cual implicaría también cambios menores en el núcleo.

Al igual que con el HAL de interrupciones, el **CPUHAL debe ser compilado y probado** utilizando las pruebas de unidad existentes.

Paso 4: Combinación de versiones en etapa intermedia (opcional)

Es posible que se desee comenzar a trabajar lo antes posible en portar el código del núcleo, por ejemplo si se quieren reducir riesgos de desarrollo, o si se trabaja en equipo y se desea continuar el desarrollo del

HAL y el resto del núcleo independientemente. En este caso se pueden utilizar los archivos del HAL de CPU e interrupciones creados en los pasos anteriores, para enlazarlos con el núcleo PARTEMOS. Para los restantes archivos del HAL, es posible implementar versiones temporales "nulas" o muy sencillas. Una versión nula implementaría las rutinas necesarias para enlazar con el núcleo sin error, pero sin colocar código en estas rutinas. En el paso siguiente se indicará para cuales archivos del HAL se pueden crear versiones "nulas", además de como crear las versiones finales.

En el caso de utilizarse una técnica de desarrollo independiente, este paso implicaría realizar la combinación adelantada de las carpetas de desarrollo del HAL y del núcleo. El desarrollo del HAL puede continuar en la carpeta combinada, o se puede seguir desarrollando en la carpeta independiente para realizar otra combinación al final.

Paso 5: Portar el resto del HAL

Una vez portados el HAL de interrupciones y el HAL de CPU, podemos proseguir a realizar la misma tarea con los restantes módulos del HAL: **HAL de excepciones**, **HAL de memoria**, y **HAL de reloj**.

El HAL de excepciones (EXCHAL) no define ningún archivo de encabezado, y no exporta ningún servicio al núcleo aparte de su inicialización (**`initExcHard`**) y finalización (**`endExcHard`**), por el contrario es el encargado de notificar eventos de excepciones al núcleo (mediante la invocación a **`excHandler`**). Por tanto si se desea omitir la creación del HAL de excepciones en etapas tempranas (para concentrarse en el resto del sistema) se podría crear un HAL de excepciones nulo, con las rutinas de inicialización y finalización vacías. Por otro lado el EXCHAL guarda mucha similitud con el HAL de interrupciones, por lo que una vez implementado este último no debería ser muy difícil **implementar el EXCHAL**, basándose en el código realizado para el INTHAL y otras versiones del EXCHAL existentes.

El HAL de memoria (MEMHAL), cuya interfaz esta definida por "`iMemHal.h`", tiene como única función obtener un bloque de memoria libre (idealmente el mas grande posible). Si se desea postergar la **implementación de detección de memoria** para concentrase en la programación del resto del sistema, se puede implementar una versión temporal muy sencilla (tomando como ejemplo "`HAL4IA32\MEM\tmpMemHal.c`") que devuelva una zona de memoria fija y conocida.

Por último se debe **crear un módulo que implemente las rutinas definidas por "`iCLKHAL.H`"**, para la configuración del temporizador del sistema. Esto no fue necesario en la versión de PARTEMOS para la arquitectura IA-32 debido a que esta utiliza el mismo módulo que la versión de 16 bits ("`8253HAL.C`"). La función del CLKHAL es configurar la frecuencia de interrupción del temporizador del sistema al valor que se le indique, por lo que es posible que crear una versión nula del HAL de reloj (CLKHAL) provoque que los servicios de tiempo no funcionen correctamente. Algunas plataformas inician con el temporizador configurado a una frecuencia predefinida, por lo que el uso de un CLKHAL nulo provocará que los servicios de tiempo trabajen con esa frecuencia. En otros casos el temporizador del sistema puede estar deshabilitado por defecto, por lo que la utilización de un CLKHAL nulo provocará que los servicios de tiempo dejen de funcionar.

Paso 6: Combinación de versiones final (opcional)

Si se utilizó la técnica de desarrollo independiente para crear una versión completa del HAL, entonces el último paso del proceso de portar la capa de abstracción de hardware es combinar la versión independiente obtenida con el código del HAL existente en la carpeta de PARTEMOS, como se describió en secciones anteriores.

Una vez portada la capa HAL, utilizando el procedimiento descrito en este capítulo, se puede proceder a portar el núcleo del sistema; esta debería ser una tarea relativamente sencilla, como se explica en el siguiente capítulo.

Capítulo 6
Desarrollo del núcleo de PARTEMOS

En el capítulo anterior se explica el trabajo realizado sobre la capa de abstracción de hardware (HAL) de PARTEMOS. El presente capítulo se enfoca en el trabajo realizado con el resto del código PARTEMOS. Se explica los cambios realizados sobre el código del núcleo, fundamentalmente dirigidos a que pueda compilar en cualquier plataforma (de 16 ó 32 bits). También se explican las técnicas utilizadas para la depuración del código PARTEMOS, y se proporciona el procedimiento a seguir para portar PARTEMOS a otra plataforma.

6.1 Cambios en el código PARTEMOS

Durante el desarrollo de la versión portable de PARTEMOS fue necesario realizar varios cambios al código existente. La mayor parte de los cambios estuvieron dirigidos a eliminar las dependencias de la plataforma y el compilador presentes en el código del núcleo, pero también se realizaron otros cambios no relacionados con la portabilidad. Las subsecciones siguientes describen los cambios realizados, como parte del presente trabajo, al código existente.

6.1.1 Restructuración de carpetas.

Anteriormente el código de PARTEMOS separaba los archivos escritos en ensamblador del código fuente en C. Los archivos escritos en ensamblador, mayormente de la capa HAL, estaban contenidos en una carpeta nombrada "`SRA`", y los restantes archivos fuentes en la carpeta "`SRC`". Esta separación dificultaba el trabajo con ciertos módulos, como por ejemplo las nuevas versiones de algunos módulos de la capa HAL, que están escritas en ambos lenguajes de programación. Para facilitar el trabajo con estos módulos la estructura de carpetas de PARTEMOS sufrió una ligera modificación, eliminándose la carpeta "`SRA`". Los archivos escritos en ensamblador fueron movidos a subcarpetas dentro de la carpeta de fuentes de PARTEMOS ("`SRC`"). De esta forma el código fuente ya no está separado por el lenguaje que se escribe, sino por su función y plataforma destino (en los casos que aplique).

Además de la eliminación de la carpeta "`SRA`", la estructura de carpetas de PARTEMOS no sufrió otros cambios mayores, solo se agregaron nuevas carpetas con archivos específicos para la nueva arquitectura IA-32. En la sección 6.6 se muestra la estructura de carpetas actual de PARTEMOS.

6.1.2 Cambios relacionados a los tipos de datos

Como se mencionó en el capítulo anterior, el archivo "`CPUTYPE.H`" define los tipos de datos básicos utilizados en el sistema. El uso de algunos de estos tipos de datos dificultaba el portado de PARTEMOS.

Uno de los tipos de datos mas utilizados en PARTEMOS16 era el tipo **`WORD`**, definido como un entero sin signo de 16 bits. Al portar el código de PARTEMOS a 32 bits, la mayoría de las ocurrencias del tipo **`WORD`** debían ser traducidas como un entero sin signo de 32 bits. Sin embargo, en algunos lugares del código el tipo **`WORD`** todavía debía entenderse como de 16 bits, por ejemplo el tipo de datos devuelto al leer los contadores del temporizador 8253 siempre es de 16 bits. Esto indica que en PARTEMOS16 el tipo **`WORD`** era utilizado incorrectamente, ya que no se diferenciaba entre su uso para representar tipo de datos del

tamaño de la palabra de la máquina (dependiente de la plataforma) y los tipos datos que siempre serán de 16 bits.

Para evitar cualquier problema relacionado con el uso del tipo **WORD**, se decidió eliminar la definición del tipo, de forma que cualquier uso del mismo fuera detectado como un error de compilación. En muchos lugares la utilización del tipo **WORD** fue remplazada por el tipo **WORD_**, que representa un entero sin signo cuyo tamaño varía según la arquitectura. En otros lugares fue remplazado por **UINT16**, indicando que el dato es de 16 bits sin importar el tamaño de palabra de la arquitectura usada. Por último, otras ocurrencias del tipo **WORD** fueron remplazadas por el tipo **WORD16**, que es similar a **UINT16** pero sólo está definido para arquitecturas de 16 bits. El tipo **WORD16** es útil para enfatizar el hecho de que el código sólo es utilizable en arquitecturas de 16 bits, y cualquier intento de compilarlo para otras arquitecturas resultará en un error de compilación.

Otro tipo que también era sensible de interpretarse de varias formas era el tipo **DWORD** (definido en PARTEMOS16 como un entero sin signo de 32 bits), por lo que también fue eliminado y remplazado por otros tipos, de forma similar a como se hizo con el tipo **WORD**.

Para hacer el código de PARTEMOS consistente con el modelo de memoria plano (y compilable para 32 bits) prácticamente todos los apuntadores de tipo FAR (formados por un segmento y un desplazamiento) fueron eliminados del núcleo. El uso del tipo **FARMEMPTR** dentro del núcleo fue remplazado por el tipo **MEMPTR**, y la definición de **FARMEMPTR** sólo permanece disponible a códigos específicos de la plataforma original de 16 bits. El tipo **VRAMPTR**, que representa un apuntador a la memoria de video en modo texto, también es definido como un apuntador FAR en la arquitectura PC de 16 bits, sin embargo esto es transparente al código del núcleo y no afecta su portabilidad.

A diferencia de los otros tipos de datos, la eliminación de apuntadores FAR implicó no solo el remplazo de un tipo por otro. En algunos casos se hizo necesario modificar código adicional que funcionaba específicamente con este tipo de apuntadores. En el capítulo anterior vimos como la eliminación de apuntadores FAR provocó cambios en la interfaz del módulo CPUHAL, específicamente de la rutina de inicialización de contextos.

La tabla 5 muestra los diferentes tipos de datos definidos en el archivo "`CPUTYPE.H`", los cuales son definidos condicionalmente según la plataforma destino. Se puede apreciar como algunos tipos mantienen su tamaño independientemente del tamaño de palabra de la arquitectura, mientras otros varían de tamaño. Otros tipos sólo están definidos para una plataforma específica, y sólo pueden ser utilizados en código dependiente de la plataforma, por ejemplo en archivos de la capa HAL o en archivos que brindan mecanismos de acceso al hardware.

Como muestra la tabla, el archivo "`CPUTYPE.H`" no sólo define tipos de datos, sino también otras macros relacionadas con los tipos de datos, útiles para hacer el código que la use independiente de la plataforma.

Tabla 5: Tipos y macros definidas en "CPUTYPE.H".

Símbolo	Tamaño ó valor en 16 bits	Tamaño ó valor en 32 bits	descripción
BITSxINT	16	32	Número de bits del tipo entero
MAX_UINT	0xFFFF	0xFFFFFFFF	Máximo valor de un entero sin signo
Tipos con tamaño independiente de la arquitectura			
BYTE	8	8	Entero sin signo de 8 bits
UINT16	16	16	Entero sin signo de 16 bits
UINT32	32	32	Entero sin signo de 32 bits
Tipos que varían su tamaño según la arquitectura			
WORD_	16	32	Entero sin signo
DWORD_	32	64	Entero largo sin signo
MEMPTR	16	32	Apuntador a memoria
VRAMPTR	32	32	Apuntador a memoria de video
Tipos específicos a una arquitectura			
INT16	16	-	Entero con signo sólo para 16 bits
WORD16	16	-	Entero sin signo sólo para 16 bits
DWORD32	32	-	Entero largo sin signo para 16 bits
FARMEMPTR	32	-	Apuntador far sólo para AT16
INT32	-	32	Entero con signo sólo para 32 bits
WORD32	-	32	Entero sin signo sólo para 32 bits
Macro de acceso a memoria de video			
videoRAMXY(x,y)	Apuntador a la memoria de video del carácter [x; y]		

6.1.3 Otros cambios menores

Como la versión original de PARTEMOS fue creada para un compilador sobre MS-DOS, todas las inclusiones de archivos ".h" que requerían especificar una ruta relativa utilizaban como separador de carpetas el carácter "\". Sin embargo la versión de GCC usada requería rutas estilo UNIX, con el carácter "/" como separador. Como el compilador Borland C utilizado para compilar código de 16 bits también soporta el uso de rutas estilo UNIX, todas las ocurrencias del carácter "\" dentro de las rutas "include" fueron remplazadas por "/".

El compilador cruzado para IA-32 no tiene acceso a funciones de la biblioteca estándar de C, por lo que se hizo necesario implementar algunas de las mismas. El archivo "`syslib.c`" contiene la implementación de algunas funciones estándares para trabajo con memoria y cadenas de caracteres. Estas funciones fueron definidas en el archivo de encabezado "`global.h`", eliminándose la inclusión de encabezados estándares dentro del código de PARTEMOS. La versión de 16 bits de PARTEMOS no necesita incluir el archivo "`syslib.c`", ya que tiene acceso a la biblioteca estándar de C. Debemos destacar que estas funciones se crearon sólo para poder compilar el código de PARTEMOS existente, y

no con la intención de portar la biblioteca estándar, que es mucho más extensa. Sin embargo es posible que en un futuro se implemente toda la biblioteca estándar de C, para su uso por parte de las aplicaciones de PARTEMOS.

El código de PARTEMOS y de algunas aplicaciones creadas para el sistema hacia uso de las rutinas definidas en "`conio.h`", que es un encabezado C no estándar útil por sus funciones de entrada y salida en modo texto. Las invocaciones a las funciones de salida en pantalla de "`conio.h`" fueron remplazadas por funciones similares del administrador de ventanas en modo texto de PARTEMOS. En la sección 6.5 se brinda más información de este módulo y otras funciones de salida.

Anteriormente el driver de reloj de PARTEMOS utilizaba el identificador de IRQ0 (**`IRQ00ID`**) para capturar la interrupción de reloj, proveniente del temporizador 8254. Como se vio en el capítulo anterior, el controlador de interrupciones IOAPIC normalmente tiene la línea de interrupción del 8254 conectada a su entrada 2, a diferencia del PIC8259 que la conecta a su entrada 0. Para hacer el código existente compilable sin importar el controlador utilizado, la interfaz del HAL de interrupciones define condicionalmente la constante **`TIMERIRQ`**, la que a su vez es utilizada para crear el identificador de interrupción **`TIMERIRQID`** (en el archivo "`iKRNLINT.H`"). Ahora el driver de reloj del sistema hace uso de este identificador para poder compilar sin cambios.

La macro "**`IF()`**", que es usada dentro de las aseveraciones para verificar la bandera de interrupciones del procesador, ahora devuelve un valor distinto de cero si las interrupciones están habilitadas (antes devolvía 1), esto hizo su implementación mas sencilla, pero requirió que se modificaran muchas condiciones donde se comparaba esta macro con 1.

Algunas secciones del código del núcleo dependientes de la plataforma fueron movidas a archivos de encabezados ".h" y encapsuladas como macros, las que son definidas condicionalmente. Un ejemplo es la macro de inicialización primaria ("**`initPlatform()`**", definida en "`sysHAL.h`", ver sección 5.2), que es lo primero que debe colocarse en la función **`main`** del núcleo o de cualquier código de prueba. Otro ejemplo es la macro "**`videoRAMXY(x,y)`**", que como ya vimos está definida en el archivo "`CPUTYPE.H`", y devuelve un apuntador a un carácter específico dentro de la memoria de video en modo texto.

Las invocaciones a la función **`exit`** para detener la ejecución del núcleo fueron encapsuladas en la macro **`ABORT.`** De forma similar el uso del ensamblador integrado para activar el depurador de 16 bits (generando la interrupción 3), fue encapsulado por la macro **`BREAK`**. Ambas macros se definen condicionalmente en "`global.h`", mencionada en el capítulo anterior.

Otras secciones extraídas del código están relacionadas con la manipulación de la pila de ejecución, como se describe en la sección 6.2 más adelante.

6.1.4 Cambios asociados a las modificaciones realizadas al HAL

Como se vio en el capítulo anterior la capa de abstracción de hardware (HAL) de PARTEMOS sufrió algunos cambios en su interfaz, específicamente en los módulos INTHAL y CPUHAL. Estos cambios en la interfaz provocaron cambios en el código del núcleo que hace uso de la misma.

Los cambios en la interfaz del INTHAL no provocaron cambios significativos en el código del núcleo. Esencialmente, estos cambios se limitaron a eliminar las invocaciones de algunos servicios o remplazarlas por otras invocaciones, como se explica en el capítulo anterior (sección 5.3.2).

La introducción de la conmutación de contexto ligera en el HAL de CPU también trajo consigo algunos cambios en el código del núcleo. Todas las invocaciones a rutinas que trabajaban con contexto fueron remplazadas por sus alternativas "ligeras", exceptuando aquellas usadas dentro del mecanismo de excepciones de PARTEMOS. El uso de una estructura tipo **CPUCONTEXT** dentro de los bloques de control de actividad (estructura **ACB**, módulo KRNLAAL) fue remplazado por un apuntador al tope de pila de la tarea interrumpida, representado por el tipo de datos **LIGHTCONTEXT**.

6.1.5 Cambios en el tratamiento de procedimientos asíncronos

El envío de procedimientos asíncronos a otros contextos sufrió varios cambios dentro del código de PARTEMOS. Estos cambios fueron motivados no sólo por la introducción de los contextos ligeros en el CPUHAL, sino también por la necesidad de mejorar la portabilidad del código y hacerlo mas legible. En la sección 5.5.3 del capítulo anterior se explica brevemente el concepto de "procedimiento asíncrono", y como se agregó al módulo CPUHAL el servicio **InjectAsyncProcedure**, con el objetivo de auxiliar al núcleo en el envío de procedimientos asíncronos.

La figura 36 muestra las principales rutinas involucradas en el envío (inyección) de procedimientos asíncronos, localizadas en el archivo "`krnlAsy.c`". La función **_asyncProcedure** (líneas 1-10) es el manejador interno de todos los procedimientos asíncronos, que se encarga de invocar otros manejadores de capas superiores (manejadores de señales asíncronas). La ejecución de la función **_asyncProcedure** puede ser "forzada" en otros contextos (lo que conocemos como "inyectar la función"), o puede invocarse en el contexto de la propia tarea en ejecución, lo que llamaremos "invocación directa".

La función **_asyncProcedure** recibe un contexto (parámetro **interruptContext**) que permite restaurar la ejecución de la tarea interrumpida por el procedimiento asíncrono. En PARTEMOS16, el parámetro **interruptContext** siempre era creado y utilizado para abandonar el procedimiento asícrono, sin importar la forma en que este era invocado. A diferencia de la versión anterior, la versión actual de esta función retorna normalmente si fue invocada de forma directa. En caso de ser invocada directamente, el parámetro **interruptContext** tendrá un valor cero, lo cual se verifica en la línea 6 de la figura 36. La línea 7 restaura el contexto de ejecución interrumpido, sólo en caso de que el procedimiento asíncrono haya sido inyectado al contexto. Esta optimización mejora la velocidad de ejecución de los procedimientos asíncronos, y le da más claridad al código.

El servicio **_deliveryAsyncProcedure** (figura 36, líneas 12-22) es el encargado de inyectar el procedimiento asíncrono **(_asyncProcedure**) en el contexto de una tarea detenida. La mayor parte del trabajo de inyección es realizado por el servicio **InjectAsyncProcedure** (línea 21) del CPUHAL (véase sección 5.5.3).

```
1  LOCAL void _asyncProcedure(WORD_ arg, LIGHTCONTEXT interruptContext)
2    {
3      /* Manejo interno e invocación a manejadores superiores */
4      /*                          ...                         */
5
6      if(interruptContext) {        /* Si la invocación fue indirecta    */
7          lightJmpContext(interruptContext); /* Salta al contexto ligero */
8      }
9      /* si se invocó directamente retornamos normalmente     */
10   }
11
12 PUBLIC void _deliveryAsyncProcedure(void)
13   {
14     WORD_ param;
15     /* Verificar si es posible enviar la señal, sino salir  */
16     /*                          ...                         */
17
18     param = currentTask->asyncPosted;  /* Obtiene código de la señal    */
19     currentTask->asyncPosted = 0;      /* Limpia notificación de señal  */
20     /* "Inyectar" procedimiento _asyncProcedure en la tarea currentTask */
21     InjectAsyncProcedure( &currentTask->stContext, _asyncProcedure, param);
22   }
23
24 LOCAL void _doAsyncProcedure(void)
25   {
26     WORD_ param;
27     /* Verificar si es posible enviar la señal, sino salir  */
28     /*                          ...                         */
29     param = currentTask->asyncPosted; /* Obtiene código de la señal    */
30     currentTask->asyncPosted = 0;     /* Limpia notificación de señal  */
31     _asyncProcedure(param, 0 ); /* invocación directa a _asyncProcedure*/
32   }
```

Figura 36: Rutinas (simplificadas) para envío y tratamiento de procedimientos asíncronos.

Con el uso del servicio **`InjectAsyncProcedure`**, la función deliveryAsyncProcedure se simplificó grandemente y se hizo más portable. Se eliminaron secciones de código no portables, e invocaciones a rutinas como **`contextPush`** y **`changeContextCP`**. El código existente previamente asumía una forma específica de pase de parámetros a las funciones C, que no siempre es usada en todas las arquitecturas. En el caso de que se quiera portar el código a otra plataforma, o se quiera implementar otro método de conmutación de contexto, la rutina **`InjectAsyncProcedure`** puede encapsular toda la lógica necesaria sin requerirse cambios en el núcleo.

La rutina **`_doAsyncProcedure`** (líneas 24-32 de la figura 36), es usada internamente por el módulo "`krnlAsy.c`" cuando necesita realizar la invocación directa de **`_asyncProcedure`**. Se puede apreciar en la línea 31 como esta invocación se realiza enviando 0 como parámetro de contexto. De esta forma el código de la rutina queda mas claro, y se ahorra la sobrecarga de generar un contexto para posteriormente restaurarlo.

6.1.6 Corrección de errores

A continuación se mencionan los cambios realizados en el código de PARTEMOS para corregir algunos errores detectados, de los cuales la mayor parte sólo se manifestaba en la versión de 32 bits. Aunque

también se corrigieron algunos errores presentes en la versión de 16 bits, y que habían permanecido sin detectar hasta la fecha.

Condiciones de carrera dentro del núcleo

Se detectó un problema dentro del núcleo de PARTEMOS que podía impedir el inicio de algunas tareas. En PARTEMOS puede suceder que algunos tipos de tarea (específicamente las tareas periódicas y aperiódicas) se reinicien a si mismas. En algunos casos muy raros una conmutación de contexto dentro del código de reinicio podía impedir que la próxima instancia de la tarea se ejecutara. La corrección de este problema requirió algunos cambios menores dentro del código del núcleo.

Los detalles de las circunstancias específicas que podían provocar las condiciones de carrera antes mencionadas fueron reportados en un documento independiente. El documento incluye además varias alternativas de solución al problema, así como la solución elegida finalmente, incluyendo las modificaciones realizadas al código.

*Posible problema con las macros **ATOMIC/ENDATOMIC***

El par de macros **ATOMIC/ENDATOMIC** permite ejecutar ciertas regiones de código con interrupciones deshabilitadas a nivel de CPU (o sea, atómicamente), restaurando luego las interrupciones a su estado original. Para lograr esto, **ATOMIC** apila el registro de banderas al inicio de la sección protegida, y **ENDATOMIC** lo restaura posteriormente, usando código ensamblador integrado para su implementación. Se detectó que el uso de esta misma idea de implementación en 32 bits puede provocar problemas en algunas circunstancias.

Dependiendo de las opciones de compilación usadas, el compilador GCC puede generar código que accede a los parámetros en pila utilizando el registro ESP (tope de pila) en lugar de EBP. Al apilarse el registro de banderas (EFLAGS) el valor de ESP se modifica, por lo que los parámetros en pila serán accedidos de forma incorrecta. Este problema no sucede si se le indica al compilador que genere marcos de pila estándar.

El problema descrito no se puede considerar un error del código existente previamente, debido a que las macros mencionadas son específicas de la plataforma, y funcionan correctamente en la versión de 16 bits. Sin embargo lo hemos descrito dentro de esta sección, porque fue un error introducido al usar una técnica que funcionaba correctamente en una plataforma, pero que puede ser problemática al utilizarse en nuevas plataformas.

Para evitar este error, las macros **ATOMIC/ENDATOMIC** para la versión de 32 bits de PARTEMOS fueron definidas de la siguiente forma:

```
#define ATOMIC { int _SAVE_FLAGS_;\
  __asm__("pushf\n cli\n pop %0": "=m" (_SAVE_FLAGS_));
#define ENDATOMIC __asm__("push %0\n popf"::"m" (_SAVE_FLAGS_)); }
```

Esta definición provoca que la bandera de interrupciones sea salvada en la variable local **_SAVE_FLAGS_**, la cual es conocida por el compilador. De esta forma se evita la modificación del tope de pila por parte del código ensamblador, eliminándose errores de acceso a la misma.

Problemas con cálculos de memoria

Como los parámetros apilados para la versión de 32 bits de PARTEMOS tienen el doble de tamaño que para la versión de 16 bits, la pila de las tareas en 32 bits deberían tener al menos el doble de tamaño que sus equivalentes en 16 bits. Este hecho provocó situaciones de desborde de pila muy difíciles de detectar durante el desarrollo en 32 bits. Para evitar este tipo de problemas se aumentó el tamaño de pila por defecto en el módulo de configuración de PARTEMOS para 32 bits (parámetro de configuración "STACKSIZE") y se fijó el parámetro de configuración "STACKSHRINK" a falso, para impedir la creación de pilas más pequeñas. Sin embargo este aumento del tamaño de pila provocaba errores de memoria insuficiente en la versión de 32 bits, a pesar de que la máquina de ejecución contaba con suficiente memoria libre.

El problema radicaba en que la versión de 16 bits del núcleo PARTEMOS, al estar más restringida en memoria hacía un cálculo muy estricto de la cantidad de memoria que reserva para la creación de pilas y otros objetos del núcleo. Este mismo cálculo se estaba utilizando para la versión de 32 bits, a pesar de que esta última utiliza mucha mas memoria para los objetos del núcleo, lo cual provocaba los errores de memoria insuficiente. La solución adoptada para que el código pudiera funcionar tanto en 16 como en 32 bits fue multiplicar la cantidad de memoria reservada por el núcleo por un factor, definido de la siguiente manera:

```
#define FACTOR (BITSxINT/10)
```

Donde **`BITSxINT`** representa la cantidad de bits de la palabra de la máquina (16 o 32). De esta forma al compilar sobre 16 bits el factor se calcula como 1 (resultado de la división entera), por lo que se seguirá usando el cálculo de memoria estricto. Sin embargo al compilarse la versión de 32 bits, el cálculo del factor resultará en 3, por lo que se asignará tres veces mas memoria a los objetos del núcleo.

Problema causado por el uso de una constante numérica

Se detectó un problema dentro de un archivo del administrador de memoria de PARTEMOS (específicamente "`listmngr.c`"), que era provocado por el uso de un tamaño fijo (2 bytes) para el tamaño de palabra de la máquina. Esto provocaba accesos a direcciones de memoria no existentes, lo que a su vez hacía que en un emulador específico (VirtualBox) fallara una precondición[1]. La solución adoptada fue remplazar la constante por el operador **`sizeof`** para obtener el tamaño de palabra del tipo **`WORD_`** (ver sección 6.1.2 para información sobre la introducción del tipo **`WORD_`**).

6.2 Macros de manipulación de la pila de ejecución

PARTEMOS16 contenía varios fragmentos de código no portable relacionados con la manipulación de la pila de ejecución de la CPU. El código también hacía uso de funcionalidades de verificación de desbordamiento de pila que sólo están disponibles en los compiladores Turbo/Borland C. Todos estos fragmentos fueron encapsulados en macros para trabajo con pila y colocados en el archivo "`stack.h`", donde las macros son definidas condicionalmente según la plataforma. Las variantes de PARTEMOS para 16 y 32 bits tienen su propia implementación de estas macros.

[1] La diferencia de comportamiento entre emuladores se debía a que al accederse erróneamente a una dirección de memoria no existente, VirtualBox devolvía 0 mientras que Bochs devolvía otro valor.

A continuación se describen las diferentes macros para trabajo con la pila.

`stackSpace(pos)`

Devuelve el espacio de pila (en palabras de la máquina – WORD_) existente entre el tope de pila actual y la posición dada por **`pos`**. El valor devuelto es positivo si la posición se aleja del tope en sentido de crecimiento de la pila (puede servir para calcular espacio libre en pila) y negativo si la dirección está en sentido de decrecimiento de la pila (puede servir para calcular cantidad de pila usada. Esta macro actualmente sólo se utiliza en la verificación de precondiciones.

`setStack(val)`

Cambia el valor del registro apuntador al tope de pila de la CPU. Esta macro es utilizada en PARTEMOS para establecer una pila temporal durante la destrucción de una tarea.

`setStackChk(val)`

Habilita la verificación de desbordamiento de pila. Recibe el tope máximo de la pila actual. La macro debe dejar el margen de seguridad que estime conveniente para detectar el desbordamiento. La versión de esta macro para 16 bits utiliza el mecanismo de verificación de pila del compilador. La versión para 32 bits utiliza un registro de depuración para detectar el acceso a una zona de memoria cercana al tope de pila[1].

`disableStackChk()`

Deshabilita verificación de desbordamiento de pila.

`mainStackLimit()`

Devuelve el límite de pila utilizado por la función **`main`**, que posteriormente será la pila de la tarea nula. Sólo garantiza devolver un valor válido si se usa antes que **`setStackChk`** durante la inicialización del núcleo.

6.3 Archivos específicos de la plataforma

Cada plataformas de hardware soportada por PARTEMOS puede tener un conjunto de archivos específicos, que brindan mecanismos de acceso al hardware. Los mecanismos brindados por estos archivos pueden ser usados por módulos de la capa HAL o por los drivers de dispositivos, como un medio para implementar sus servicios. A diferencia de la capa HAL o de los drivers de dispositivos, estos archivos no brindan una interfaz estándar reconocida por el núcleo.

Como se mencionó en el capítulo anterior, estos archivos específicos de la plataforma deben colocarse en una subcarpeta de la carpeta de fuentes "`SRC`", cuyo nombre debe estar formado por la letra "H" seguida de una cadena que identifique la plataforma. Como las versiones de 16 y 32 bits de PARTEMOS utilizan básicamente el mismo hardware, muchos de los archivos específicos existentes para la versión de 16 bits se utilizaron sin cambio en la versión de 32. Por este motivo se decidió mantener la carpeta de archivos específicos existente ("`HPC`") como carpeta común para ambas arquitecturas. Los archivos

[1] Puede suceder que una función se salte la zona de la pila verificada sin accederla, pudiendo ocurrir un desbordamiento sin detectar, pero esta situación es muy rara.

específicos de la versión de 32 bits de PARTEMOS fueron colocados en la subcarpeta "HPC/IA32", y se mencionan a continuación.

APIC.c

Brinda funciones de acceso a los controladores de interrupción IOAPIC y LAPIC típicos de una PC moderna. Los servicios de este módulo son utilizados por las versiones del INTHAL sobre APIC (ver sección 5.3.4)

Descript.c

Brinda funciones para la inicialización y manejo de las tablas de descriptores GDT e IDT. Estas funciones son utilizadas en la macro de inicialización primaria para la arquitectura IA-32, y en los módulos INTHAL y EXCHAL.

HwDebug.c

Brinda rutinas de acceso a los registros de depuración de la CPU, permitiendo el establecimiento de diferentes tipos de breakpoints de hardware. Estas rutinas se utilizan en la detección de desbordamientos de pila, y para la depuración de errores.

6.4 Modulo de configuración

El módulo de configuración de PARTEMOS tiene como función la lectura de parámetros de configuración del núcleo, que pueden ser cambiados sin recompilar el sistema. La lectura de parámetros en la versión de 16 bits de PARTEMOS es implementada por dos archivos: "KRNLCFG.C", que brinda al núcleo la interfaz del módulo de configuración, y "CONFIG.C" que proporciona servicios auxiliares. La carga de los parámetros se realiza antes de que PARTEMOS tome el control de la máquina, lo que permite a estos archivos utilizar funciones de entrada/salida brindadas por MS-DOS para leer archivos texto con la configuración del sistema.

Los archivos de lectura de configuración antes mencionados utilizan servicios de MS-DOS, fragmentos de ensamblador integrado y otras características que los hacen no portables. Por este motivo se realizó una versión limitada de los mismos, que puede ser utilizada en la versión de 32 bits de PARTEMOS, y teóricamente en cualquier plataforma. La versión limitada está compuesta por los archivos "KCfgLi.c" y "configLi.c", que remplazan a sus equivalentes "KRNLCFG.C" y "CONFIG.C", respectivamente.

La versión limitada de los archivos de configuración no brinda grandes funcionalidades comparada con la versión de 16 bits, pero tiene la ventaja de que puede compilar prácticamente en cualquier plataforma. Esta nueva versión utiliza una tabla de configuración estática, para evitar el uso de la rutina estándar **malloc**. En lugar de hacer el análisis sintáctico de un texto de configuración, la versión limitada simplemente invoca en secuencia a las funciones adecuadas para insertar estáticamente los diferentes parámetros de configuración, y así llenar la tabla de configuración. La versión limitada tampoco soporta perfiles de configuración, los cuales requieren interactuar con el usuario por medio de menús, utilizando servicios de entrada/salida no portables.

La versión limitada de los archivos de configuración es útil para compilar el núcleo PARTEMOS en etapas tempranas de su desarrollo en cualquier plataforma. Posteriormente, estos archivos pueden remplazarse

por versiones específicas dependientes de la plataforma, como por ejemplo un módulo que cargue parámetros desde una ROM de configuración.

6.5 Funciones de salida

Llamamos funciones de salida a los procedimientos utilizados para brindar al usuario algún tipo de retroalimentación acerca de la ejecución de un programa. Una forma de retroalimentación muy común consiste en mostrar cadenas de texto en una pantalla, pero no es la única posible. Las funciones de salida juegan un importante papel en la detección de errores de código.

Durante el desarrollo de la versión de PARTEMOS para la arquitectura IA-32, se utilizaron varias técnicas de salida de diferente complejidad. Las distintas técnicas desarrolladas se utilizaron dependiendo de la fase de desarrollo de PARTEMOS. Los archivos de salida específicos de la arquitectura IA-32 fueron colocados en la carpeta "`SRC/outIA32`".

Las técnicas de salida varían en complejidad, y van desde sencillas escrituras de caracteres individuales, hasta salida de texto formateada hacia diferentes regiones de la pantalla. En las primeras etapas de desarrollo del HAL se utilizaron técnicas de salida sencillas, que posteriormente fueron remplazadas por otros métodos de salida de mayor complejidad.

A continuación se describen las diferentes funciones de salida utilizadas durante el desarrollo de PARTEMOS.

6.5.1 Técnicas de salida de caracteres individuales

Estas técnicas de salida se basan en escribir caracteres individuales a un dispositivo de salida, encargado de mostrar estos caracteres al usuario. Generalmente se utilizan debido a su sencillez de implementación, y en algunos casos a la seguridad de que trabajarán sin errores y a la velocidad de las mismas. Son útiles en etapas tempranas de desarrollo de un núcleo, incluyendo las fases de prueba iniciales cuando todavía no se comienza a portar la capa HAL. Las técnicas de salida de caracteres individuales sirven de base para la implementación de funciones más complejas, que permiten la escritura de cadenas de caracteres.

Durante el desarrollo de la versión de 32 bits de PARTEMOS se utilizaron las siguientes técnicas de salida de caracteres.

Escritura directa de bytes a memoria de video

Utilizada en etapas de prueba iniciales y etapas tempranas de desarrollo de la capa HAL. Esta técnica consiste en escribir bytes a diferentes posiciones de la memoria de video en modo texto, con lo que podemos rastrear la ejecución de un programa y detectar el punto específico donde ocurrió un error. Es una técnica de salida muy rápida y puede funcionar en cualquier máquina real o virtual.

Para utilizar esta técnica el usuario debe crear su propia codificación, dándole un significado a los diferentes caracteres mostrados en pantalla. También es posible utilizar diferentes colores de caracter y de fondo, a los cuales también se les puede dar su propio significado.

Entre las desventajas de esta técnica podemos mencionar que requiere que los descriptores de memoria estén correctamente configurados, de forma que se pueda acceder a la zona de memoria de video. En el caso del desarrollo para 32 bits de PARTEMOS, el cargador GRUB cede el control con los descriptores correctamente configurados, por lo que esta técnica se puede utilizar desde un inicio. Posteriormente PARTEMOS vuelve a configurar los descriptores de memoria, y si esta operación se realiza incorrectamente la escritura a video no será visible.

Otro problema de esta técnica es que si se escriben caracteres en la misma posición sólo se mostrará el último escrito. Para minimizar este problema, se implementó una función de escritura de caracteres con apuntadores a memoria de video variable, rotando los colores de los caracteres cuando se volvía a escribir desde la posición inicial.

La escritura de caracteres individuales a pantalla fue posteriormente descontinuada, utilizándose en su lugar las funciones de escritura a pantalla brindadas por el módulo "`conio.c`" y el administrador de ventanas de PARTEMOS, que serán descritos en próximas secciones.

Escritura de bytes al puerto de trazas de Bochs

Como se ha mencionado previamente, el emulador Bochs puede mostrar en una consola cualquier carácter escrito al puerto 0xE9. Según la propia documentación de Bochs[1], la idea es proporcionar un método de salida para depuración en etapas muy tempranas de desarrollo de código de BIOS o sistemas operativos. Por supuesto este método de salida sólo funcionará si el sistema es ejecutado en una máquina emulada por Bochs, y el archivo de configuración de la máquina virtual tiene habilitada la opción **`port_e9_hack`**.

La escritura al puerto 0xE9 (que llamaremos puerto de trazas de Bochs) es uno de los métodos de salida más seguros. Una vez configurada correctamente la máquina virtual está garantizado que todas las salidas a este puerto se mostrarán en la consola de Bochs, aun cuando existan errores como una configuración de memoria incorrecta.

Toda escritura al puerto de trazas es mostrada secuencialmente en la consola, por lo que tampoco existe el problema de sobrescritura de posiciones en memoria de video. Sin embargo la consola tiene el problema de que, en caso de ser la salida muy extensa, las primeras líneas escritas pueden perderse. Esto se debe a que la consola de Bochs (al igual que otras ventanas de consola sobre Windows) posee un buffer de tamaño limitado. En caso de ser insuficiente, el tamaño del buffer de la consola se puede aumentar en las propiedades de la ventana de consola.

El uso del puerto de trazas de Bochs es un método de salida relativamente lento. En caso de utilizarse este método indiscriminadamente la ejecución del sistema, que de por si ya es lenta debido a la emulación realizada por Bochs, puede verse muy demorada.

La escritura de caracteres individuales a la consola de Bochs fue utilizada poco en la depuración de PARTEMOS, pero es la base de otras funciones de salida más complejas, contenidas en el módulo "`boshlog.c`", descrito mas adelante.

[1] http://bochs.sourceforge.net/doc/docbook/user/bochsrc.html#AEN2106

Adicionalmente al puerto de trazas, el emulador Bochs brinda la posibilidad de redirigir las escrituras a los puertos serie y puertos de impresora hacia un archivo. Con estas opciones se pueden revisar las trazas de ejecución en los archivos texto generados, por lo que sería recomendable sacar provecho de estas características en implementaciones futuras.

6.5.2 Módulo "conio.c"

El módulo "`conio.c`" contiene fundamentalmente funciones de escrituras a pantalla definidas de la misma forma que en el encabezado "conio.h" brindado por Borland. Este módulo fue creado con el objetivo de poder compilar en 32 bits código existente que dependía de la biblioteca **conio** de Borland. También se creó un archivo "conio.h" dentro de la carpeta `include` de PARTEMOS, que define las funciones implementadas en el módulo, y que permite que los archivos que incluyen este encabezado puedan compilar sin problema.

Este módulo no requiere de inicialización previa ni depende de otros módulos, por lo que era una opción para realizar salidas de depuración en lugares del núcleo en que el administrador de ventanas del núcleo no se podía utilizar por no haber sido inicializado todavía. Sin embargo la nueva versión del administrador de ventanas, que será descrita mas adelante, ya no padece de este problema.

Las funciones de "`conio.c`" no son tolerantes a condiciones de carrera, por lo que la impresión desde varias tareas podría salir entremezclada. Otra limitación de este módulo es que no implementa la función de salida de texto con formato (**`cprintf`**), sino que sólo implementa funciones sencillas para imprimir cadenas de texto (**`cputs`**), caracteres (**`putch`**), y números (mediante funciones no definidas por la **conio** de Borland).

El código de algunas pruebas utiliza la función **`getch`** para realizar esperas por teclado. Aunque esta no es una función de salida, está definida en el encabezado "conio.h" de Borland, por lo que también se realizó una versión de la misma dentro de "`conio.c`". A diferencia de la función original, la nueva versión de **`getch`** no devuelve el código de la tecla oprimida, por lo que sólo sirve para realizar esperas.

Para hacer que el código de **`getch`** funcione sin depender de drivers de teclado o manejadores de interrupción específicos, su implementación simplemente realiza lecturas del puerto de teclado, en espera de que cambie el código scan. Esta forma de implementación tiene la ventaja de que puede funcionar incluso con interrupciones deshabilitadas, lo que puede ser útil para congelar el sistema en puntos específicos, hasta que el usuario decida continuar presionando una tecla.

Aunque este módulo fue útil para brindar los servicios de **conio** en etapas tempranas de desarrollo en 32 bits, actualmente todo el código del núcleo compilado en 32 bits se encuentra limpio de llamadas a funciones de "conio.h", las cuales se sustituyeron por llamadas al administrador de ventanas del núcleo. Es por esto que el módulo "`conio.c`" se considera prácticamente obsoleto, aunque todavía puede servir para compilar código de algunas pruebas que no han sido actualizadas para usar el administrador de ventanas, y también puede servir para realizar pausas en la ejecución del sistema.

6.5.3 Módulo "boshlog.c"

Este módulo utiliza la salida al puerto de trazas del emulador Bochs, para implementar funciones de salida de mayor nivel que la escritura de caracteres individuales. Las funciones brindadas por este

módulo se limitan a la escritura de cadena de caracteres, la escritura de saltos de línea y la escritura de números en hexadecimal.

Las funciones de este módulo tienen las mismas ventajas y limitaciones que la técnica de escritura de bytes al puerto de trazas de Bochs, vista previamente, ya que se basan en esta técnica. Adicionalmente la escritura de cadena de caracteres con estas funciones no es tolerante a condiciones de carrera, así que la impresión desde varias tareas podría salir entremezclada.

Una desventaja que se ve ampliada con el uso de estas funciones es la demora introducida por las funciones de salida, ya que ahora se escriben cadenas de caracteres completas. En una prueba realizadas con gran número de salidas insertadas dentro del núcleo (para rastrear errores), la máquina emulada por Bochs pasaba tanto tiempo escribiendo trazas que prácticamente no se ejecutaba ninguna tarea en el sistema salvo el driver de reloj (por tener la mayor prioridad). Para reducir el problema anterior se pueden quitar las funciones de salida de lugares donde no sean necesarias para rastrear un error, y además se puede reducir la frecuencia de interrupción del driver de reloj.

Como el módulo "boshlog.c" está definido solamente para la arquitectura IA-32, no se recomienda usar las funciones definidas en el mismo directamente dentro del código en depuración, ya que el código resultante no podría compilar en otras plataformas. Para hacer el código compilable en varias plataformas sin tener que comentar o cambiar las funciones de salida, se recomienda utilizar las macros de emisión de trazas, que se explican mas adelante, y pueden indirectamente hacer uso de este módulo.

6.5.4 Administrador de ventanas "screen.c"

El administrador de ventanas "`screen.c`" es un módulo de software creado para PARTEMOS, cuya función es permitir la salida en modo texto en pantalla por parte de varias tareas. La escritura en pantalla se realiza en zonas rectangulares de la misma, denominadas **ventanas**. La escritura en diferentes ventanas puede realizarse concurrentemente sin que una tarea estorbe ni detenga la ejecución de la otra.

El administrador de ventanas fue creado para su utilización en PARTEMOS16, y su código original tuvo que sufrir varios cambios para poder compilar con GCC para la arquitectura IA-32. Los apuntadores a memoria de video fueron definidos condicionalmente según la plataforma, y las rutinas de impresión de números fueron modificadas para funcionar independientemente del tamaño de los mismos. El nuevo código obtenido del administrador de ventanas debe poder compilar con pocos o ningún cambio para otras arquitecturas.

Otro problema que presentaba el administrador de ventanas de PARTEMOS está relacionado con la implementación de **`wprintf`**, que es una función estilo **`printf`** para salida a ventanas. La forma de uso de parámetros variables en esta función hacía posible utilizar la misma dentro de la rutina de salida del núcleo **`kprintf`**, sin embargo el compilador GCC no admitía esa forma de uso de parámetros variables, por no ser compatible con todas las arquitecturas. Ahora la función **`wprintf`** utiliza una forma de acceso a parámetros variables portable a cualquier plataforma, pero la función **`kprintf`** dentro del núcleo tuvo que ser remplazada por una macro denominada **`KPrintf_`**.

Al código del administrador de ventanas también se le hicieron algunos cambios no relacionados con su portabilidad, con el objetivo de mejorar su funcionalidad. Anteriormente, para utilizar el administrador

de ventanas se requería inicializar el mismo (invocando al servicio **`initScreen`**), y posteriormente crear al menos una ventana (invocando a **`CreateWindow`**). Cada ventana creada tiene un identificador de ventana, que debe ser pasado a las funciones de salida. Los pasos iniciales requeridos para comenzar a utilizar el administrador de ventanas hacían imposible su uso en puntos del núcleo anteriores a su inicialización.

El administrador de ventanas fue modificado para permitir la invocación de funciones de impresión incluso antes de la inicialización del módulo, siempre que se use como identificador de ventana el número 0. Con la introducción de la ventana cero (ó ventana por defecto), ahora es posible realizar impresiones a pantalla en etapas tempranas cuando todavía no se ha inicializado el administrador, o impresiones luego que la mayor parte del sistema ha finalizado y todas las ventanas han sido destruidas.

Ahora las funciones de salida del núcleo (definidas en el módulo "KrnlOut.c") utilizan la ventana por defecto para realizar su salida, por lo que también pueden utilizarse sin requerir inicialización.

La ventana por defecto tiene un tamaño fijo que ocupa toda la pantalla, pero puede ajustarse posteriormente si se desea. Para permitir ajustar los tamaños de ventana, se introdujo una nueva función denominada **`ResizeWindow`**.

Con las modificaciones realizadas al administrador de ventanas, ahora es posible utilizarlo para realizar salida a pantalla en códigos compilados para 32 bits, incluyendo código independiente del núcleo PARTEMOS. Actualmente el uso de este administrador es la forma recomendada de salida a pantalla, remplazando al uso de funciones de "conio.c", y es utilizado incluso en código de bajo nivel, como en las pruebas de los módulos de la capa HAL.

6.5.5 Emisión de trazas

Las funciones de salida juegan un papel muy importante en la depuración del núcleo y otros archivos relacionados. Siguiendo la secuencia de impresiones (trazas) realizadas por estas dentro del núcleo es posible descubrir la causa de un comportamiento anómalo, o al menos ayuda a formarse una teoría acerca del problema. Para depurar un error dentro del núcleo muchas veces es necesario instrumentar el código del mismo con las diferentes funciones de salida, lo que puede en algunos casos estorbar en el desarrollo del sistema. La mayoría de las funciones de salida vistas no son portables, por lo que de usarse dentro del núcleo sería necesario comentarlas cada vez que se quiera probar el mismo en otras plataformas. Otras veces podría desearse desactivar temporalmente las impresiones realizadas dentro del núcleo (o parte de ellas), o cambiar el dispositivo de salida (por ejemplo dejar de imprimir en la pantalla, y hacerlo en la consola de Bochs), cualquiera de estas acciones implicaría realizar muchos cambios manuales si se usaran directamente las funciones de salida descritas. Para minimizar los problemas antes mencionados, como parte del presente trabajo se creó **el encabezado de emisión de trazas de PARTEMOS**.

El encabezado de emisión de trazas de PARTEMOS es un archivo ".h" ("`dbgLog.h`") que contiene varias definiciones de símbolos de salida. Estos **símbolos de emisión de trazas** están definidos condicionalmente, lo que permite cambiar las mismas fácilmente para imprimir salida de texto a dispositivos diferentes, o desactivar la salida completamente. Los símbolos de emisión de trazas se dividen en dos grupos: los símbolos de volumen de salida normal, y los de volumen de salida detallado

(*verbose*). Los símbolos de volumen de salida detallado pueden desactivarse individualmente, dejando activos sólo los símbolos de volumen normal.

A cada dispositivo de salida soportado le llamaremos "***tracer***", que podría traducirse como "emisor de trazas". El encabezado de emisión de trazas permite definir un tracer independiente para cada plataforma soportada. En caso de quererse compilar el sistema para una plataforma nueva, es posible crear fácilmente un nuevo tracer para la misma, o indicar que la nueva plataforma usa el tracer nulo (que desactiva la salida). Actualmente existen tres tracers definidos en el encabezado: el **tracer nulo** (desactiva todas las salidas), el **tracer sobre administrador de ventanas** (utiliza el administrador de ventanas para escribir la salida a pantalla, usando la ventana por defecto), y el **tracer para consola de Bochs** (realiza la salida al puerto de trazas de Bochs).

Los símbolos de emisión de trazas generalmente se definen como macros que utilizan otras funciones, pero dependiendo del tracer también pueden definirse como funciones. La tabla 6 muestra los símbolos de volumen de salida normal definidos actualmente.

Tabla 6: Símbolos de emisión de trazas de volumen de salida normal.

Símbolo	Función
DBG_LOG_STR(text)	Envía una cadena al tracer para su impresión
DBG_LOG_NewLine()	Provoca que el tracer imprima un salto de línea
DBG_LOG_HEX(val)	Imprime un número en hexadecimal
DBG_LOG_PTR(addr)	Imprime una dirección (normalmente en hexadecimal)

Los símbolos con volumen de salida detallado son similares a los mostrados en la tabla, cambiando el prefijo "DBG" por "VRB". Por ejemplo, si se desea enviar al tracer la frase "Hola mundo" usando un símbolo de volumen detallado, se debe insertar la siguiente línea:

```
VRB_LOG_STR("Hola mundo");
```

La figura 37 muestra un fragmento de código tomado del inicio del archivo "`dbgLog.h`", que contiene líneas de configuración del encabezado de emisión de trazas. El símbolo **`TRACER`** identifica el tracer que se va a utilizar, y es definido condicionalmente según la plataforma. Las líneas 3 a 5 definen constantes que identifican los diferentes tracers soportados actualmente, mientras las líneas 7 a 14 definen que tracer se usará en cada plataforma. De acuerdo al ejemplo de la figura, al compilar PARTEMOS para la arquitectura IA-32, se utilizará el administrador de ventanas (constante TRACER_SCRDRV, ver línea 11), mientras que para la versión de 16 bits se utilizará el tracer nulo (constante TRACER_NULL, ver línea 9), para desactivar la emisión de trazas. Las restantes líneas comentadas (líneas 8, 12, y 13) definen otros posibles tracers a usar en las diferentes plataformas.

En la figura 37 se muestra una línea comentada (línea 1) definiendo el símbolo **`VERBOSE_TRACE`**, usado para habilitar la salida detallada. Si se define el símbolo en esa posición (descomentando la línea), entonces se habilita globalmente la emisión de trazas con volumen detallado. En caso de desearse la emisión detallada solo en un módulo ".c", se debe definir el símbolo **`VERBOSE_TRACE`** antes de la inclusión de "`dbgLog.h`" dentro de dicho módulo.

```
1  //#define VERBOSE_TRACE
2  /* Posibles "tracers":     */
3  #define TRACER_NULL 0      /* Nulo - ignora las trazas                    */
4  #define TRACER_SCRDRV 1    /* imprime las trazas a ventana por defecto    */
5  #define TRACER_BochsE9 2   /* Trazas a la consola de Bochs (puerto 0xE9)  */
6
7  #ifdef ARCH_AT16                          /* PC-AT en modo real 16 bits */
8    //#define TRACER TRACER_SCRDRV
9    #define TRACER TRACER_NULL
10 #elif defined(ARCH_IA32)                  /* Arquitectura IA-32         */
11   #define TRACER TRACER_SCRDRV
12   //#define TRACER TRACER_BochsE9
13   //#define TRACER TRACER_NULL
14 #endif
```

Figura 37: Sección de configuración de tracer en "dbgLog.h".

Actualmente el núcleo PARTEMOS se encuentra instrumentado con varios símbolos de emisión de trazas, lo que ha permitido la localización de errores en el núcleo. En algunos puntos de código que se ejecutan con mucha frecuencia, como en la conmutación de tarea o el manejador de interrupciones del núcleo, se incluyeron símbolos para volumen de salida detallado.

6.6 Estructura de carpetas

La figura 38 muestra las principales carpetas de primer nivel de la estructura de directorios de PARTEMOS, tal como quedó luego de portarse a 32 bits, con una descripción de las mismas. Algunas carpetas no se muestran debido a que tienen poca relevancia. La carpeta "`gcc32prj`" fue agregada para almacenar archivos de proyecto específicos de la arquitectura IA-32, mayormente scripts de compilación.

Carpeta	Descripción
KERNEL	Raíz de la carpeta de PARTEMOS
BCPRJ	Carpeta de proyectos para Borland C
bolas	Aplicación de prueba
Doc	Carpeta de documentación
gcc32prj	Carpeta de proyectos para GCC
include	Carpeta de encabezados ".h"
SRC	Código fuente (lenguajes C y ensamblador)

Figura 38: Estructura de carpetas de primer nivel en PARTEMOS

La carpeta "`include`" contiene todos los archivos de encabezados ".h" utilizados en el desarrollo de PARTEMOS. La figura 39 muestra las principales subcarpetas dentro de "`include`", muchas de las cuales están relacionadas con carpetas de módulos definidas dentro de la carpeta de fuentes "SRC", cuyas principales subcarpetas se muestran en la figura 40. Se puede notar como ambas figuras muestran una subcarpeta "`HPC/IA32`", agregada para almacenar archivos de acceso al hardware específicos de la arquitectura IA-32. También se puede apreciar como existen varias carpetas de código fuente para la capa HAL, divididas según la plataforma. Otra carpeta agregada fue la carpeta de fuentes "`outIA32`" que contiene archivos con funciones de salida específicas de la arquitectura de 32 bits.

Carpeta	Descripción
include	Carpeta de encabezados ".h"
AAL	Encabezados de la capa de abstracción de actividad (AAL)
CFG	" del módulo de configuración
CLKDRV	" del driver de reloj
HAL	" de la capa de abstracción de hardware (HAL)
HPC	" específicos de acceso al hardware de PC
IA32	" de acceso al hardware específicos para IA-32
MEM	" del administrador de memoria
scr	" del driver de ventanas
TAL	" de la capa de abstracción de tarea (TAL)
XCP	" del módulo de excepciones del núcleo

Figura 39: Principales carpetas de encabezados ".h".

Carpeta	Descripción
SRC	Carpeta de código fuente (archivos ".c" y ".asm")
AAL	Código de la capa de abstracción de actividad (AAL)
CFG	" del módulo de configuración
CLKDRV	" del driver de reloj
HAL4AT16	" de la capa HAL para PC en 16 bits
HAL4IA32	" de la capa HAL para PC en 32 bits (IA-32)
HAL4PC	" de la capa HAL para PC (16 ó 32 bits)
HPC	Archivos específico de acceso al hardware de PC
IA32	Archivos de acceso al hardware específicos para IA-32
KbdDrv	Código del driver de teclado
KRNL	Otros archivos del núcleo PARTEMOS
MEM	Archivos del administrador de memoria
outIA32	Archivos de salida específicos de la arquitectura IA-32
Scr	Código del driver de ventanas
TAL	Archivos de la capa de abstracción de tarea (TAL)
XCP	Archivos del módulo de excepciones del núcleo

Figura 40: Principales carpetas de código fuente.

6.7 Compilación de PARTEMOS

El núcleo PARTEMOS se compila en un solo archivo imagen, enlazado junto con el código de la aplicación a ejecutar. Para generar este archivo de imagen podemos utilizar dos alternativas, la primera consiste en compilar y enlazar todos los archivos necesarios en un solo paso, mientras que la segunda se basa en compilar los archivos de aplicación y enlazarlos contra archivos del núcleo previamente compilados y

encapsulados en una biblioteca de enlazador. Las comandos mencionados en esta sección deben ejecutarse sobre la plataforma Cygwin, la cual debe haber sido previamente configurada como se explica en el Anexo A.

La primera alternativa de compilación del núcleo es útil cuando se está trabajando activamente en el código de este, por lo que se requiere recompilarlo frecuentemente. En este caso se evita el paso de generación de una biblioteca de enlazador, y además se garantiza que siempre se utilizan las últimas versiones de los archivos del núcleo, con lo que se evita errores por uso de versiones desactualizadas. Generalmente cuando se utiliza esta alternativa el código del núcleo se enlaza con una aplicación que prueba funcionalidades de este, en lugar de una aplicación final.

En la carpeta "`gcc32prj`" podemos encontrar varios archivos script que compilan el núcleo y generan el archivo imagen en un solo paso. Por ejemplo el script "`build.sh`" compila todo el núcleo y una aplicación de ejemplo, y "kball.sh" genera de un paso la imagen para la aplicación de ejemplo "bolas". Estos scripts de compilación ensamblan y compilan todo el código del núcleo, y de la aplicación a ejecutar; y enlazan los archivos objeto resultantes en un archivo imagen. Como estos scripts compilan el núcleo junto con una aplicación específica, si se desea compilar una aplicación diferente es necesario modificar los mismos. La alternativa recomendada para hacer esto es crear una nueva copia de uno de los scripts, y luego editarla para remplazar las referencias a los archivos de aplicación existentes por las de la nueva aplicación.

La segunda alternativa para generar un archivo imagen es útil cuando se desea desarrollar una aplicación final para PARTEMOS, y se cuenta con una versión relativamente acabada y estable del núcleo. En esta alternativa todo el código del núcleo está precompilado y encapsulado en una biblioteca de enlazador, por lo que es útil para distribuir el sistema a desarrolladores PARTEMOS que no requieren acceso al código fuente del mismo. El uso de la biblioteca de enlazador hace más sencilla la distribución del núcleo y el desarrollo de aplicaciones, ya que no se requiere distribuir un gran número de archivos objeto, ni listarlos uno por uno al enlazar el sistema. Otra ventaja del uso de la biblioteca de enlazador es que se pueden encapsular muchos archivos objeto que pueden o no ser usados a la hora de enlazar la imagen final, el enlazador utiliza automáticamente los archivos objetos encapsulados que se requieran.

Para generar una biblioteca de enlazador para PARTEMOS, se puede ejecutar el script "`buildLib.sh`", localizado en la carpeta "gcc32prj". Este script es muy similar a los script de compilación antes mencionados, pero solo compila (y ensambla) archivos del núcleo, sin incluir código de aplicación. En lugar de invocar al enlazador, el script "`buildLib.sh`" invoca la aplicación `i586-elf-ar` para generar la biblioteca, como se muestra en la sección 4.3.3. En el próximo capítulo se muestra como utilizar esta biblioteca de enlazador para generar aplicaciones PARTEMOS.

Si se revisan los comandos de compilación dentro de cualquiera de los scripts antes mencionados, se puede notar que no se utiliza ninguna opción de optimización para el compilador (opciones que comienzan por "-O"). Es importante que no se habiliten las opciones de optimización ya que el mecanismo actual de excepciones del núcleo requiere que las variables locales estén almacenadas en la pila, no en registros de CPU.

Los scripts de compilación generan código para la versión del HAL de interrupciones sobre 8259, y tienen comentadas las líneas para generar la versión sobre APIC. En caso de que se desee probar otra versión del INTHAL, se pueden comentar las líneas que hacen referencia a los archivos de la versión usada, y agregar las líneas necesarias para generar la nueva versión. En el capítulo anterior se listan los archivos de cada versión del INTHAL existente.

Una vez creado el archivo imagen, se puede proceder a cargarla y ejecutarla con GRUB, como se explica en el próximo capítulo.

6.8 Depuración de PARTEMOS

El desarrollo de un sistema operativo es una actividad relativamente compleja, durante la cual se pueden introducir errores en el código que a veces son muy difíciles de localizar. Las técnicas de emisión de trazas vistas en la sección 6.5 pueden ser muy útiles para acotar el punto específico donde ocurre el error, pero no son suficientes. Muchas veces se requiere poder consultar el estado de la máquina, revisar los valores de las variables o los registros de la CPU en un punto de ejecución específico, o detectar el momento exacto en que una variable es modificada. Para realizar las acciones antes mencionadas, generalmente es conveniente contar con aplicaciones especiales, denominadas depuradores (*debuggers*).

En el desarrollo de PARTEMOS se cuenta con poderosas herramientas de depuración que se ejecutan sobre MS-DOS[1]. Por ejemplo el depurador "Turbo Debugger" permite depurar directamente el código fuente del núcleo, soporta la inspección detallada de la memoria utilizando nombre de variables, y permite la colocación de breakpoints tanto de software como de hardware. Otra ventaja de realizar depuración en el código de 16 bits es que en esta versión se puede utilizar un módulo INTHAL falso [Martínez-Cortés 2005], que permite repetir exactamente secuencias de interrupciones para detectar un error. Debido a que el código de PARTEMOS es mayoritariamente el mismo para 16 y 32 bits, la depuración del sistema en 16 bits fue una técnica de mucha ayuda, aun cuando los cambios realizados al núcleo se hicieron con la intención de portarlo a la plataforma de 16 bits.

Generalizando, podemos afirmar que aun cuando se esté portando PARTEMOS a una plataforma nueva, a veces es conveniente probar y depurar los cambios en una plataforma ya soportada si esta tiene mejores herramientas de depuración[2]. Por supuesto esto no aplica a secciones de código específicas de la nueva plataforma, o a la depuración de errores que sólo se manifiestan en la nueva plataforma.

Aun con la disponibilidad de poderosas herramientas de depuración para 16 bits, se hizo necesario realizar la depuración del código de PARTEMOS en 32 bits. Esto se debe a que existen secciones de código específicas a la versión de 32 bits (algunas relativamente complejas como los módulos INTHAL y CPUHAL), y muchas veces algunos errores solo se manifestaban en esta versión.

---

[1] Las versiones de Windows para 32 bits pueden ejecutar directamente aplicaciones para MS-DOS, pero en las versiones de 64 bits es necesario utilizar algún software de emulación para ejecutar estas herramientas (como DOSBox).

[2] Nos referimos a fases de desarrollo tempranas cuando la depuración juega un papel importante, ya que la introducción de cualquier cambio en el código del núcleo siempre debe ser acompañada de pruebas en todas las plataformas soportadas.

Para la depuración de PARTEMOS en 32 bits, las máquinas virtuales juegan un importante papel. Dos de los software de máquina virtual usados ("VirtualBox" y "Bochs") soportan depuración, aunque esta se realiza mediante una consola de comandos, lo cual es un poco incómodo y tardado. El emulador Bochs soporta el uso de aplicaciones independientes que permiten acceder al depurador integrado del emulador mediante una interfaz gráfica, lo cual facilita mucho el uso del depurador.

6.8.1 Uso del depurador Peter-Bochs

Peter-Bochs es uno de los "*front-ends*" gráficos para depuración en Bochs más acabados existentes. Este depurador se ejecuta asociado a una instancia del emulador Bochs, que se puede ejecutar, pausar, detener y reiniciar si se desea. La pantalla principal de Peter-Bochs muestra una gran cantidad de paneles con información del estado actual de la máquina virtual, entre los que se destacan un panel con el contenido de la memoria, un panel de instrucciones mostrando código en ensamblador, y un panel que muestra el contenido de los registros de la CPU y la pila de ejecución, entre otros. En el anexo B1 se explica como instalar y ejecutar este depurador.

Peter-Bochs soporta varias facilidades de depuración como la colocación de breakpoints, y la ejecución de instrucciones paso a paso, entre otras. En este documento asumimos que el lector se encuentra familiarizado con las opciones del depurador, para lo cual puede utilizar la ayuda del software, disponible en línea.

Activación del depurador

Una técnica para depurar un error es tratar de detectarlo lo antes posible y congelar el sistema en ese punto, para poder revisar las trazas de ejecución en busca del error. Pero la revisión de las trazas muchas veces no es suficiente para aclarar lo que está sucediendo, a veces se necesita ejecutar instrucciones paso a paso y revisar el estado de la máquina, lo cual requiere pausar la ejecución de la máquina virtual y activar el depurador.

Mientras se está ejecutando una máquina virtual asociada al depurador Peter-Bochs, es posible pausar la misma pulsando un botón en la interfaz del depurador. Sin embargo esta acción puede detener la ejecución del sistema en cualquier lugar, muchas veces alejado de la zona de código de interés. Una alternativa puede ser utilizar la interfaz del depurador para colocar breakpoints en alguna función del núcleo que se quiera depurar, lo que requiere conocer la dirección específica de la función, como se explica en la próxima sección "Trabajo con símbolos". Otras veces es preferible instrumentar el código del núcleo para que provoque la activación del depurador en un punto del código específico.

El emulador Bochs interpreta la instrucción en ensamblador "XCHG EBX, EBX" como un breakpoint, activando el depurador cada vez que detecta dicha instrucción. En el núcleo PARTEMOS la instrucción de breakpoint mencionada es usada en la macro **`BREAK`**, que al ser definida condicionalmente según la plataforma, se puede insertar en puntos de código deseado, sin afectar la portabilidad del código.

También es posible congelar el sistema sin pausar la ejecución de la máquina, como alternativa al uso de la macro **`BREAK`**. Para congelar el sistema de esta forma se puede colocar la máquina en un ciclo infinito con las interrupciones deshabilitadas. Esta técnica tiene la ventaja de que se puede detener el sistema y revisar las trazas y el estado de pantalla sin requerir la presencia de un depurador. Si además se cuenta

con un depurador, se puede revisar el estado interno de la máquina virtual pausando la ejecución de la misma.

Actualmente la técnica de "congelación en vivo" es utilizada por la variante de 32 bits de la macro **ABORT**, lo que permite revisar el estado de la máquina en caso de detectarse un error en el sistema[1]. Otra forma de congelar el sistema es invocar la rutina **getch** dentro una sección con interrupciones deshabilitadas, con la ventaja de que congela el sistema hasta que se presione una tecla. La desventaja de este método es que no es portable, funcionando solo para la versión de 32 bits.

Trabajo con símbolos

Una gran desventaja del depurador Peter-Bochs es que presenta muy poco soporte para trabajar con información simbólica, por lo que toda la depuración se debe realizar inspeccionando código en ensamblador y direcciones de memoria en hexadecimal. Peter-Bochs soporta la carga de imágenes de núcleo compiladas en formato ELF con información simbólica, sin embargo en este proceso desensambla todo el código máquina del sistema, lo que es extremadamente tardado. Es por eso que la opción de cargar archivos en formatos ELF solo se recomienda para archivos muy pequeños, como las pruebas realizadas al INTHAL. Actualmente este depurador sigue siendo activamente desarrollado por su autor, por lo que es posible que versiones futuras brinden mejor soporte para depuración con información simbólica.

Sin embargo muchas veces es necesario conocer en que posición de la memoria se encuentra una variable (para poderla inspeccionar) o donde comienza la ejecución de una rutina específica (por ejemplo para colocar un breakpoint en ella). También es posible que se desee realizar la operación inversa, es decir, dada una dirección conocer el nombre de función o variable que la contiene, y el nombre del módulo donde se encuentra definida. En estos casos la única solución es realizar una búsqueda dentro del mapa de símbolos del sistema.

Un mapa de símbolos (o archivo "map") es un archivo de texto generado por el enlazador, que contiene todos los símbolos públicos disponibles en un sistema y la dirección de memoria donde se localizan. El mapa de símbolos se genera con la opción "`-Map`" del enlazador, como se muestra en la sección 4.3.1. Todos los scripts de compilación existentes en la carpeta "`gcc32prj`" generan un archivo "map" denominado "`kernel.map`".

Para obtener un mapa de símbolos de PARTEMOS completo, puede ser deseable definir la macro **LOCAL** (normalmente definida como "**static**" en el archivo "`global.h`") como vacía. Esta acción provoca que todas las funciones privadas del núcleo queden definidas como públicas, y que por tanto se muestren en el mapa de símbolos. Si no se modifica la macro **LOCAL** puede suceder que una dirección dentro de una función privada parezca pertenecer a otra función (pública) mostrada en el mapa de símbolos. El cambio a la macro **LOCAL** solo debe hacerse para depuración, y posteriormente se debe regresar a su valor original.

[1] La macro "ABORT" es utilizada por las aseveraciones del núcleo, que verifican la consistencia del mismo y abortan el sistema en caso de detectar un error.

Un uso frecuente que se da a los mapas de símbolos durante la depuración del núcleo es conocer en que lugar específico ocurrió un evento, como la activación de un breakpoint, una excepción de depuración, o el fallo de una precondición.

Ejemplo: Rastreo de la pila de llamadas

Esta sección muestra con un ejemplo como obtener la pila de llamadas en un punto de ejecución específico, utilizando el depurador Peter-Bochs y ayudados por el mapa de símbolos del sistema. En este ejemplo se asume que los archivos del sistema se compilaron utilizando el marco de pila estándar (es decir, sin utilizar la opción "`-fomit-frame-pointer`" durante la compilación de los archivos).

En la figura 41 se muestra la estructura de los marcos de pilas creados en una secuencia de llamadas anidadas. El registro EBP es usado para acceder a elementos del marco de pila de la función actual, tanto parámetros como variables locales, y puede considerarse que es un apuntador al marco de pila actual. Se puede apreciar como el registro EBP contiene la dirección en la pila donde se salvó el registro EBP de la función anterior, creando una lista enlazada de apuntadores (marcos de pila). Justo antes de apilarse el registro EBP podemos encontrar apilada la dirección de retorno de la función actual, con la cual se puede identificar que función invocó a la función actual. De esta forma podemos movernos por todos los marcos de pila (siguiendo los apuntadores EBP), e identificar las funciones usando las direcciones de retorno.

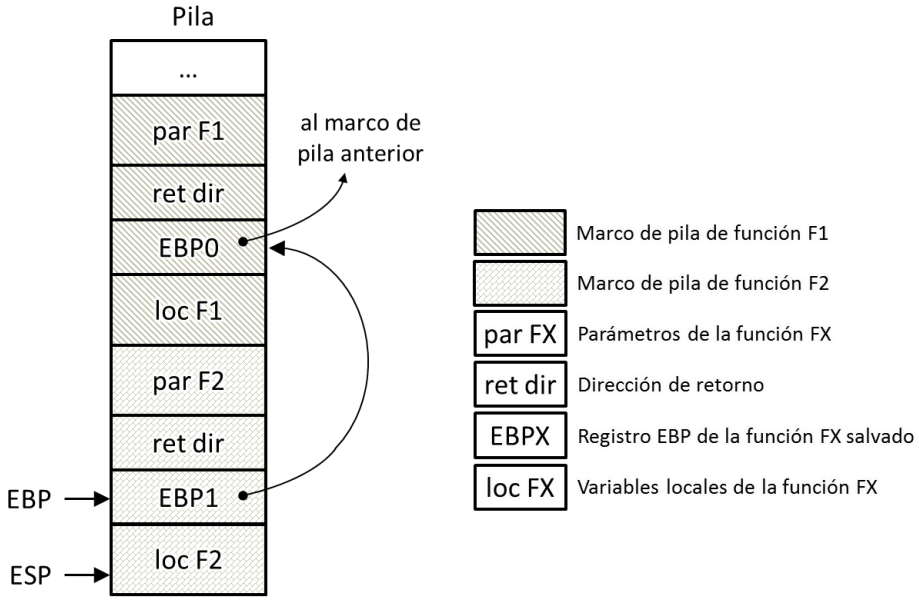


Figura 41: Enlace entre marcos de pila.

Supóngase que la ejecución del sistema se ha detenido en un punto específico, y el depurador Peter-Bochs se muestra como en la figura 42. En este punto es posible inspeccionar el valor de diferentes registros como EIP para identificar la función actual, y EBP para localizar su marco de pila. A continuación se explica como se puede identificar la función que invocó a la función actual.

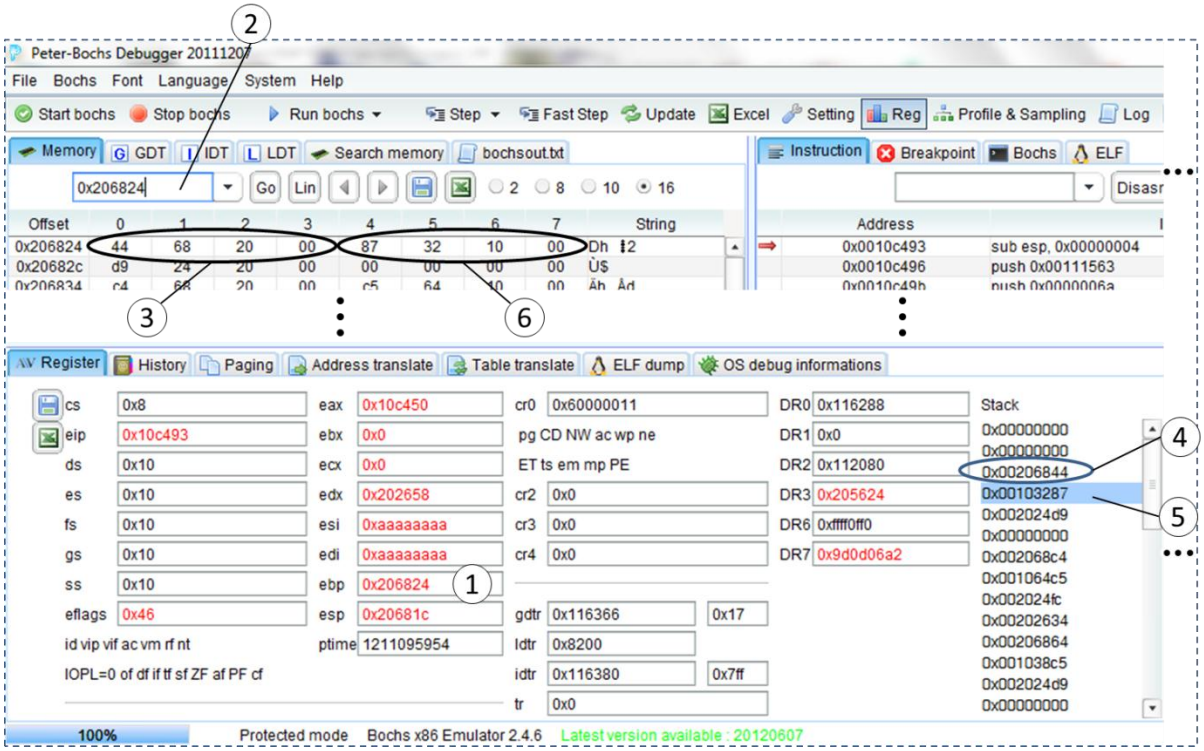


Figura 42: Ventana de Peter-Bochs

El valor del registro EBP (ver ① en la figura 42) identifica el marco de pila actual. Copiando su valor (**0x206824**) en el cuadro de búsqueda ② del panel de memoria podemos obtener el valor almacenado en esta dirección, que se muestra en los cuatro bytes señalados por ③. Leyendo los bytes en orden inverso obtenemos el valor **0x206844**. Para evitar leer los bytes individuales en la pila, es preferible localizar el valor anterior en el panel de pila de Peter-Bochs, en la figura el símbolo ④ indica donde se encontró este valor. Inmediatamente debajo del valor encontrado (símbolo ⑤) encontramos la dirección de retorno de la rutina actual (**0x103287**). Para asegurarnos que estamos en el marco correcto, podemos confirmar en el panel de memoria (ver símbolo ⑥) que los valores coinciden.

Una vez obtenida la dirección de retorno, es fácil localizar quien invocó a la rutina actual buscando en el archivo texto "`kernel.map`", que contiene el mapa de símbolos generado para este ejemplo. La figura 43 muestra un fragmento de "`kernel.map`" donde se resalta la rutina que estamos buscando. La dirección de retorno (**0x103287**) se encuentra entre 0x103240 y 0x10328c, lo que indica que la rutina que invocó a la función actual se llama **`m_krnlChkAddr`**, y está contenida en el módulo "`KRNLMEM.o`".

```
*fill*          0x00102dee        0x2 00
.text           0x00102df0      0xb10 ./libkernel.a(KRNLMEM.o)
                0x00102e05                _initKrnlMem
                0x00102fe7                createPool
                0x00103075                destroyPool
                0x001030cd                krnlAlloc
                0x00103164                m_krnlFree
                0x00103240                m_krnlChkAddr
                0x0010328c                krnlFirst
```

Figura 43: Fragmento de "kernel.map" resaltando la rutina que contiene la dirección 0x103287.

El valor **0x206844**, señalado por ④ en la figura 42 y ① en la figura 44, contiene el apuntador del marco de pila anterior, por lo que podemos seguir un procedimiento similar al mencionado para localizar la rutina que invocó a **m_krnlChkAddr**. A continuación enumeramos los pasos necesarios, usando como guía la figura 44.

1. Buscar el nuevo marco de pila, escribiendo su dirección (**0x206844**) en ②.
2. Obtener su contenido de la zona ③, en el panel de memoria. En este caso el contenido es **0x206864**.
3. Localizar el valor anterior (**0x206864**) en el panel de pila. En la figura se muestra que el valor fue localizado en la zona ④. Nótese que el valor siempre se debe buscar debajo del apuntador al marco de pila anterior ①.
4. Obtener la dirección de retorno debajo del valor localizado, señalada en la figura por ⑤. Siempre debe validarse que encontramos el marco correcto comparando el valor en ⑤ con el de la sección ⑥. En el ejemplo la dirección de retorno obtenida es **0x1038c5**.
5. Localizar la dirección de retorno anterior (**0x1038c5**) en el mapa de símbolos. Para esto se debe buscar la dirección más grande que sea menor a la dirección buscada.

La Figura 45 muestra un fragmento de "`kernel.map`" que contiene la dirección de retorno obtenida de los pasos anteriores (**0x1038c5**). Esta dirección se localiza entre **0x1038b1** y **0x1038ca**, indicando que la rutina **krnlChkAddr** fue la que invocó a **m_krnlChkAddr**, y también está contenida en "`KRNLMEM.o`".

La inspección de los marcos de pila no solo permite identificar las funciones invocadas, sino también sus parámetros[1]. Por ejemplo si sabemos que la función **m_krnlChkAddr** tiene el siguiente prototipo:

```
BOOL m_krnlChkAddr( POOLID pid, MEMPTR blkAddr );
```

Entonces podemos buscar el valor de sus parámetros debajo de la dirección de retorno a su invocadora **krnlChkAddr**. En la figura 44 la zona ⑦ muestra los parámetros pasados a **m_krnlChkAddr**, donde el parámetro **pid** es **0x2024c9** y **blkAddr** es cero[2].

[1] Con un poco de habilidad también se podría revisar los valores de las variables locales.

[2] Este parámetro normalmente no debe ser cero. Las figuras mostradas en esta sección fueron tomadas de una sesión de depuración real, donde se pasaba el valor cero por error, y por eso se hizo necesario rastrear la pila para localizar la zona de error.

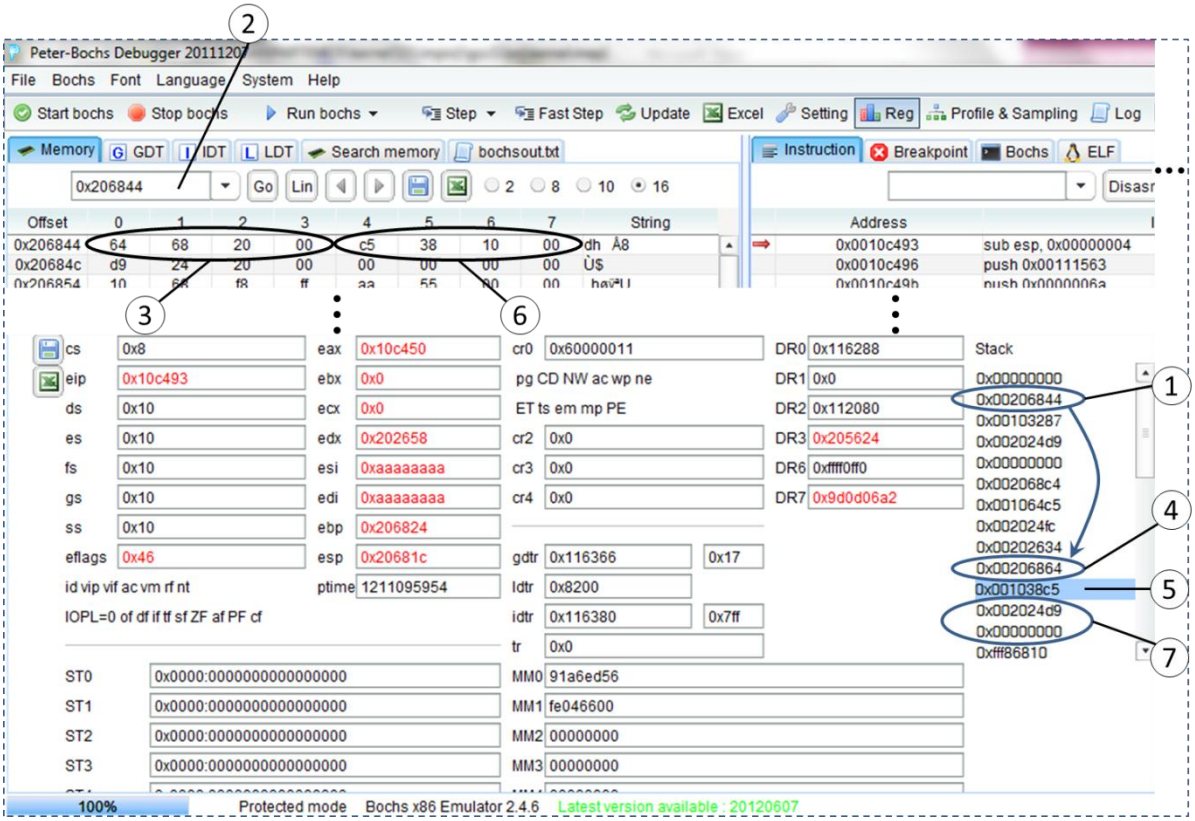


Figura 44: Localizando el siguiente marco de pila con Peter-Bochs

```
.text           0x00102df0      0xb10 ./libkernel.a(KRNLMEM.o)
                0x00102e05            _initKrnlMem
                  ...                   ...
                0x001037fe            krnlFree
                0x001038b1            krnlChkAddr
                0x001038ca            krnlAvlMem
                0x001038ea            taskDump
```

Figura 45: Fragmento de "kernel.map" que contiene la dirección 0x1038c5.

Repitiendo los pasos mostrados anteriormente, usando siempre el apuntador al marco de pila anterior (por ejemplo **`0x206864`** en el último caso) es posible obtener toda la secuencias de llamadas (*call stack*), y obtener los parámetros pasados a cada función.

6.8.2 Uso de registros de depuración

Algunas veces sucede que un error en el sistema causa que una variable (o cualquier zona de memoria) sea modificada indebidamente, pero se desconoce el punto exacto donde ocurrió esta modificación. En estos casos el uso de los registros de depuración del procesador (ver sección 3.1.5) puede resultar de ayuda. Los registros de depuración pueden configurarse de forma que se levante una excepción de depuración al accederse a una zona de memoria deseada.

En la versión de 32 bits de PARTEMOS, el manejador de la excepción de depuración imprime un mensaje de error informativo, y congela la ejecución del sistema (usando la macro **`ABORT`**). El mensaje de error muestra la dirección de código que provocó la excepción, y el valor del registro DR6 para identificar la

causa de la misma. En este punto también puede ser necesario revisar las trazas de ejecución, ó rastrear la pila de llamadas para entender la secuencia de acciones que provocó el problema.

Para facilitar la programación de los registros de depuración se creó el archivo auxiliar "`HPC/IA32/HwDebug.c`", que brinda una función que permite fijar un breakpoint de hardware, con el siguiente prototipo:

```
void setHardBreak(int regNo, void *addr, int type, int length);
```

El primer parámetro (**`regNo`**) es un número entre 0 y 3 que identifica uno de los cuatro registros de dirección DR0-DR3. El parámetro **`addr`** debe contener la dirección de memoria que se quiere supervisar, y en **`type`** se debe pasar que tipo de acción activará el breakpoint (escritura, lectura/escritura, o ejecución). El último parámetro es una constante que identifica el tamaño de la zona de memoria supervisada (puede ser 1, 2, 4, ó 8 bytes). El archivo de encabezados "`HPC/IA32/HwDebug.h`" contiene constantes para identificar las acciones de activación y el tamaño del área de memoria.

A manera de ejemplo, supóngase que tenemos una variable global llamada **`prueba`**, y queremos detectar en que momento la misma es modificada. La figura 46 muestra como se debe proceder en este caso. La línea 1 incluye el archivo de encabezados "`HwDebug.h`", que define las constantes y los prototipos de las funciones de acceso a los registros de depuración. En la línea 4 se activa el breakpoint de hardware, donde el parámetro 0 indica que se usará el registro DR0. El segundo parámetro es la dirección a supervisar, en este caso la dirección de la variable **`prueba`**. La constante BRK_DATA_W indica que el breakpoint solo se activará en caso de escritura a la variable, y BRK_DWORD que el área a supervisar es de 4 bytes (que es el tamaño de un entero). En la línea 6 se está modificando la variable **`prueba`**, por lo que si la ejecución alcanza este punto se generará la excepción de depuración.

```
1 #include <HPC/IA32/HwDebug.h>
2 int prueba;
3 …
4 setHardBreak(0, &prueba, BRK_DATA_W, BRK_DWORD);
5 …
6 prueba = 1; /* lanza la excepción de depuración */
```

Figura 46: Ejemplo de uso de breakpoints de hardware.

Las CPUs con arquitectura IA-32 soportan cuatro breakpoints de hardware simultáneos, ya que poseen cuatro registros de direcciones (DR0-DR3). Sin embargo durante la depuración de PARTEMOS se recomienda utilizar únicamente los registros DR0 a DR2, ya que el registro DR3 se utiliza para verificar desbordamientos de pila. En caso de que 3 breakpoints no sean suficientes, y sea necesario utilizar el registro DR3, es posible desactivar la verificación de pila definiendo el símbolo **`NO_STACK_CHK`** dentro del archivo "`include/stack.h`".

Para desactivar un breakpoint de hardware se puede utilizar la rutina **`disableHardBreak`**, también definida en "`HwDebug.h`", pasándole el número del registro de dirección deseado.

Es importante destacar que no todo el software de máquina virtual soporta el uso de registros de depuración. En el caso específico del emulador Bochs, la versión ejecutable distribuida por los desarrolladores tiene desactivada la verificación de direcciones de depuración, por motivos de velocidad. Se puede recompilar el código fuente de Bochs para obtener una versión del mismo con soporte de

depuración, aunque es posible que esto ralentice grandemente la ejecución del emulador. El software VirtualBox soporta correctamente el uso de la depuración por hardware, sin sufrir problemas de velocidad de ejecución, ya que utiliza la tecnología de virtualización presente en la CPU física.

6.9 Guía para portar PARTEMOS a otras plataformas

Para portar PARTEMOS a otras plataformas, lo primero que se debe realizar es **portar la capa de abstracción de hardware** (HAL), como se describió en el capítulo anterior. Una vez portada la capa HAL, solo quedaría realizar cambios mínimos en algunos archivos ".h", y escribir los drivers para los dispositivos de la nueva plataforma que sean necesarios. Idealmente no debe ser necesario realizar cambios al código fuente del núcleo, aunque en la práctica esto no siempre se cumple. En esta sección describimos los cambios (mínimos) necesarios para portar PARTEMOS, y se discuten algunos criterios a tener en cuenta en caso de requerirse cambios en el núcleo.

6.9.1 Cambios necesarios

Una vez portada la capa HAL, el trabajo que resta para portar PARTEMOS se limita a implementar condicionalmente macros en unos pocos archivos de encabezado ".h", e implementar algunos módulos y drivers de dispositivos requeridos, como se describe a continuación.

Archivos de encabezado ".h"

La mayor parte de los archivos de encabezados ".h" que deben modificarse para portar PARTEMOS forman parte de la capa HAL, ó son archivos que, sin considerarse parte de la capa HAL, tuvieron que ser portados para poder portar el HAL o realizar las pruebas a esta capa. Es por eso que las modificaciones a estos archivos fueron mencionadas en el procedimiento para portar el HAL, descrito en el capítulo anterior.

El único archivo de encabezados que requiere modificaciones para poder compilar el núcleo en otra arquitectura es "`stack.h`", en el cual **deben definirse condicionalmente las macros de manipulación de pila**., cuya función se describe en la sección 6.2. En caso de que no se desee implementar verificación de desbordamiento de pila, se pueden definir vacías las macros "**`setStackChk(val)`**" y "**`disableStackChk()`**".

Debemos mencionar que aunque nos referimos a algunos símbolos como macros, en realidad es posible implementar los mismos como funciones. Para permitir esto, la definición de algunos símbolos incluyen un par de paréntesis, aun cuando no reciban parámetros. Un ejemplo de esto es el símbolo "**`IF()`**", de "`intCPU.h`", que está definido como una macro para la versión de 16 bits de PARTEMOS, pero en su versión para IA-32 se define como una función estática "inline".

Existen macros cuya definición no tiene sentido en algunas plataformas, en cuyo caso debe hacerse una definición vacía, o de ser necesario una definición "tonta" (*dummy*). Por ejemplo la macro **`videoRAMXY`** se usa para acceder directamente a caracteres en memoria de video, pero si la nueva plataforma no soporta salida a video en modo texto entonces habría que desactivar esa macro. Para desactivar esta macro se puede hacer que siempre escriba en un byte *dummy*. Es posible que en versiones futuras de PARTEMOS se elimine completamente el uso de **`videoRAMXY`** en el núcleo.

Al crear versiones de las macros para nuevas plataformas, es necesario tener presente que a veces adaptar una forma de implementación existente puede provocar errores, como sucedió con la implementación de las macros ATOMIC/ENDATOMIC (ver sección 6.1.6)

Drivers de dispositivo

Los drivers de dispositivo disponibles actualmente para PARTEMOS (incluyendo drivers para teclado y puerto serie) pueden compilar sin cambio en las dos plataformas soportadas (de 16 y 32 bits), ya que acceden al mismo hardware. Sin embargo al portar el sistema a otras plataformas, será necesario escribir nuevos driver para los dispositivos físicos que se deseen soportar.

Una excepción importante a lo mencionado anteriormente es el driver de reloj de PARTEMOS. Este driver[1] brinda servicios específicos al núcleo y su código no es dependiente de la plataforma, por lo que no es necesario cambiarlo (a menos que se desee implementar nuevos algoritmos sin modificar su interfaz). Lo único que este driver requiere para funcionar correctamente es que el HAL de reloj (CLKHAL) sea portado correctamente, y que el símbolo **`TIMERIRQ`** identifique el número de IRQ generado por el temporizador del sistema, como se explicó en el capítulo anterior.

Funciones de salida

Es posible que al portar PARTEMOS se desee implementar nuevos métodos de salida, específicos de la nueva plataforma. En este caso los archivos fuente con funciones de salida definidas deben colocarse en una subcarpeta dentro de la carpeta de fuentes de PARTEMOS, cuyo nombre debe comenzar con el prefijo "out". Por ejemplo si se escribe código de salida para una plataforma denominada "XYZ", la carpeta para colocar este código podría llamarse "`outXYZ`".

El uso directo de funciones específicas debe limitarse a código específico de la plataforma, o líneas temporales para depuración. Si se desea utilizar el nuevo método de salida para emitir información de depuración dentro del núcleo, entonces lo recomendado es utilizar los símbolos de emisión de trazas vistos previamente (sección 6.5.5), e implementar un nuevo tracer basado en el nuevo método de salida.

Para crear un nuevo método de emisión de trazas se debe modificar el archivo "`dbgLog.h`", del cual se mostró un fragmento en la figura 37. Lo primero que debe hacerse es insertar un nuevo símbolo identificador de tracer (debajo de la línea 5 en la figura), con un valor diferente de los ya existentes. Posteriormente se debe definir el símbolo **`TRACER`** dentro de una sección dependiente de la plataforma, de forma similar a como se muestra en las líneas 7-14. Por último se deben definir los símbolos de emisión de trazas listados en la tabla 6, utilizando el valor de **`TRACER`** para controlar la compilación condicional.

Si no se desea implementar un nuevo tracer, se puede definir una sección para la nueva plataforma que defina el símbolo **`TRACER`** como **`TRACER_SCRDRV`**, para realizar salida de trazas a la pantalla. De no definirse una nueva plataforma dentro de "`dbgLog.h`", el símbolo **`TRACER`** tomará por defecto el valor **`TRACER_NULL`**, desactivando la emisión de trazas.

[1] Realmente existen varias implementaciones del driver de reloj con diferentes algoritmos internos. Estas implementaciones son intercambiables ya que exportan la misma interfaz.

Módulo de configuración

Como se mencionó en la sección 6.4, el módulo de configuración de PARTEMOS puede ser implementado de muchas formas, y cada forma de implementación depende de características específicas de la plataforma. En caso de desearse compilar una versión de PARTEMOS sin preocuparse de la creación de un módulo de configuración específico, se puede utilizar la versión limitada de los archivos de configuración, descrita en la sección mencionada. La versión limitada del módulo de configuración no contiene código dependiente de la máquina, por lo que debe de compilar sin problemas en cualquier plataforma.

6.9.2 Notas sobre posibles cambios en el núcleo

Idealmente el código del núcleo PARTEMOS resultante de este trabajo no debe necesitar ningún cambio para compilar en otras plataformas. Sin embargo en la práctica puede suceder que para portar el sistema a otra plataforma se requiera realizar alguna modificación en el núcleo. En esta sección se discute un caso conocido de donde puede pasar esto, y se brindan algunas sugerencias de trabajo obtenidas de la actual experiencia de portado.

Un caso conocido donde se puede necesitar modificar el código del núcleo es cuando se quiera portar PARTEMOS a la arquitectura AMD64, y posiblemente a cualquier arquitectura de 64 bits. Algunos compiladores para 64 bits generan código donde los apuntadores y los números enteros no tienen el mismo tamaño (los apuntadores a memoria utilizan 64 bits, mientras los enteros se mantienen en 32), rompiendo con la convención de los compiladores para arquitecturas de 8, 16 y 32 bits existentes. El código actual de PARTEMOS asume en algunos puntos que un tipo entero y un apuntador utilizan la misma cantidad de memoria, por lo que es posible que se requieran cambios en el núcleo para eliminar esta suposición. Se sugiere definir un nuevo tipo de dato entero, cuyo tamaño sea igual al de un apuntador, el cual debe ser usado en todos los puntos donde se requiera asignación entre apuntadores y números enteros.

Aunque la nueva versión de PARTEMOS creada en este trabajo ha sido restructurada para ser lo mas portable posible. Es posible que existan secciones de código que deban incluirse u omitirse para portarlo a una nueva plataforma. En este caso una solución temporal es utilizar compilación condicional dentro del código para implementar las secciones problemáticas. Sin embargo se recomienda que estas secciones sean extraídas hacia archivos de encabezados y encapsuladas como macros creadas condicionalmente, o implementadas en servicios de la capa de abstracción de hardware. Esta **extracción de dependencia de la plataforma** fue realizada en muchas secciones del código original de PARTEMOS, para lograr el código altamente portable disponible actualmente. Como norma general, el código fuente (archivos ".c") de PARTEMOS no debe incluir secciones compiladas condicionalmente según la plataforma, quedando restringido el uso de la compilación condicional a algunos archivos de encabezado ".h".

Entre las excepciones a esta regla tenemos los casos de escritura de código (temporal) de depuración, por ejemplo si estamos trabajando con una versión del INTHAL sobre APIC, puede que en algún lugar del código C se escriba algo como esto:

```
#ifdef USE_APIC
  showAPICinfo();
#endif
```

Esto muestra información de los registros del APIC, útil para depuración. En este caso la compilación condicional impide que se invoque a **`showAPICinfo`** cuando se enlaza el núcleo contra una versión del HAL que no depende del APIC.

En algunos casos también es posible utilizar compilación condicional en archivos de código específicos a la plataforma, aunque estos casos son muy raros precisamente por el hecho de que este código solo se compilará para una plataforma. Un ejemplo de esto son las rutinas de escritura a puerto para las arquitecturas PC, que están definidas condicionalmente dependiendo de si la arquitectura es de 16 o 32 bits, lo que permite que el resto de las rutinas de acceso al hardware de la PC sean las mismas sin importar la plataforma.

Otro caso donde es válido utilizar compilación condicional en un archivo ".c", es cuando se usa código condicional que no es dependiente de plataformas específicas, sino de características generales como el tamaño de la palabra de máquina. Este tipo de compilación condicional se utiliza actualmente en rutinas de impresión en el administrador de ventanas, y en el código de prueba de la CPU, pero no se utiliza dentro del código del núcleo.

6.10 Evaluación de la portabilidad de PARTEMOS

La portabilidad de un sistema es una medida de la facilidad con que se puede escribir una versión del mismo para una plataforma distinta a las soportadas originalmente. La evaluación de la portabilidad es algo muy subjetivo, sin embargo, en esta sección pretendemos dar una idea de cuan portable es el código de PARTEMOS.

Para el caso particular del micro-núcleo PARTEMOS, el código fuente tiene las siguientes características: (1) todos los componentes tienen una "clase de código" de naturaleza bastante similar (código de bajo nivel); (2) han sido escritos utilizando el mismo lenguaje de programación o, en el peor de los casos, lenguajes de un nivel de abstracción similar (medio y bajo); (3) han sido escritos utilizando un estilo de codificación consistente y por un número muy reducido de programadores. Dado estas características, y dado que no pretendemos tener una idea muy precisa de la complejidad de la programación sino sólo darnos una idea del grado de portabilidad del núcleo, podemos considerar que, a estos efectos, en promedio, el esfuerzo de codificación de cada componente guarda una relación con el tamaño del archivo fuente. En consecuencia, podemos utilizar como métrica para evaluar el esfuerzo requerido para codificar un componente de código el tamaño de su archivo fuente (en número de caracteres), incluyendo tanto los caracteres de código como los comentarios (lo cual tiene sentido si consideramos que, en buena medida, el volumen de comentarios guarda una relación con la complejidad del código). Esta medida constituye una forma relativamente sencilla de cuantificar el esfuerzo de codificación de cada archivo de código, para lo cual simplemente se toma el tamaño ocupado por el archivo en disco.

Para evaluar el esfuerzo de codificación y la portabilidad de PARTEMOS, se realizó un listado de los archivos fuentes que forman parte del sistema, como se muestra en el Anexo D. Solo se tuvieron en cuenta los archivos que forman parte del núcleo y sus dependencias, por lo que no se listan los archivos

de aplicación, archivos de prueba, ni drivers de dispositivos (con la excepción del driver de reloj). Cada archivo fue clasificado según la parte del sistema a la que pertenece (HAL, núcleo, o archivos dependientes de la plataforma), y la plataforma de hardware en la que es utilizado (AT16, IA32, o ambas). También fueron clasificados en tres categorías según la portabilidad del archivo: **portable**, **no portable**, y **multiplataforma**. Los archivos portables son aquellos que son utilizados en las versiones de PARTEMOS para las dos plataformas soportadas, y deberían poder utilizarse sin cambio al portar el sistema a otra plataforma. Los archivos no portables contienen código específico de una plataforma, por lo que no pueden ser utilizados en otra[1]. Los archivos multiplataforma son aquellos que deben (o pueden) ser usados en la compilación de PARTEMOS para cualquier plataforma, pero que contienen código condicional dependiente de la plataforma; por tanto los archivos multiplataforma requieren un esfuerzo de codificación a la hora de portar PARTEMOS. Para cada archivo listado se muestra su tamaño en bytes (que estamos usando como medida del esfuerzo de codificación)

Para estimar la dificultad de portar PARTEMOS a una nueva plataforma, introducimos el concepto de **esfuerzo de portado**, definido como el esfuerzo de codificación requerido (o sea, la cantidad de caracteres de código que se deben escribir) para portar PARTEMOS a una nueva plataforma. Evidentemente este esfuerzo depende de la plataforma específica a la que se va a portar el sistema, y no puede saberse con exactitud hasta tanto no se haga el portado real. Solo podemos realizar un cálculo aproximado de este valor basado en la cantidad de código específico existente para cada una de las plataformas actualmente soportadas. Una forma de evaluar la portabilidad de un sistema consiste en comparar el esfuerzo de portado con el tamaño de código que no requiere ser modificado para portar el sistema.

6.10.1 Cálculo del esfuerzo de portado de PARTEMOS

Para estimar el esfuerzo de codificación requerido para portar PARTEMOS a una nueva plataforma, asumimos que la cantidad de código que se debe escribir para portar el sistema debe ser similar a la cantidad de código escrita específicamente para una de las arquitecturas soportadas. Por esta razón usamos como medida del esfuerzo de portado el promedio entre la cantidad de código específico a la arquitectura IA-32 y la cantidad de código específico a la plataforma AT16.

Solo los archivos clasificados como no portables y multiplataforma contienen secciones de código escritas específicamente para alguna de las dos plataformas soportadas por PARTEMOS. Por eso cada uno de estos archivos realiza un aporte al promedio del tamaño de código específico, y por tanto al esfuerzo de codificación que queremos estimar. Este aporte es calculado aproximadamente multiplicando el tamaño de cada uno de estos archivos por un número que denominaremos **Factor de Aporte al Esfuerzo de Portado** (**FAEP**). En la tabla del Anexo D se muestra los valores del factor FAEP y aporte al esfuerzo de portado (columna esfuerzo) para los archivos que aplica. El valor del factor FAEP es una medida aproximada y fue determinado usando alguno de los siguientes criterios:

[1] Existen archivos no portables comunes a las plataformas AT16 e IA-32, lo cual sucede porque ambas plataformas contienen un hardware común. Sin embargo estos archivos no son utilizables en nuevas plataformas.

- Salvo algunas excepciones, los archivos **no portables** fueron escritos específicamente para una de las dos plataformas soportadas actualmente, por lo que su aporte al promedio del esfuerzo de portado es la mitad del tamaño del archivo. O sea tienen FAEP 0.5.
- En la mayoría de los casos consideramos que los archivos **multiplataformas** solo están formados por código no portable, separando condicionalmente las secciones específicas a cada plataforma. Por tanto el FAEP estimado para estos archivos también es 0.5.
- Algunos archivos no portables son comunes a ambas plataformas usadas (debido a que acceden al mismo hardware), por lo que en este caso son tratados como si estuvieran duplicados en cada plataforma, asignándosele un FAEP de 1.
- A algunos archivos específicos de la arquitectura IA32 se les dio un FAEP mayor a 0.5 debido a que implementan características que no existen en la arquitectura de 16 bits, pero que probablemente existan en otras arquitecturas de 32 bits. De esta forma enfatizamos el esfuerzo que se requerirá para hacer algo similar.
- A los archivos del módulo de configuración específicos a la plataforma AT16 se les asignó un FAEP muy pequeño (0.1), ya que existe una versión portable (limitada) de los mismos, y se consideró poco probable que otras adaptaciones de PARTEMOS implementen un módulo de configuración con la complejidad de este.
- El HAL de interrupciones falso [Martínez-Cortés 2005] solo existe para la versión de 16 bits de PARTEMOS, sin embargo es posible realizar una versión del mismo para 32 bits, y debido a su utilidad en la depuración del sistema es muy probable que se desee realizar en nuevas plataformas. Por tal motivo a este archivo se le colocó un FAEP igual a uno, tratándolo como si estuviera duplicado en cada plataforma.
- El administrador de ventanas de PARTEMOS, que fue incluido como parte del núcleo debido que actualmente es utilizado por este, contiene pequeños segmentos de código que usan compilación condicional, motivo por el cual se le asignó un FAEP pequeño (0.1)

En el listado de archivos mostrado en el Anexo D solo se incluyó una versión del módulo INTHAL para IA32, asumiendo que al portar PARTEMOS a una nueva plataforma solo se creará una versión de este módulo. En este caso se utilizó el INTHAL sobre 8259 ya que su codificación es más compleja, y es la única que implementa exactamente el modelo de prioridades de PARTEMOS. Como se mencionó previamente el único driver de dispositivo listado en el anexo es el driver de reloj, que se incluyó como parte del núcleo debido a que el núcleo depende del mismo. No se incluyen otros drivers como el de teclado, ya que el núcleo no depende de estos. Aunque para una nueva plataforma evidentemente será necesario un esfuerzo de codificación para crear nuevos drivers, esto lo consideramos un trabajo extra, independiente del esfuerzo de portar el núcleo.

6.10.2 Análisis de portabilidad

La tabla 7 muestra un resumen de los datos listados en el Anexo D, donde las celdas sombreadas representan cantidades de archivos, mostrando como se distribuyen los diferentes tipos de archivos entre diferentes secciones del sistema. Los archivos listados fueron divididos en dos grupos, de forma que la fila encabezada por la palabra “Núcleo” resume los datos de los archivos pertenecientes al micronúcleo PARTEMOS, mientras la fila con encabezado “HAL y otros” resume los datos de los archivos

de la capa HAL y otros archivos dependientes de la plataforma. Como era de esperar, el código de la capa HAL (y archivos dependientes de la plataforma) contiene muy pocos archivos portables, estando formada mayormente por archivos no portables. Por el contrario, el código del micronúcleo está formado mayoritariamente por archivos portables, lo cual se puede apreciar con más claridad en la figura 47. Los archivos no portables y multiplataforma solo representan el 11% de todos los archivos del núcleo, y si descartamos los archivos no portables (ya que solo son utilizados en la plataforma AT-16), podemos afirmar que para portar el micronúcleo PARTEMOS solo deben modificarse 6 archivos del mismo, lo que es un indicador de la alta portabilidad de este.

Tabla 7: Resumen de archivos de PARTEMOS.

	Portables	No portables	Multi-plataformas	Total de archivos	Código portable (bytes)	Esfuerzo (bytes)
Núcleo	67	2	6	75	591925	12113
HAL y otros	2	28	4	34	972	168888
Total	69	30	10	109	592897	181001

La última columna de la tabla 7 muestra el esfuerzo de codificación estimado para portar diferentes secciones del sistema. Estos esfuerzos se calcularon sumando el aporte al esfuerzo de codificación de los archivos en las diferentes secciones, que se estimaron como se explicó en la sección anterior. A manera de comparación, la columna etiquetada como "Código portable" representa el tamaño total de los archivos portables en las diferentes zonas. Se puede apreciar como el tamaño del código portable dentro de la capa HAL es extremadamente pequeño.

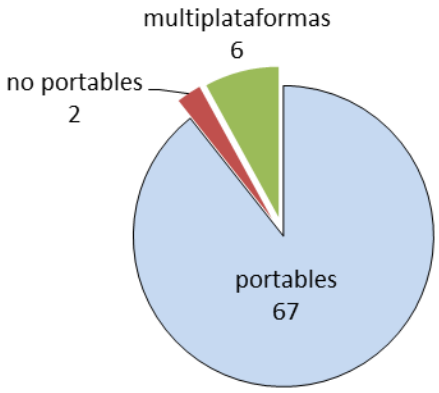


Figura 47: Distribución de archivos del micronúcleo PARTEMOS.

La Figura 48 compara gráficamente el esfuerzo requerido para portar todo el sistema PARTEMOS (incluyendo el micronúcleo y los archivos dependientes de la plataforma) con el tamaño del código portable del sistema. Se puede apreciar como aun cuando este esfuerzo representa una cantidad relativamente grande de código, sigue siendo mucho menor que la cantidad de código portable, donde el esfuerzo de codificación representa un 23% del total del código.

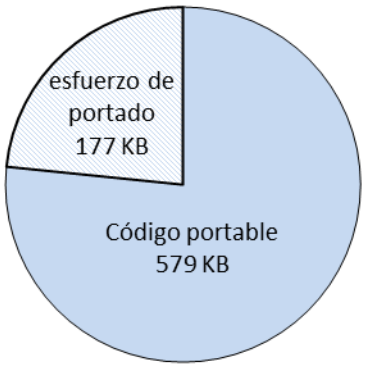


Figura 48: Esfuerzo requerido para portar el sistema PARTEMOS.

El esfuerzo de codificación requerido para portar PARTEMOS está concentrado mayormente en la capa HAL y los archivos dependientes de la plataforma usados por esta capa, requiriéndose muy poco esfuerzo de codificación para portar el micronúcleo. En la figura 49 se puede apreciar como el esfuerzo de codificación en archivos del micronúcleo, requerido para portar el mismo es mínimo comparado con el tamaño de código del micronúcleo, representando tan solo el 2% del código.

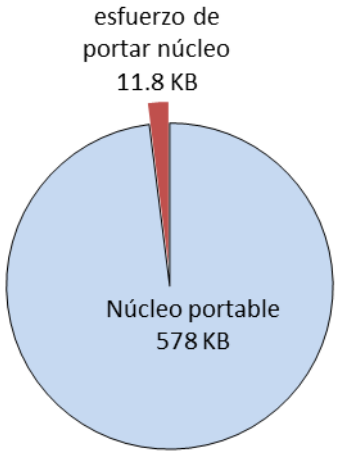


Figura 49: Esfuerzo requerido para portar el código del micronúcleo.

De lo visto anteriormente se concluye que el código de PARTEMOS puede considerarse como **altamente portable**, estando concentrado el mayor esfuerzo de portado en la creación de la capa HAL para una nueva plataforma, con un esfuerzo mínimo requerido para modificar archivos pertenecientes al micronúcleo. La mayor parte del trabajo de adaptación del micronúcleo a otra plataforma consiste en modificar unos pocos archivos multiplataforma. La concentración del código a modificar en unos pocos archivos los hace fácil de localizar y facilita el trabajo de codificación.

Capítulo 7
Desarrollo de aplicaciones en PARTEMOS

PARTEMOS es un micronúcleo experimental que sirve como plataforma de prueba a investigaciones científicas en las áreas de sistemas operativos y de tiempo real. El sistema como tal todavía se debe considerar en fase de desarrollo, faltándole muchos componentes de un sistema operativo acabado, como drivers para muchos dispositivos o módulos para administrar sistemas de archivos. No existe cargador de aplicaciones o de módulos ejecutables independientes, por lo que actualmente toda aplicación que se ejecute sobre este micronúcleo debe enlazarse estáticamente con el mismo. El sistema tampoco brinda una interfaz de alto nivel para el desarrollo de aplicaciones, sino que proporciona un API de bajo nivel (API del micronúcleo) que por el momento es la única disponible al desarrollador de aplicaciones. Sin embargo sobre esta API del micronúcleo se podría implementar en un futuro interfaces más ricas y complejas, como un API compatible con POSIX, las que se ejecutarían en modo usuario.

En este capítulo se brinda una breve descripción del API del micronúcleo PARTEMOS, y se muestra con un ejemplo como se puede compilar y ejecutar una aplicación PARTEMOS para la arquitectura IA-32.

7.1 API de PARTEMOS

El micronúcleo PARTEMOS brinda tres abstracciones fundamentales, además de un grupo de servicios extras. Las abstracciones fundamentales son: **Tarea**, **Objeto de Sincronización** y **Temporizador**. La figura 50 muestra un resumen de las principales abstracciones de PARTEMOS y las operaciones que se pueden realizar sobre ellas.

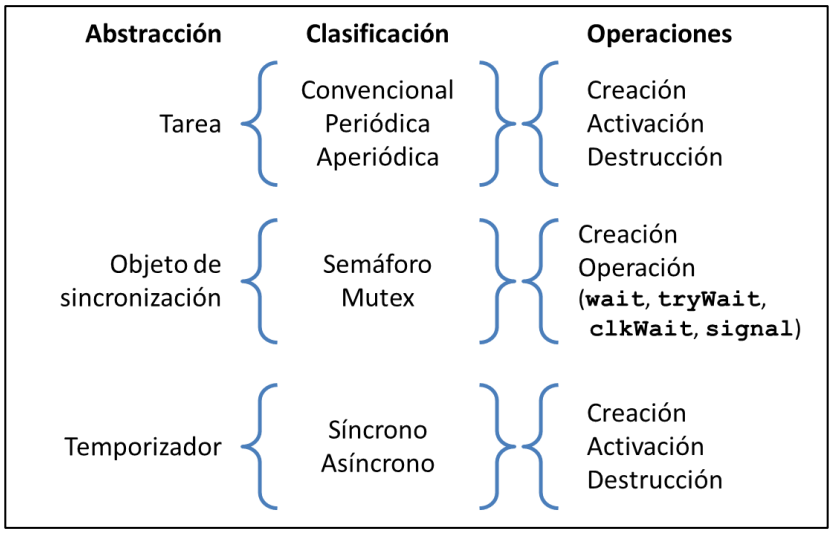


Figura 50: Principales abstracciones de PARTEMOS.

La administración de procesos o tareas constituye el servicio fundamental que tiene que suministrar el núcleo de un sistema operativo. El modelo de tareas de PARTEMOS soporta la creación y destrucción dinámica de tareas. Además, las tareas pueden ser tanto de ejecución continua (un lazo sin fin) como tareas que se ejecutan hasta terminar y pueden volverse a activar de forma periódica o aperiódica.

Internamente, el sistema utiliza temporizadores para activar las tareas periódicas cada cierto tiempo. Las tareas aperiódicas se activan ante una señal en un objeto de sincronización.

Las tareas convencionales se crean mediante la llamada **`createTask`** y se activan de forma explícita a solicitud del programa mediante la llamada a la función **`runTask`**. Las tareas periódicas y aperiódicas se crean de la misma forma, pero se activan a través de eventos (periódicos o aperiódicos) que se indique mediante las llamadas **`makeTaskPeriodic`** y **`makeTaskAperiodic`** respectivamente.

PARTEMOS suministra un mecanismo básico para soportar sincronización y comunicación entre tareas, lo que logra mediante la utilización de semáforos. Para evitar la inversión de prioridad no acotada, fenómeno que impide el establecimiento de garantías de desempeño para las tareas de tiempo real, PARTEMOS brinda semáforos binarios para exclusión mutua (**mutex**), que soportan el protocolo de herencia de prioridad básico. Los objetos de sincronización se crean, se opera sobre ellos y se destruyen. Las operaciones sobre ellos son comunes: **`wait`**, **`signal`**, **`tryWait`** (intento de adquisición sin bloqueo) y **`clkWait`** (intento de adquisición con límite de tiempo al bloqueo).

En el campo del tiempo real es muy común que se necesite poner un límite de tiempo a la ejecución de algunas acciones. PARTEMOS cumple con este requerimiento suministrando los **temporizadores de vigilancia** o **temporizadores perro guardián** (*watchdog*). Una tarea puede activar un watchdog para que el núcleo le notifique cuándo expira el temporizador. En el caso de un watchdog síncrono, esta notificación se realiza sobre un objeto de sincronización, mientras que el vencimiento de un watchdog asíncrono provoca que se efectúe una llamada a una rutina asíncrona. La rutina asíncrona interrumpe la ejecución de la tarea que estableció este servicio, y generalmente realiza acciones de recuperación.

Además de los servicios para supervisar el tiempo de ejecución del código, PARTEMOS también utiliza los temporizadores para brindar servicios de espera temporizada. Estos servicios permiten detener la ejecución de una tarea por un tiempo, que puede ser absoluto (servicio **`delayUntil`**) o relativo al tiempo actual (servicio **`delayFor`**).

Las tareas en PARTEMOS se planifican exclusivamente por su prioridad; o sea, una tarea se ejecuta sólo si no existen tareas de mayor prioridad listas para ejecutarse, además, las tareas de mayor prioridad expropian, en el instante de su activación, a cualquier tarea de menor prioridad que se esté ejecutando (incluso si es una tarea del sistema). Las prioridades de las tareas se asignan en el momento de su creación, y pueden estar en el rango de 1 a 255, donde valores mayores indican mayor prioridad. Las interrupciones también se acogen a este espacio de prioridades. De esta forma el sistema puede tener perfectamente tareas con prioridad mayor que la de **cualquier** interrupción.

7.2 Ejemplo de aplicación PARTEMOS

Para mostrar la metodología utilizada para crear aplicaciones PARTEMOS, se creó una aplicación de ejemplo muy sencilla, a la cual denominamos “Bolas”. La aplicación Bolas muestra una pantalla negra en modo texto con unos caracteres asterisco (“*”) moviéndose y rebotando a manera de bolas. Inicialmente el usuario no verá nada en la pantalla, pero cada vez que presione la tecla “S” aparecerá una nueva bola con dirección y velocidad aleatoria.

La aplicación consta de tres archivos, listados en la Tabla 8. El contenido de estos archivos se lista en el Anexo C.

Tabla 8: Archivos fuente de la aplicación "Bolas".

ball.h	Encabezado de "ball.c". Contiene definiciones comunes a los dos módulos de la aplicación.
ball.c	Módulo que encapsula el comportamiento de cada bola.
MainBall.c	Módulo principal, contiene el código de la tarea principal de la aplicación (la primera que se ejecuta). Administra la interfaz de usuario y utiliza al módulo "ball.c" para crear bolas.

Cada bola está representada por una estructura de tipo **`bolaStruct`** (archivo "`ball.h`", líneas 9-13), que almacena datos como la posición, velocidad y el identificador de una tarea asociada a la bola.

Cuando se crea una nueva bola, también se crea una nueva tarea periódica asociada (archivo "`ball.c`", función **`crearBola`**, líneas 65 y 72), que se encarga de actualizar su posición con cierta frecuencia. El código de esta tarea está contenido en la función **`bolaCod`** (líneas 18-51), y recibe como argumento un apuntador a la estructura bola específica asociada a la tarea. Podemos notar en la línea 9 de "`ball.c`" como se incluye el archivo "`kernel.h`", que contiene las definiciones del API del núcleo usadas en el módulo.

La tarea principal de cualquier aplicación PARTEMOS debe definirse como una función con el nombre **`mainCode`**, en nuestro ejemplo definida en el archivo "`MainBall.c`" (líneas 53-86). El archivo mencionado define un arreglo estático de bolas (línea 19), cuyos elementos serán pasados de uno en uno a la función **`crearBola`** (en la línea 73).

La versión actual de PARTEMOS incluye un sencillo driver de teclado que permite a una tarea bloquearse (sin consumir CPU) en espera de una tecla. La función que brinda esta funcionalidad se llama **`getTcl`** (invocada en la línea 69 de "`MainBall.c`"), y devuelve el código de exploración de la tecla presionada.

La tarea principal también es la encargada de actualizar la pantalla, lo cual normalmente no podría realizar mientras está bloqueada esperando una tecla. Para evitar este problema, establece previamente un manejador de señal asíncrona (línea 66 de "`MainBall.c`"), cuya acción (función **`rutinaAsync`**, líneas 44-48) es actualizar la pantalla. La lectura del teclado se incluye en una sección con tiempo supervisado (líneas 68-70), de forma que si la tarea principal se bloquea por mucho tiempo, se desbloquee temporalmente para ejecutar la rutina asíncrona y actualizar la pantalla.

Para evitar la lectura de valores de posición erróneos debido al acceso concurrente a los datos por parte de la tarea principal y las tareas asociadas a las bolas, se utiliza el semáforo de exclusión mutua **`mutex1`**. Con **`mutex1`** las tareas crean secciones de código críticas (con acceso mutuamente exclusivo) para acceder a los valores de posición de cada bola.

Debemos señalar que el objetivo buscado con esta aplicación fue crear un ejemplo que utilizara algunas funcionalidades del API de PARTEMOS, que tuviera más de un archivo fuente y que al mismo tiempo se mantuviera relativamente sencilla. **No se buscó crear un diseño óptimo** o eficiente desde el punto de

vista de consumo de recursos, por lo que la aplicación resultante podría ser objeto de varias críticas. Entre los señalamientos que se le pueden hacer a esta aplicación tenemos:

- Sección crítica demasiado grande: La tarea principal bloquea las restantes tareas de la aplicación mientras está actualizando la pantalla. Se podría hacer la sección crítica más pequeña y utilizar un mutex diferente para cada bola.
- Se podría evitar el uso de manejadores de señales asíncronas si la lectura por teclado y la actualización de pantalla se hicieran en 2 tareas independientes.
- Número excesivo de tareas: una sola tarea podría encargarse de actualizar todas las bolas, e incluso hacer la actualización de pantalla en el mismo hilo de ejecución.

La aplicación se creó para usar una frecuencia de temporizador de 100 tics por segundo. El uso de frecuencias diferentes provocaría que las bolas se muevan con una velocidad diferente a la original. La frecuencia del temporizador se puede modificar en el archivo de configuración.

7.3 Compilación de la aplicación

A continuación se muestra cómo se puede compilar y ejecutar la aplicación de ejemplo “Bolas”, descrita en la sección anterior, usando la arquitectura IA-32 como plataforma de destino. Se asume que se ha creado previamente la biblioteca de enlazador del núcleo PARTEMOS, como se explica en la sección 6.7 “Compilación de PARTEMOS”. Igualmente, los pasos que siguen deben ejecutarse en la ventana de comandos de Cygwin, que debe estar correctamente configurado (ver Anexo A).

Para construir la aplicación “Bolas” debemos compilar cada archivo fuente por separado, y luego enlazarlos junto con la biblioteca de enlazador de PARTEMOS. En lo adelante se asume que la biblioteca de PARTEMOS se llama “`libkernel.a`”, y se encuentra en la carpeta de compilación.

Lo mas conveniente para crear la aplicación es incluir todos los comandos necesarios en un script de shell (archivo .sh), de esta forma nos evitamos rescribirlos cada vez que deseemos recompilar la aplicación. La figura 51 muestra un ejemplo de como puede ser este archivo para el caso de la aplicación “Bolas”.

```
# Script para compilar la aplicación "Ball"
# junto con el Kernel de PARTEMOS

echo Borrando archivos objeto y temporales
rm *.o
rm *.i
rm *.s

echo Compilando
export INCL="-I../include -I/usr/local/cross/lib/gcc/i586-elf/3.4.4/include"
export OPTIONS="-Wall -fstrength-reduce -g -finline-functions -nostdinc -fno-builtin"

i586-elf-gcc $OPTIONS $INCL -c -o MainBall.o ../bolas/MainBall.c
i586-elf-gcc $OPTIONS $INCL -c -o ball.o ../bolas/ball.c

echo Enlazando

i586-elf-ld -static -T link.ld -Map kernel.map MainBall.o ball.o -L. -lkernel -o bolas.bin
```

Figura 51: Script de compilación de la aplicación “Bolas”.

El script mostrado realiza la compilación de los dos módulos de la aplicación en las líneas 13 y 14, dando como resultado los archivos objeto "`MainBall.o`" y "`ball.o`", respectivamente. Para evitar líneas de comandos excesivamente largas, y tener que repetir valores en cada comando de compilación, se han definido dos variables de entorno `INCL` y `OPTIONS` (líneas 10 y 11). Las variables se incluyen precedidas por un signo "$" en cada comando de compilación.

La variable `INCL` define las rutas donde el compilador busca los archivos de encabezado ".h". En el ejemplo se han incluido la ruta relativa a la carpeta de encabezados de PARTEMOS ("`../include`"), y la ruta absoluta a la carpeta "include" del compilador cruzado (realmente no es necesaria en esta aplicación, pero pudiera utilizarse en otras).

La variable `OPTIONS` define las opciones de compilación usadas. Para conocer más de estas y otras opciones se puede consultar cualquier referencia del compilador GCC.

La última línea del script contiene el comando para enlazar nuestros archivos objeto junto con el núcleo PARTEMOS, para formar el binario "`bolas.bin`", de forma similar a como se explicó en la sección 4.3.3 "Creación y uso de bibliotecas de enlazador".

En caso de que se quiera compilar cualquier otra aplicación PARTEMOS, se puede utilizar un script similar al anteriormente mostrado, remplazando los archivos a compilar (líneas 13 y 14) por los de la nueva aplicación. También se tienen que poner los nombres de los nuevos archivos objetos generados en la línea de enlace (línea 18), y probablemente remplazar el nombre "`bolas.bin`" por un nombre de imagen acorde a la nueva aplicación.

7.4 Ejecución de la aplicación

Para ejecutar la aplicación ya compilada, es necesario ponerla en un disco de arranque que incluya el cargador GRUB. Si la aplicación se piensa ejecutar sobre una máquina virtual, se recomienda usar una imagen de disco flexible que tenga el cargador GRUB instalado. Existen muchas versiones de estas imágenes disponibles para su descarga desde internet, y también se pueden encontrar en el CD de instalación del entorno de desarrollo, que acompaña al presente documento de tesis. En el Anexo B se brinda información adicional acerca del cargador GRUB y del trabajo con discos virtuales.

Una vez iniciada la máquina utilizando el disco GRUB, el próximo paso es cargar la imagen de la aplicación a ejecutar en memoria. Asumiendo que la imagen de nuestra aplicación "`bolas.bin`" se copió en la raíz del disco de arranque, el comando para cargarla en memoria es:

```
kernel /bolas.bin
```

Este comando se escribe en una consola mostrada por el cargador GRUB. Por último, para lanzar la aplicación se debe escribir el siguiente comando:

```
boot
```

Alternativamente, se puede crear un archivo de configuración para indicarle a GRUB que cargue automáticamente la imagen del núcleo. Para obtener más información de como crear un archivo de configuración para GRUB, se puede consultar la documentación en línea del cargador.

Capítulo 8
Valoración del trabajo realizado

En este capítulo se realiza una valoración crítica de la versión de PARTEMOS obtenida con el presente trabajo de tesis. Se resaltan los logros y limitaciones del trabajo realizado, diferenciándolas de las ventajas y limitaciones inherentes a la versión actual de PARTEMOS, y que no se pueden atribuir al presente trabajo de tesis. También se brindan sugerencias de como se pudiera continuar trabajando en un futuro para mejorar el sistema obtenido.

8.1 Valoración del trabajo de portabilidad de PARTEMOS

En esta sección se describen los logros obtenidos durante el presente trabajo de tesis, así como las limitaciones que se pueden atribuir al trabajo realizado. Por último se brindan recomendaciones para dar continuación al presente trabajo, y sobre como pudiera mejorarse el sistema obtenido.

8.1.1 Logros del trabajo realizado para la portabilidad de PARTEMOS

En este trabajo se logró implementar un núcleo de sistema operativo sobre la arquitectura IA-32, funcionando en modo protegido de 32 bits. Este núcleo constituye la primera versión multiplataforma de PARTEMOS, que fue denominada PARTEMOS32.

Para poder diseñar y realizar la implementación de PARTEMOS32, fue necesario asimilar conocimiento sobre el funcionamiento de la arquitectura IA-32. Parte de ese conocimiento quedó plasmado en el presente documento de tesis, aunque debido a la complejidad de la arquitectura solo se abordaron temas relacionados con la implementación de PARTEMOS, y se indicaron las referencias adecuadas para profundizar en el funcionamiento de esta arquitectura.

La arquitectura IA-32 brinda facilidades para la depuración, cuyo uso dentro del núcleo de PARTEMOS puede considerarse un logro en si mismo. Se implementó un módulo “C” que brinda funcionalidades para programar los registros de depuración de la CPU, que puede ser utilizado para colocar breakpoints de hardware en PARTEMOS y otros sistemas. La versión de 32 bits de PARTEMOS brinda soporte limitado a los breakpoints de hardware, mediante la implementación de un manejador para la excepción de depuración. Los breakpoints de hardware, combinados con los mecanismos de emisión de trazas y un depurador adecuado, pueden constituir una herramienta clave en la localización de errores. Los breakpoints de hardware también son utilizados en PARTEMOS para la arquitectura IA-32 (sin afectar la portabilidad del sistema) como un mecanismo sencillo y eficiente para detectar el desbordamiento de pila.

También se asimiló conocimiento sobre el funcionamiento y programación de los controladores APIC, mostrándose las limitaciones de estos controladores para proporcionar el modelo de prioridades de PARTEMOS. Se implementó un módulo reutilizable con servicios de acceso al hardware APIC, que fue utilizado en la programación de versiones del INTHAL sobre APIC.

Uno de los logros más importantes, relacionado con la implementación de PARTEMOS en 32 bits, fue la obtención de un entorno para el desarrollo de sistemas operativos sobre la arquitectura IA-32; el cual puede utilizarse en el desarrollo de otros sistemas operativos, y no solo de PARTEMOS. Formando parte

de este entorno se encuentran todas las herramientas usadas en el desarrollo de PARTEMOS, incluyendo herramientas de compilación, depuración, ejecución, entre otras; y las técnicas de uso de estas herramientas. También se describe el procedimiento para instalar este entorno en la máquina de desarrollo.

El código de PARTEMOS sufrió una restructuración para abordar las deficiencias de portabilidad que presentaba. Algunas secciones de código no portables fueron rescritas para lograr que puedan compilar en cualquier plataforma, mientras otras secciones dependientes de la plataforma fueron movidas a la capa HAL. Por último, algunas secciones de código, dependientes de la plataforma y el compilador usado, fueron extraídas del código C del núcleo PARTEMOS y encapsuladas en macros, definidas en archivos de encabezado. Como resultado, se obtuvo un código del núcleo altamente portable, donde los cambios necesarios para portarlo a otra plataforma están concentrados en unos pocos archivos ".h". En la sección 6.10 se realiza una evaluación de la portabilidad de PARTEMOS.

El sistema también sufrió cambios no relacionados con las deficiencias de portabilidad presentes en PARTEMOS16. Por ejemplo la simplificación de la interfaz del INTHAL hizo que la implementación de este módulo sea más sencilla, y permite que el modelo de prioridades de PARTEMOS pueda ser soportado directamente por un hardware controlador de interrupciones más sencillo y fácil de implementar.

Con la introducción de la conmutación de contexto ligera se redujo la latencia de conmutación entre tareas. Los bloques de control de tarea dentro del núcleo también se redujeron de tamaño, tamaño que ahora es independiente de los registros salvados al conmutar de contexto. La conmutación de contexto ligera, unido a una restructuración en el funcionamiento de los procedimientos asíncronos, permitió mejorar la eficiencia de invocación de estos procedimientos.

El soporte de emisión de trazas permite revisar las trazas de ejecución del núcleo para rastrear errores en el sistema. Esto es posible gracias al uso del encabezado de emisión de trazas, y a la instrumentación del código del núcleo con símbolos de emisión de trazas.

8.1.2 Limitaciones del trabajo realizado

Debido a limitaciones de tiempo, algunos componentes que no eran imprescindibles para la ejecución de PARTEMOS no fueron portados a la arquitectura IA-32. Entre estos componentes podemos mencionar módulos alternativos del administrador de memoria, y versiones alternativas del driver de reloj. El HAL de interrupciones falso [Martínez-Cortés 2005], que puede ser útil para la depuración del sistema, y que existía desde PARTEMOS16, fue adaptado para soportar la nueva interfaz del INTHAL en 16 bits, pero no se creó una versión de 32 bits del mismo. Tampoco se crearon versiones de 32 bits de los módulos de medición de desempeño [Martínez-Cortés 2005], que permiten el registro temporal de eventos con el objetivo de realizar mediciones de desempeño.

Aunque se implementó una versión funcional del módulo INTHAL sobre controladores APIC, debido a características técnicas de estos controladores el módulo implementado no proporciona el modelo exacto de prioridades de PARTEMOS. Quedó propuesta una alternativa de implementación del INTHAL que debe ser capaz de brindar este modelo, aunque la implementación de esta alternativa puede ser compleja y computacionalmente costosa.

La versión actual de PARTEMOS sobre la arquitectura IA-32 no hace detección de la memoria disponible en el sistema, lo que limita al micronúcleo a una cantidad de memoria fija, establecida por software.

El entorno de desarrollo obtenido no cuenta con un depurador con información simbólica, lo que hace la depuración del sistema mucho más compleja. Para depurar el sistema, el desarrollador debe interpretar código en lenguaje ensamblador, y posiciones de memoria sin nombres asociados; auxiliado posiblemente del mapa de símbolos del sistema. Los depuradores sobre máquinas virtuales usados en el entorno de desarrollo están en evolución y mejora constante, por lo que es posible que esta deficiencia se elimine en un futuro.

8.1.3 Recomendaciones

La versión de PARTEMOS sobre la arquitectura IA-32, obtenida como resultado del presente trabajo, fue ejecutada y depurada fundamentalmente sobre máquinas virtuales. El sistema también fue ejecutado y comprobado sobre máquinas físicas, pero sin posibilidad de depuración del mismo. Aunque son innegables las ventajas que brinda el uso de las plataformas virtuales para la depuración de un sistema operativo, en algunas circunstancias pudiera ser deseable contar con facilidades de depuración del sistema ejecutándose en máquinas físicas, por lo que esta es una de las líneas de trabajo sugeridas para mejorar la facilidad de desarrollo en PARTEMOS32. Las facilidades de depuración sobre máquinas físicas generalmente usan una conexión por cable (típicamente serie) con una máquina de depuración. La máquina que ejecuta PARTEMOS podría utilizar el cable para enviar trazas de ejecución, o recibir comandos de depuración de la máquina remota. Los comandos de depuración podrían seguir un protocolo propio o el de algún depurador existente (como GDB).

El uso de puertos de comunicación (serie o paralelo) también puede ser útil para envío de trazas en máquinas virtuales, ya que algunos simuladores de plataformas de ejecución (como Bochs) permiten registrar los caracteres enviados a estos puertos en archivos de la máquina física. Estos archivos pueden ser estudiados posteriormente para intentar localizar la causa de algún error. Se sugiere la creación de nuevos tracers (ver sección 6.5.5) que utilicen estos puertos.

En este trabajo se propuso una variante de implementación del INTHAL sobre APIC que brinda el modelo de interrupciones de PARTEMOS sin enmascaramiento virtual. Esta propuesta no fue implementada debido a la complejidad de la misma, y al costo en operaciones de entrada/salida que su uso conllevaría. Se sugiere que en un futuro se implemente el modelo propuesto, con el fin de validar su factibilidad de implementación y que se realicen mediciones de tiempo para comparar su desempeño con el de otros módulos INTHAL de 32 bits implementados.

Se propone que en trabajos futuros se utilice el cargador GRUB para detectar la memoria disponible en el sistema, con lo que se eliminaría las limitaciones del HAL de memoria usado actualmente en la versión de 32 bits de PARTEMOS.

Se sugiere la implementación de un HAL de interrupciones falso para la arquitectura iA-32, siguiendo a misma idea del INTHAL falso existente para 16 bits [Martínez-Cortés 2005]. Este módulo brindaría la misma interfaz del INTHAL verdadero pero no notificaría al núcleo interrupciones verdaderas sino aquellas que se le indiquen mediante el teclado. Este módulo puede resultar muy útil para repetir secuencias de interrupción, con la intención de reproducir situaciones de error durante la depuración del

sistema. El INTHAL falso para 16 bits brinda otras facilidades, además de la posibilidad de enviar interrupciones falsas por teclado, descritas en [Martínez-Cortés 2005]; entre estas funcionalidades tenemos la posibilidad de congelar la ejecución del sistema, y un servicio que permite simular la ocurrencia de interrupciones de forma síncrona. Se sugiere agregar al INTHAL falso (tanto a la versión de 16 bits existente como a otras posibles versiones para otras plataformas) la posibilidad de activar y desactivar el envío de interrupciones verdaderas al núcleo, tanto por teclado como programáticamente.

Los procesadores con arquitectura IA-32 son muy conocidos y abundantes, siendo los que dominan el mercado de las computadoras personales; por lo que la obtención de una versión de PARTEMOS sobre estos es un logro importante en términos de la difusión futura del sistema. Sin embargo, estos procesadores tienen menor presencia en el mundo de los dispositivos embebidos, donde existe una gran variedad de plataformas de hardware que utilizan diferentes tipos de microprocesadores, microcontroladores, y procesadores de señales digitales. Es por esto que, si se desea que un RTOS tenga éxito en este mercado, debe ser portado a un gran número de plataformas. Se sugiere que, en un próximo trabajo de portado de PARTEMOS, se realice una versión del mismo para ejecutarse sobre procesadores ARM, ya que los mismos están altamente difundidos entre los dispositivos embebidos actuales, y existen muchas herramientas para desarrollo incluyendo simuladores de plataformas que utilizan estos procesadores.

8.2 Valoración de la Versión actual de PARTEMOS

En esta sección se valora la versión actual de PARTEMOS, resultado del presente trabajo, mencionando ventajas y limitaciones de este sistema que no son atribuibles al presente trabajo de tesis.

8.2.1 Ventajas de la versión actual de PARTEMOS

La nueva versión de PARTEMOS mantiene las principales ventajas de su predecesora PARTEMOS16. Entre las que podemos mencionar una fácil asimilación del API del micronúcleo por parte de los programadores de aplicación, debido a su reducido número de abstracciones. Al mismo tiempo estas abstracciones brindan una gran flexibilidad, siendo suficientes para implementar cualquier API de alto nivel. El sistema brinda características útiles para sistemas de tiempo real, como la posibilidad de limitar el tiempo de ciertas operaciones, y el uso del protocolo de herencia de prioridad para evitar la inversión de prioridad no acotada.

Una de las principales de PARTEMOS sobre otros sistemas es su novedosa forma de manejar las interrupciones de hardware, que garantiza el determinismo temporal del sistema incluso en situaciones de sobrecarga temporal de interrupciones. Esto hace que PARTEMOS sea una buena opción para implementar sistemas de tiempo real duro.

Desde su concepción, PARTEMOS fue diseñado para ser portable. Esta ventaja se ve ampliada con las modificaciones realizadas en el presente trabajo, que resultaron en la primera versión multiplataforma de PARTEMOS, y en un aumento de la facilidad de portado del mismo.

8.2.2 Limitaciones de la versión actual de PARTEMOS

Debido a que PARTEMOS es un micronúcleo experimental, que todavía se encuentra en fase de desarrollo, la versión actual adolece de muchas limitaciones que dificultan su uso en entornos

productivos. PARTEMOS carece de muchos componentes presentes en otros sistemas operativos, como sistemas de archivo, soporte de protocolos de red, interfaz gráfica de usuario, y muchos drivers de dispositivo.

Actualmente no se soporta la carga de módulos de software independientes, por lo que las aplicaciones escritas para PARTEMOS deben enlazarse con el núcleo, lo que limita la flexibilidad y usabilidad del sistema. El enlace entre las aplicaciones y servicios del núcleo es estático, en lugar de utilizarse algún tipo de indirección (como trampas de CPU) para el acceso a los servicios, lo cual facilitaría la carga dinámica de módulos.

El desarrollo de aplicaciones PARTEMOS también se ve dificultado debido a la ausencia de un entorno de desarrollo para aplicaciones, y a que no existen versiones de las bibliotecas estándares de C para desarrollo de aplicaciones. El API del micronúcleo es de muy bajo nivel, y no brinda la posibilidad de usar cadenas de caracteres para nombrar abstracciones. La ausencia de un depurador de aplicaciones que se ejecute sobre PARTEMOS también puede dificultar el desarrollo de estas.

PARTEMOS tampoco aprovecha características presentes en muchos procesadores modernos. En especial el sistema no hace uso de los mecanismos de protección de memoria para crear aislamiento entre tareas, y para proteger al núcleo de accesos incorrectos. El núcleo PARTEMOS tampoco soporta multiprocesamiento, desaprovechando las facilidades multinúcleo presentes en muchos procesadores modernos.

El sistema no proporciona soporte a las unidades de punto flotante de los procesadores con arquitectura IA-32[1]. Estos procesadores brindan la posibilidad de detectar el uso de las unidades de punto flotante por parte de diferentes tareas, lo que permite que los sistemas operativos salven los registros de punto flotante usados por las tareas solo cuando sea necesario. Sin embargo PARTEMOS no aprovecha estas características, por lo que actualmente solo una tarea en el sistema puede hacer uso de operaciones de punto flotante.

El código del micronúcleo PARTEMOS presenta ciertas deficiencias que deben ser corregidas en próximos trabajos. Por ejemplo el administrador de memoria actual sólo puede inicializarse con un bloque único de memoria libre, cuando típicamente la memoria física libre en las computadoras actuales se encuentra fragmentada. El diseño actual del mecanismo de excepciones del núcleo impide la compilación con optimización, lo cual reduce la eficiencia del código compilado resultante. Algunas versiones del INTHAL no funcionan correctamente con interrupciones activadas por nivel, en la sección 5.3.5 se explican los problemas derivados del uso de este tipo de interrupciones, y se dan sugerencias de como resolverlos.

La implementación del módulo KRNLINT está pensada para que las interrupciones estén numeradas de forma consecutiva. Próximas versiones podrían tener "huecos en la numeración" para permitir aumentar las interrupciones de diferentes fuentes sin cambiar las existentes. Por ejemplo, las interrupciones externas (controladas por IOAPIC) podrían comenzar a numerarse desde 0, mientras las interrupciones locales controladas por el LAPIC podrían numerarse desde 128.

[1] El 80386 no tiene unidad de punto flotante, pero brinda soporte a un coprocesador matemático externo.

Conclusiones

El presente trabajo constituye la primera experiencia de portado de PARTEMOS a otra plataforma, como resultado de la cual **se obtuvo una versión de PARTEMOS ejecutable sobre procesadores con arquitectura Intel de 32 bits (IA-32)**. El trabajo realizado no se limitó a portar el sistema de 16 bits previamente existente (que fue rebautizado como PARTEMOS16) a la arquitectura IA-32, sino que se hicieron cambios al núcleo para que el código resultante pueda ser fácilmente portado a otras plataformas. También hubo varios cambios no relacionados con la portabilidad, como rediseño de la interfaz del módulo HAL de interrupciones, la introducción de una nueva forma de conmutación de contexto (conmutación de contexto ligera), y la corrección de errores en el núcleo, entre otros.

La nueva versión de PARTEMOS obtenida, denominada PARTEMOS32, puede considerarse como **altamente portable**. Se identificaron secciones de código dependientes de la plataforma o de las herramientas de compilación usadas, las cuales fueron encapsuladas en archivos específicos de la plataforma, o en encabezados ".h" que utilizan compilación condicional para ser multiplataforma. Se creó una **metodología a seguir para portar PARTEMOS a otras plataformas**, que quedó dividida en una metodología para portar la capa de abstracción de hardware, y una metodología para portar el resto del sistema. En las metodologías se brindan las indicaciones necesarias para portar PARTEMOS, incluyendo que archivos específicos deben ser creados o modificados, y en que orden deberían portarse los mismos. Como forma de comprobar la portabilidad del código, se implementaron los archivos dependientes necesarios para que **PARTEMOS pueda compilar en dos plataformas**: las PC con arquitectura IA-32 (en modo protegido de 32 bits), y las PC-AT y compatibles (en modo real de 16 bits).

Para portar la nueva versión de PARTEMOS a otra plataforma, básicamente hay que crear la capa de abstracción de hardware (HAL) para la nueva plataforma (usando la metodología para portar el HAL), modificar unos pocos archivos del núcleo (indicados en la metodología para portar PARTEMOS), y crear los drivers de dispositivos necesarios. Durante este proceso, la mayor parte de debería permanecer intacta. Sin embargo, a pesar de los esfuerzos realizados por crear un núcleo portable, **en la práctica nunca se puede garantizar la portabilidad del núcleo a una plataforma** hasta que no se porte el mismo a la plataforma en cuestión. Por eso adicionalmente la metodología para portar PARTEMOS incluye **pautas a seguir a la hora de realizar cambios en el núcleo, requeridos para portarlo a una nueva plataforma**. En particular en el documento de tesis se advierten de posibles problemas que pueden surgir a la hora de portar PARTEMOS a una plataforma de 64 bits, y se indica como resolverlos.

Posiblemente la tarea más compleja a la hora de portar el micronúcleo PARTEMOS a otras plataformas sea crear la capa HAL para la nueva plataforma, ya que esta capa encapsula la mayor parte del código dependiente de la plataforma. Actualmente existen dos versiones de la capa HAL, una para cada plataforma soportada. La capa HAL destinada a la plataforma PC-AT de 16 bits fue creada con el objetivo de probar la portabilidad del resto del sistema, y se realizó adaptando la capa HAL existente de PARTEMOS16, escrita totalmente en lenguaje ensamblador. Por otro lado, **la capa HAL para PCs con arquitectura IA-32 fue escrita mayoritariamente en lenguaje C**, relegándose el uso del ensamblador a unas pocas secciones donde el uso del lenguaje C era poco factible.

Se encontraron algunas similitudes entre diferentes versiones realizadas de los módulos de la capa HAL, como por ejemplo similitudes entre las porciones en lenguaje "C" de los módulos CPUHAL para diferentes plataformas, o similitudes entre las versiones del módulo INTHAL para diferentes controladores. Sin embargo a partir de estas similitudes **no fue factible extraer secciones de código HAL comunes a cualquier plataforma**. Aun así podemos afirmar que la introducción del lenguaje C **mejoró la portabilidad de la capa HAL**, ya que su uso facilita la comprensión y la adaptación del código del HAL a otras plataformas.

Los controladores de interrupción APIC, disponibles en muchas PCs con arquitectura IA-32 modernas, han remplazado a los clásicos PIC8259 en muchos sistemas operativos modernos; motivo por el cual nos propusimos evaluar el uso de estos controladores para implementar el modelo de prioridades de PARTEMOS. De los estudios y pruebas realizados se llegó a la conclusión de que **los controladores APIC no son apropiados para implementar el modelo de prioridades de PARTEMOS**, aun cuando brindan características semejantes a este modelo. Como parte de esta investigación se realizaron algunas implementaciones del INTHAL sobre APIC, algunas de las cuales fueron fallidas, y una de ella capaz de funcionar como módulo INTHAL, pero sin brindar todas las ventajas del modelo de prioridades integrado. También se propuso una alternativa de implementación que, a pesar de ser costosa en operaciones de entrada/salida, posiblemente permita proporcionar el modelo de prioridades de PARTEMOS sobre APIC.

Para crear la versión de PARTEMOS para la arquitectura IA-32, fue necesario realizar una investigación para seleccionar las herramientas de desarrollo necesarias y sus técnicas de uso. Como resultado de esta investigación se logró estableces y documentar **un entorno para el desarrollo de sistemas operativos sobre la arquitectura IA-32**. Este entorno es útil para desarrollo general de sistemas operativos, no solo de PARTEMOS. Constituyendo parte esencial de este entorno se encuentran la herramientas núcleo, formadas por el compilador GCC, herramientas del paquete GNU binutils, y el ensamblador NASM. Debido a que las herramientas de compilación y enlace utilizadas han sido portadas a muchas plataformas, las técnicas de creación y uso del núcleo del entorno de desarrollo deben ser fácilmente adaptables para usar estas plataformas como destino de compilación.

También **se creó una metodología para el desarrollo de aplicaciones en PARTEMOS**, que indica como se deben utilizar las herramientas del entorno antes mencionado para compilar y ejecutar aplicaciones PARTEMOS para la arquitectura IA-32.

De los párrafos anteriores podemos concluir que **los objetivos de este trabajo fueron cumplidos,** sirviendo el mismo de base para continuar el futuro desarrollo de PARTEMOS sobre la arquitectura IA-32 y otras plataformas; y abriendo nuevas puertas a la investigación y desarrollo en el área de sistemas operativos.

Referencias

Accetta, M. J., Baron, R. V., Bolosky, W. J., Golub, D. B., Rashid, R. F., Tevanian, A. and Young, M. (1986). Mach: A New Kernel Foundation for UNIX Development. USENIX Association Conference Proceedings**:** 93-113, Atlanta GA, USA, USENIX Association.

ACPI (2011). Advanced Configuration and Power Interface Specification. Revision 5.0 December 6, 2011. Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd.,Toshiba Corporation.

Beck, M., Bohme, H., Kunitz, U., Magnus, R., Dziadzka, M. and Verworner, D. (1998). Linux kernel internals, Addison-Wesley Longman Publishing Co., Inc.

Dijkstra, E. W. (1968). "The structure of the "THE"-multiprogramming system." Communications of the ACM **11**(5): 341-346.

Eykholt, J. R., Kleiman, S. R., Barton, S., Faulkner, R., Shivalingiah, A., Smith, M., . . . Sunsoft, D. W. (1992). Beyond multiprocessing: Multithreading the SunOS kernel. Proceedings of the summer 1992 USENIX Technical Conference**:** 11-18.

Frampton, N. (2002) "Interrupt Architecture in Microsoft Windows CE .NET." Microsoft Developer Network. http://msdn.microsoft.com/en-us/library/ms836807.aspx.

Intel Corporation (1996). 82093AA I/O Advanced Programmable Interrupt Controller (IOAPIC). Order Number: 290566-001 May 1996

Intel Corporation (1997). MultiProcessor Specification. Version 1.4 May 1997

Intel Corporation (2012). Intel® 64 and IA-32 Architectures Software Developer's Manual. Combined Volumes 1, 2A, 2B, 2C, 3A, 3B, 3C. Order Number: 325462-043US May 2012

Jung, K. J., Jung, S. G. and Park, C. (2004). Stabilizing execution time of user processes by bottom half scheduling in Linux. Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on**:** 71 - 78.

Kenah, L. J. and Bate, S. F. (1984). VAX/VMS internals and data structures, Digital Press.

Kleiman, S. and Eykholt, J. R. (1995). "Interrupts as Threads." Operating Systems Review **29**(2): 21-26.

Leyva-del-Foyo, L. E. (2008). Administración de interrupciones en sistemas operativos de tiempo real. Departamento de computación. Mexico D.F., CINVESTAV-IPN. **Tesis en opción al grado de doctor en ciencias**.

Leyva-del-Foyo, L. E. and Mejia-Alvarez, P. (2004). Custom Interrupt Management for Real-Time and Embedded System Kernels. IEEE International Real-Time Systems Symposium (RTSS04), Embedded Real-Time Systems Implementation Workshop

Leyva-del-Foyo, L. E., Mejia-Alvarez, P. and de Niz, D. (2006a). "Predictable Interrupt Management for Real Time Kernels over conventional PC Hardware." Real-Time and Embedded Technology and Applications Symposium, IEEE: 14-23.

Leyva-del-Foyo, L. E., Mejia-Alvarez, P. and de Niz, D. (2006b). "Predictable Interrupt Scheduling with Low Overhead for Real-Time Kernels." International Workshop on Real-Time Computing Systems and Applications: 385-394.

Li, Q. and Yao, C. (2003). Real-Time Concepts for Embedded Systems, CMP Books.

López-Martínez, P. R. (2003). Micro núcleo para aplicaciones embebidas y de tiempo real. Departamento Computación. Santiago de Cuba, Universidad de Oriente, Cuba. **Tesis de licenciatura en Ciencia de la computación**.

Martínez-Cortés, W. (2005). Tratamiento de Interrupciones y su caracterización en un Micro-Núcleo de Tiempo Real. Departamento de computación. Santiago de Cuba, Cuba, Universidad de Oriente. **Tesis de Licenciatura en ciencia de la computación**.

Mauro, J. and McDougall, R. (2000). Solaris Internals, Core Kernel Architecture, Sun Microsystems Press.

McKusick, M. K., Bostic, K., Karels, M. J. and Quarterman, J. S. (1996). The design and implementation of the 4.4BSD operating system, Addison Wesley Longman Publishing Co., Inc.

Ousterhout, J. K. (1990). Why Aren't Operating Systems Getting Faster As Fast as Hardware? USENIX Summer**:** 247-256.

Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., . . . Neuhauser, W. (1991). "Overview of the CHORUS Distributed Operating Systems." Computing Systems **1**: 39-69.

Shi, H., Cai, M. and Dong, J. (2006). "Interrupt Synchronization Lock for Real-time Operating Systems." Computer and Information Technology, International Conference on **0**: 171.

Solomon, D. A. (1998). Inside Windows NT, Microsoft Press.

Stankovic, J. A. (1988). "Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems." Computer **21**(10): 10-19.

Stodolsky, D., Chen, J. B. and Bershad, B. N. (1993). Fast interrupt priority management in operating system kernels. USENIX Symposium on USENIX Microkernels and Other Kernel Architectures Symposium - Volume 4. San Diego, California, USENIX Association**:** 9-9.

Tanenbaum, A. S. and Woodhull, A. S. (2006). Operating systems - design and implementation (3. ed.), Pearson Education.

Thomson, M. and Browne, J. (2002) "Designing and Optimizing Microsoft Windows CE .NET for Real-Time Performance." Microsoft Developer Network. http://msdn.microsoft.com/en-us/library/ms836791.aspx.

Zhang, Y. and West, R. (2006). Process-Aware Interrupt Scheduling and Accounting. Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International**:** 191 -201.

Anexo A
Instalación del entorno de desarrollo

En este anexo se explican los pasos necesarios para instalar las herramientas esenciales (o herramientas núcleo) del entorno de desarrollo para sistemas operativos sobre la arquitectura IA-32, utilizando como plataforma de desarrollo una PC con sistema operativo Windows. Las herramientas pueden instalarse utilizando el CD de instalación del entorno que acompaña al presente documento, con lo que se logra obtener un entorno probado e idéntico al utilizado en este trabajo. Alternativamente se pueden descargar las herramientas del entorno desde internet, lo que permite instalar las últimas versiones de las mismas.

La Figura A-1 muestra los pasos a seguir para instalar el entorno de desarrollo. Se puede apreciar como tanto para el entorno Cygwin como para el compilador GCC cruzado existe la alternativa de instalar la versión disponible en el CD del entorno, o descargarlas desde internet. En el caso del compilador GCC cruzado, la descarga de internet implica la creación del compilador cruzado, descargándose únicamente el código fuente del mismo.

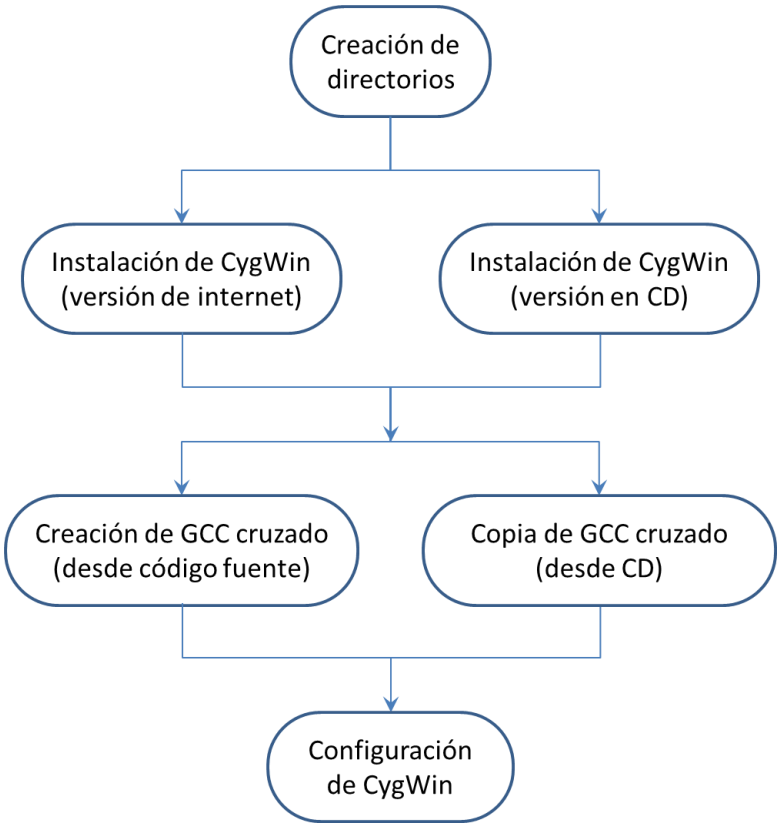


Figura A-1: Pasos de instalación del entorno de desarrollo.

A continuación se detallan cada uno de los pasos de instalación del entorno. En caso de que se decida utilizar las últimas versiones de las herramientas (descargadas de internet) también se explica como crear un CD de instalación del entorno de desarrollo con estas.

A1. Creación de directorios en la máquina de desarrollo

Los siguientes pasos de instalación del entorno requieren la existencia de dos directorios, uno donde estarán instaladas las herramientas del entorno de trabajo, y otro donde se colocarán los archivos necesarios para la instalación de estas herramientas. Aunque no es obligatorio usar un nombre específico para estos directorios, en lo adelante asumiremos que siempre se usarán los siguientes directorios:

```
C:\Developm
C:\INSTALLER
```

Que se usarán respectivamente para colocar los componentes del entorno de desarrollo y sus instaladores. Estos directorios deben ser creados en la computadora de desarrollo en caso de que no existan.

A2. Instalación de Cygwin

Las herramientas núcleo del entorno de desarrollo creado se ejecutan sobre la plataforma Cygwin, que implementa la mayor parte del API de Linux encima de Windows. Es por eso que esta debe ser la primera aplicación a instalar. A continuación se brindan los pasos para instalar la versión más reciente de esta aplicación, para lo cual se requiere contar con conexión a internet.

1. Descargar el instalador de Cygwin desde el sitio web oficial (http://www.cygwin.com/), que debe ser un archivo con nombre "`setup.exe`". Este archivo debe ser colocado en una subcarpeta dentro de la carpeta de instaladores, cuyo nombre identifique la versión de Cygwin. Por ejemplo "`C:\INSTALLER\Cygwin (v 1.7.10-1)`".
2. Ejecutar el instalador de Cygwin. Se debe observar una ventana similar a la mostrada en la Figura A-2.

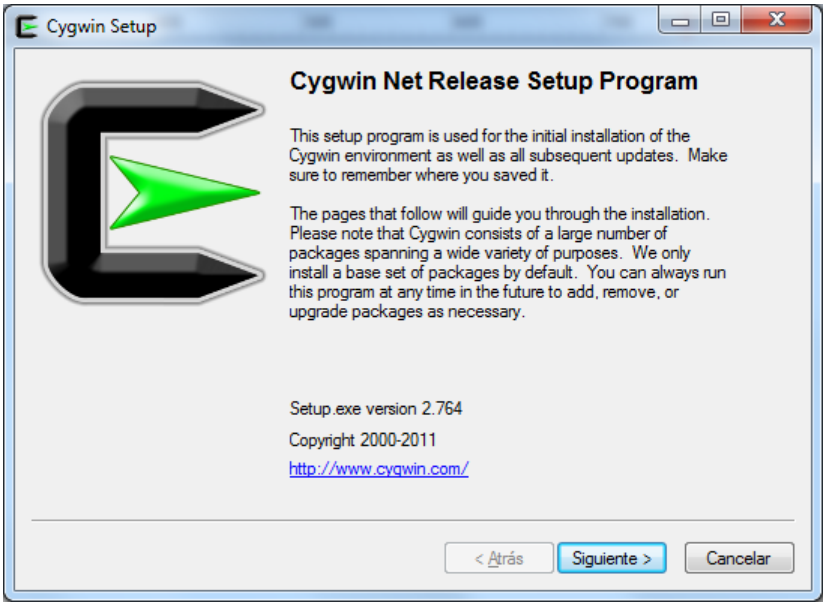


Figura A-2: Asistente de instalación de Cygwin.

3. El asistente mostrará una serie de ventanas solicitando información al usuario, se debe proporcionar la siguiente información:

   - **Tipo de instalación:** elegir la opción “Instalar desde Internet” (las opciones adicionales son “Descargar sin instalar” e “Instalar desde un directorio local”).
   - **Directorio de instalación:** “`C:\Developm\Cygwin`”
   - **Directorio local de paquetes:** Indica la carpeta donde se descargaran los archivos de instalación de los diferentes paquetes de Cygwin. Debe indicarse una subcarpeta dentro de la carpeta donde se colocó el instalador. Por ejemplo

     ```
     C:\INSTALLER\Cygwin (v 1.7.10-1)\Package
     ```

4. Luego de los pasos anteriores debe aparecer una ventana que permite la selección de los paquetes de software que deseamos instalar junto con Cygwin. Se deben seleccionar los siguientes paquetes:

   - nasm
   - gcc–core
   - gcc-g++
   - make
   - flex
   - bison
   - libgmp-devel
   - libmpc-devel
   - libmpfr-devel

   El paquete “nasm” instala el ensamblador del mismo nombre, útil si se quiere desarrollar código en ensamblador con la notación de Intel. Los restantes paquetes son necesarios si se quiere generar el compilador cruzado a partir del código fuente, como se explica mas adelante. En caso de utilizarse el compilador cruzado precompilado (ver sección A4 más adelante), puede obviarse la instalación estos paquetes. La Figura A-3 muestra la ventana de selección de paquetes, en la cual se ha realizado una búsqueda para localizar el paquete “nasm”.

   Este paso se puede aprovechar para seleccionar cualquier otro paquete que sea de interés al usuario. En caso de que se olvide seleccionar algún paquete, es posible instalarlo posteriormente ejecutando nuevamente el instalador de Cygwin.

Luego de realizar los pasos anteriores se debe finalizar la instalación, aceptando las opciones por defecto que brinda el asistente de instalación. Más adelante (sección A6) se explica como configurar Cygwin para facilitar el uso de las herramientas de desarrollo.

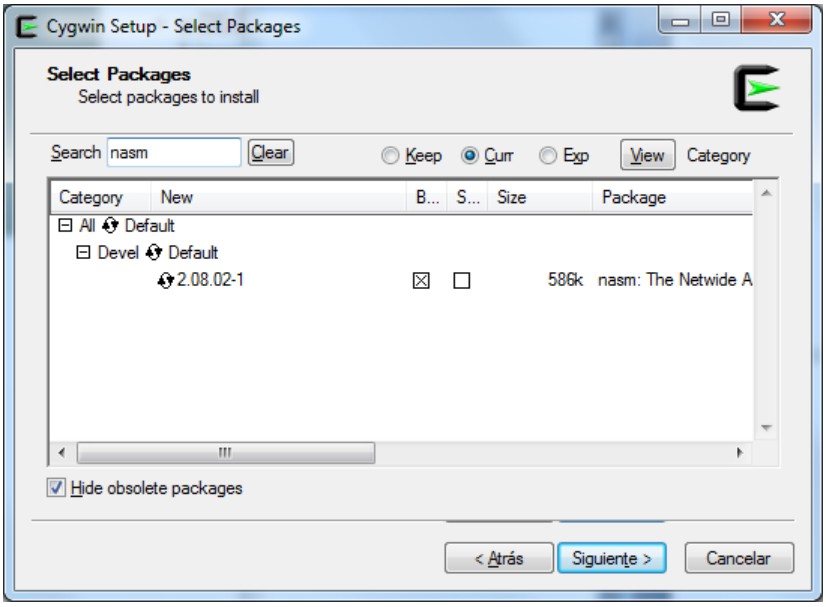


Figura A-3: Selección de paquetes de Cygwin.

A3. Instalación de Cygwin desde el CD de instalación del entorno

Este proceso instala Cygwin desde una carpeta local, que puede ser la encontrada en el CD de instalación del entorno de desarrollo, u otra carpeta como la creada en el proceso de instalación desde internet antes descrito. Un escenario de uso de este tipo de instalación puede ser la instalación de Cygwin en varias máquinas. En este caso se puede instalar Cygwin en una PC utilizando el procedimiento de descarga desde internet, y luego copiar la carpeta "C:\INSTALLER" a las restantes computadoras para instalar Cygwin desde esa carpeta local.

A continuación se brindan los pasos para instalar Cygwin desde una carpeta local. Asumiremos que se está utilizando el CD del entorno, y que se encuentra en la unidad "D:", de no ser así se deberá cambiar la letra de unidad por la correcta.

1. Ejecutar el instalador "`setup.exe`" localizado en una subcarpeta dentro de la carpeta de instaladores del CD, que contiene el nombre "Cywin" y un número de versión. Por ejemplo "`D:\INSTALLER\Cygwin (v 1.7.10-1)`". Se debe observar una ventana similar a la mostrada en la Figura A-2.
2. Se procede de forma similar a la instalación desde internet, con las siguientes diferencias:
   - Cuando se solicite el tipo de instalación se debe seleccionar la opción "Instalar desde un directorio local" en lugar de "Instalar desde Internet".
   - En lugar de solicitar una carpeta para realizar la descarga de paquetes, el instalador solicitará una carpeta local con los paquetes ya descargados. Debe indicarse la carpeta de paquetes de Cygwin contenida en el CD. Por ejemplo

     `D:\INSTALLER\Cygwin (v 1.7.10-1)\Package`

A4. Copia del compilador cruzado preinstalado

El CD de instalación del entorno de desarrollo cuenta con una versión precompilada del compilador GCC cruzado. El uso de esta versión nos ahorra todo el proceso de generación del compilador cruzado, y también nos da la seguridad de trabajar con una versión de las herramientas de compilación que ya ha sido probada previamente.

Para instalar la versión precompilada de compilador cruzado, simplemente debemos tomar el archivo "`cross-GCC-3.4.4.zip`"[1] de la carpeta de instalación en el CD y descompactarlo en el directorio "`C:\Developm\Cygwin\usr\local`", manteniendo la estructura de directorio del archivo comprimido.

A5. Creación del compilador cruzado desde el código fuente

A continuación se describe como crear un compilador GCC cruzado a partir del código fuente. Las instrucciones que se muestran asumen que el ambiente contiene los paquetes listados en la sección A2; de no ser así se debe ejecutar nuevamente el instalador de Cygwin para instalarlos.

Antes de proceder con los siguientes pasos, es buena idea asegurarse que las rutas en la variable de entorno $PATH no contienen la dirección de otras versiones de GCC, como MinGW (incluido con Dev-C++) ó DJGPP para Windows.

Descarga de los fuentes

El próximo paso es descargar los archivos con el código fuente de los paquetes "binutils" y "GCC", estos se pueden descargar compactados de http://ftp.gnu.org/gnu/, o se puede usar el gestor de paquetes de Cygwin para descargar los mismos. En el caso de GCC se recomienda descargar el paquete con nombre "`gcc-core-x.x.x`" en lugar del paquete "`gcc-x.x.x`", ya que el primero es suficiente para construir el compilador de C, mientras el último incluye el código de compiladores de Fortran, ADA, y otros lenguajes que no utilizados en el desarrollo de sistemas. Luego los archivos descargados deben descomprimirse, en lo adelante asumiremos que se descomprimieron dentro de la carpeta "`/usr/src`" del sistema de archivos de Cygwin (o sea, en la carpeta "`C:\Developm\Cygwin\usr\src`" dentro del sistema de archivos de Windows.).

En caso de usarse el gestor de paquetes de Cygwin para la descarga del código fuente, se debe marcar el cuadro de selección de la columna "source" para los paquetes "binutils" y "gcc-core-x.x.x", esto descargará el código fuente de cada paquete en la carpeta "`/usr/src`" del sistema de archivo de Cygwin. El código fuente del paquete **gcc-core** se descargará en un archivo que se debe descomprimir.

Compilación

Antes de iniciar la generación del compilador cruzado se deben configurar algunas variables de entorno que serán utilizadas más adelante. Para ello se deben escribir las siguientes instrucciones en la consola de comandos de Cygwin:

[1] El número "3.4.4" identifica la versión del compilador GCC. El CD de instalación del entorno podría contener un archivo con un número de versión diferente.

```
export PREFIX=/usr/local/cross
export TARGET=i586-elf
cd /usr/src
mkdir build-binutils build-gcc
```

La variable `PREFIX` indica la carpeta destino para los archivos del compilador cruzado. La variable TARGET indica el formato de salida que generará el nuevo compilador y para qué tipo de máquina lo hará. Las dos últimas líneas del cuadro anterior se utilizan para crear carpetas de trabajo.

El siguiente paso es compilar el paquete **binutils,** que incluye el enlazador LD y otras herramientas, para esto se deben ejecutar los siguientes comandos:

```
cd /usr/src/build-binutils
../binutils-x.xx/configure --target=$TARGET --prefix=$PREFIX --disable-nls
make all
make install
```

Las letras "`x.xx`" deben remplazarse por el numero de versión correcto descargado. La opción "`--disable-nls`" configura a **binutils** para que no incluya soporte al lenguaje nativo. Esto es opcional pero reduce las dependencias y el tiempo de compilación. También provoca que todos los mensajes de estas herramientas se proporcionen en inglés, lo cual en general es deseable ya que se puede encontrar más información sobre un mensaje específico si este se encuentra en ese idioma.

Por último debemos compilar el paquete de GCC, usando los siguientes comandos:

```
cd /usr/src/build-gcc
export PATH=$PATH:$PREFIX/bin
../gcc-x.x.x/configure --target=$TARGET --prefix=$PREFIX \
--disable-nls --enable-languages=c,c++ --without-headers
make all-gcc
make install-gcc
```

El segundo comando extiende la variable de entorno `PATH`, ya que durante el proceso de compilación de GCC se necesitará del paquete **binutils** recién compilado. En la próxima sección se explicará cómo extender esta variable permanentemente, de forma que no se necesite escribir la ruta completa para usar el compilador cruzado. La cadena "`x.x.x`" se debe remplazar por el numero de versión correcto descargado.

La opción "`--disable-nls`" tiene el mismo significado que para el caso anterior de la compilación de **binutils**. La opción "`--without-headers`" le dice a GCC que no se apoye en ninguna biblioteca de C disponible para la plataforma destino, mientras que "`--enable-languages`" le dice que no compile los programas de interfaz para otros lenguajes soportados, sino solo para C (y opcionalmente C++). En el caso de haberse usado el gestor de Cygwin, para poder dar soporte a C++ se deben descargar los archivos fuente del paquete de C++.

A6. Configuraración de directorios para uso del entorno

Para utilizar las herramientas sin tener que escribir su ruta completa, se debe incluir la ruta "`/usr/local/cross/bin/`" en la variable de entorno PATH. Una forma de hacer esto es crear un

comando que nos permita configurar el ambiente Cygwin para el desarrollo de sistemas. Por ejemplo podemos editar el archivo `\etc\profile`, y agregarle las siguientes líneas:

```
osdev()
{
  echo Configurando ambiente CygWin para desarrollo de OS
  echo
  echo rutas PATH previas:
  echo $PATH | tr : \\n

  export PATH=/usr/local/bin:/usr/bin:/usr/local/cross/bin
  echo
  echo rutas PATH actuales:
  echo $PATH | tr : \\n

  cd "carpeta_de_trabajo"
}
```

Esto crea un nuevo comando llamado `osdev`, que al ejecutarse configura el entorno Cygwin para invocar las herramientas del compilador cruzado, sin necesidad de escribir las rutas completas. La última instrucción contiene la frase "`carpeta_de_trabajo`", que debe remplazarse por una ruta válida a una carpeta donde se realice el desarrollo de sistema, por ejemplo la carpeta "`gcc32prj`" del sistema PARTEMOS.

Cada vez que se desee utilizar las herramientas de compilación cruzada para el desarrollo de sistemas, se debe abrir la consola de comandos de Cygwin y escribir la instrucción:

```
osdev
```

La ventaja de la técnica antes descrita sobre otra que configure la variable `PATH` de forma permanente es que permite configurar el entorno Cygwin para realizar otras actividades. Por ejemplo se podría crear otro comando similar al anterior, que configure el ambiente CygWin para el desarrollo de aplicaciones nativas para Android.

A7. Creación de un CD de instalación del entorno

A continuación se describe como crear un CD de instalación de las herramientas del entorno de desarrollo, similar al utilizado en los pasos anteriores. Este disco es útil para crear una "instantánea" del software de desarrollo utilizado en un momento dado, evitando así problemas asociados con la variabilidad de las herramientas de software libre utilizadas en el entorno. Los pasos mostrados a continuación solo aplican al caso en que las herramientas del entorno fueron descargadas desde internet, ya que no tiene sentido volver a crear un disco con las mismas versiones de las herramientas disponibles en el CD de instalación.

Una vez realizados los pasos para la instalación de las herramientas desde internet (o sea, los pasos que no requieren del CD de instalación del entorno), la creación del disco de distribución es un proceso sencillo. Casi todos los componentes del núcleo del entorno que deben grabarse en el disco se encuentran en la carpeta "`C:\INSTALLER`", sólo resta copiar los archivos binarios del compilador GCC

cruzado. Los archivos del compilador GCC cruzado se localizan en la carpeta "`C:\Developm\Cygwin\usr\local\cross`", que debe ser compactada en un archivo con formato "ZIP", manteniendo la estructura de directorios. Utilizamos la notación "`cross-GCC-X.Y.Z.zip`" para nombrar al archivo comprimido resultante, donde "`X.Y.Z`" representa la versión del compilador GCC cruzado. Por último el archivo ZIP debe ser copiado a la carpeta "`C:\INSTALLER`", la cual debe ser grabada en el nuevo CD de instalación utilizando cualquier aplicación destinada al efecto.

Adicionalmente, previo a la grabación del CD de instalación se puede colocar en la carpeta "`C:\INSTALLER`" otros instaladores de cualquier otra herramienta deseada. Por ejemplo pueden copiarse instaladores de simuladores de plataformas, herramientas para trabajo con discos virtuales, entre otras.

Anexo B
Instalación y uso de otros componentes del entorno

En este anexo se explica el proceso de instalación y uso de algunos componentes del entorno para el desarrollo de sistemas operativos para la arquitectura IA-32. Aunque estos componentes no forman parte del núcleo del entorno de desarrollo, se consideró importante dedicarles esta sección debido a jugaron un importante papel en el desarrollo de PARTEMOS para 32 bits, y a que no son herramientas de uso común.

B1. Instalación de la máquina virtual Bochs y depurador Peter Bochs

La carpeta "INSTALLER\Bochs" dentro del CD de instalación del entorno contiene los instaladores del emulador Bochs y del depurador Peter Bochs, utilizados en este trabajo; así como otros archivos que facilitan su uso y que serán mencionados más adelante. Las aplicaciones mencionadas se encuentran en constante desarrollo, por lo que se recomienda descargar las últimas versiones de las mismas desde sus páginas oficiales en internet[1].

La instalación del emulador de máquina virtual Bochs[2] en Windows es tan sencilla como ejecutar el instalador del software, y seguir las instrucciones. Se recomienda instalar este software fuera de la carpeta de instalación por defecto de Windows, lo cual facilita el trabajo sobre esta carpeta. A continuación asumimos que el emulador fue instalado en "`C:\Developm\Bochs-X.Y.Z`", donde "`X.Y.Z`" representa la versión de Bochs instalada.

Para crear una máquina virtual para Bochs, es necesario editar un archivo texto (con extensión "`.bxrc`") que contenga la configuración de la máquina. El CD de instalación del entorno contiene un archivo llamado "`miKernel.bxrc`", que configura una máquina virtual con opciones de depuración habilitadas, la cual asume que el disco flexible "`A:`" es un disco colocado en la unidad "`G:`" de la máquina física. En la próxima sección se explica como montar una imagen de disco en una letra de unidad de la máquina física, para utilizarla como disco de la máquina virtual.

Se sugiere que el archivo de configuración de máquina virtual usado se coloque en la carpeta principal de instalación de Bochs (o sea, en "`C:\Developm\Bochs-X.Y.Z`"). Para la ejecución directa de la máquina virtual (sin depurador Peter Bochs), se puede dar doble click sobre el icono del archivo "`.brxc`", ya que el instalador asocia esta extensión con el emulador Bochs. En lo adelante se asumirá que la máquina virtual usada es descrita por "`miKernel.bxrc`", que fue copiado desde el CD de instalación a la carpeta de Bochs.

[1] Puede suceder que exista alguna incompatibilidad entre la última versión de Bochs y el depurador Peter Bochs, en este caso puede ser necesario descargar una versión anterior del emulador Bochs.
[2] http://bochs.sourceforge.net/

Peter Bochs[1] es una interfaz gráfica de depuración para Bochs. Su autor lo creó para facilitar su propio trabajo de desarrollo de sistemas operativos, y hasta la fecha todavía continúa agregándole funcionalidades. Peter Bochs está escrito en Java, su código fuente es libre y se puede descargar desde internet. La máquina de desarrollo debe contar con alguna máquina virtual de Java para ejecutar esta aplicación.

El archivo binario de PeterBochs (con extensión ".jar") puede ser descargado desde el sitio de la aplicación en internet, o alternativamente usa la versión disponible en el CD de instalación del entorno. Para facilitar la ejecución de esta aplicación se recomienda copiarla en la misma carpeta de instalación de Bochs, junto con el archivo de configuración de la máquina virtual. Para ejecutar Peter Bochs en Windows, se puede escribir el siguiente comando:

```
java -jar peter-bochs-debugger20111224.jar
C:\Developm\Bochs-X.Y.Z\bochsdbg.exe -q -f miKernel.bxrc
```

donde "`peter-bochs-debugger20111224.jar`" es el archivo ejecutable de Peter Bochs (la parte numérica puede variar según la versión). El siguiente parámetro es la ruta completa (no se permiten rutas relativas) al archivo "`bochsdbg.exe`", localizado en la carpeta de instalación de Bochs. El archivo "`miKernel.bxrc`" es el nombre del archivo de configuración de la máquina virtual a ejecutar. Para evitar la escritura frecuente del comando anterior, se recomienda escribirlo en un archivo de procesamiento por lotes (extensión "`.bat`"), un ejemplo de este tipo de archivo es "`run_peter.bat`", que se puede encontrar en el CD de instalación.

B2. Trabajo con discos virtuales

Una forma de iniciar una máquina virtual es utilizar una imagen de disco de inicio. En la carpeta "`INSTALLER\floppy image`" del CD de instalación del entorno se puede encontrar un archivo llamado "`grub_disk.img`", que contiene una imagen de disco flexible configurado para iniciar una PC con el cargador GRUB. La mayoría de los simuladores de plataforma (incluyendo el emulador Bochs) pueden utilizar directamente este tipo de imágenes como si fueran un disco físico de la máquina virtual, sin embargo esta forma de uso de las imágenes de disco puede dificultar un poco la edición de las mismas.

Muchos simuladores de máquinas virtuales también pueden acceder directamente a algunas unidades de disco de la máquina hospedera, como unidades de disco flexibles. En esto caso una alternativa más cómoda para utilizar las imágenes de los discos flexibles dentro de la máquina virtual es montar las mismas como unidades de la máquina hospedera, utilizando algún software destinado al efecto. Esta técnica tiene la ventaja de que se puede editar el archivo imagen de disco accediendo a la unidad montada de forma similar a como se haría con cualquier disco físico; luego puede utilizarse el archivo imagen como disco de la máquina virtual sin necesidad de desmontarlo en la máquina hospedera.

El archivo de configuración de la máquina virtual Bochs descrito en la sección anterior utiliza la unidad de disco "`G:`" de la máquina hospedera como disco flexible "`A:`". Por eso los siguientes pasos describen, a manera de ejemplo, como se puede montar un archivo de imagen en esta unidad de disco. Para utilizar

[1] http://code.google.com/p/peter-bochs/

el archivo imagen con el cargador GRUB mencionado previamente, se debe copiar el mismo del CD de instalación a una carpeta en el disco duro de la máquina de desarrollo.

La carpeta "`INSTALLER\floppy image`" del CD de instalación del entorno contiene un archivo instalador denominado "`imdiskinst.exe`"[1], que instala un software denominado "ImDisk Virtual Disk Driver". Esta aplicación permite montar una imagen de disco flexible como si fuera un disco real. Una vez instalado sólo se puede hacer click derecho sobre una imagen .img y seleccionar la opción "Mount as ImDisk Virtual Disk", lo que abrirá una ventana similar a la mostrada en la Figura B-1.

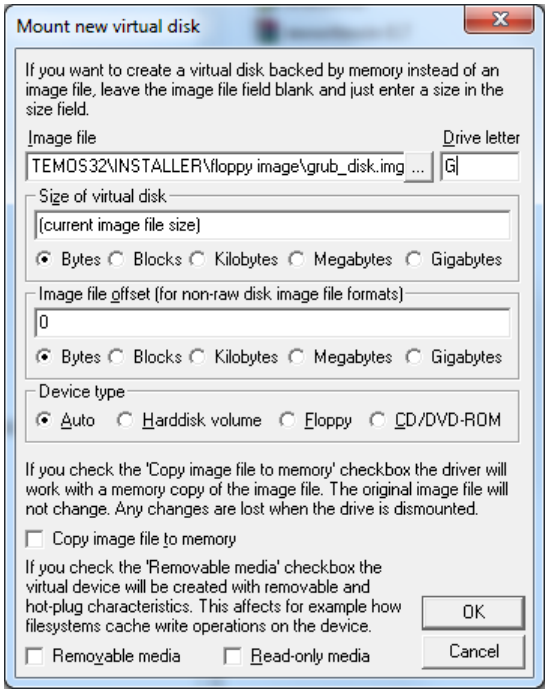


Figura B-1: Software para montar una imagen de disco.

Luego de escribir la letra de unidad deseada (en este caso "G") y hacer click en [OK] se creará una unidad de disco que los programas (incluyendo Bochs) detectarán como una unidad real.

B3. Instalación de GRUB en una memoria USB

A diferencia de las máquinas virtuales, muchas máquinas modernas no incluyen unidad de disco flexible para iniciar el sistema, por lo que no es posible utilizar estos medios para probar la ejecución de sistemas operativos en desarrollo. Por otro lado esas mismas máquinas brindan la posibilidad de iniciar el sistema desde dispositivos de almacenamiento externo conectables por USB, de los cuales los más comunes y portables son las denominadas "memorias USB". A continuación se explica un método para instalar el cargador GRUB en estos dispositivos.

El procedimiento que se describe a continuación instala en la memoria USB una versión de GRUB denominada GRUB4DOS. Esta versión incluye un menú con muchas opciones que pueden resultar de interés para cargar y recuperar sistemas operativos, además de la clásica interfaz de línea de comando, que podemos usar para cargar las imágenes de núcleo de sistema operativo desarrolladas. Para

[1] También descargable de http://www.ltr-data.se/opencode.html

proseguir, se debe contar con los archivos "`grub4dos-0.4.4.zip`" y "`grubinst-1.1-bin-w32-2008-01-01.zip`", los cuales se localizan en la carpeta "`INSTALLER\GRUB`" del CD de instalación del entorno de desarrollo, o se pueden descargar desde las páginas http://download.gna.org/grub4dos/ y http://download.gna.org/grubutil/ respectivamente.

El primer paso del procedimiento consiste en copiar los archivos de GRUB4DOS a la raíz de la memoria USB, se recomienda formatear la memoria previamente. Los archivos de GRUB4DOS deben extraerse del comprimido "`grub4dos-0.4.4.zip`", y sólo son necesarios los siguientes:

- `default`
- `grldr`
- `menu.lst`

El próximo paso consiste en configurar la memoria USB para poder iniciar desde ella. Para ello se debe extraer el archivo comprimido "`grubinst-1.1-bin-w32-2008-01-01.zip`", y ejecutar el programa "`grubinst_gui.exe`", es importante que en el caso de usarse Windows Vista/7 el programa se ejecute con permisos de administrador. En la Figura B-2 se muestra la interfaz gráfica de la aplicación en ejecución. En la lista desplegable para seleccionar el disco se debe elegir la unidad correspondiente a la memoria USB, la cual debería ser identificable por su tamaño[1]. Para concluir, se debe presionar el botón [install], con lo que la memoria USB quedará lista para ser utilizada como dispositivo de arranque.

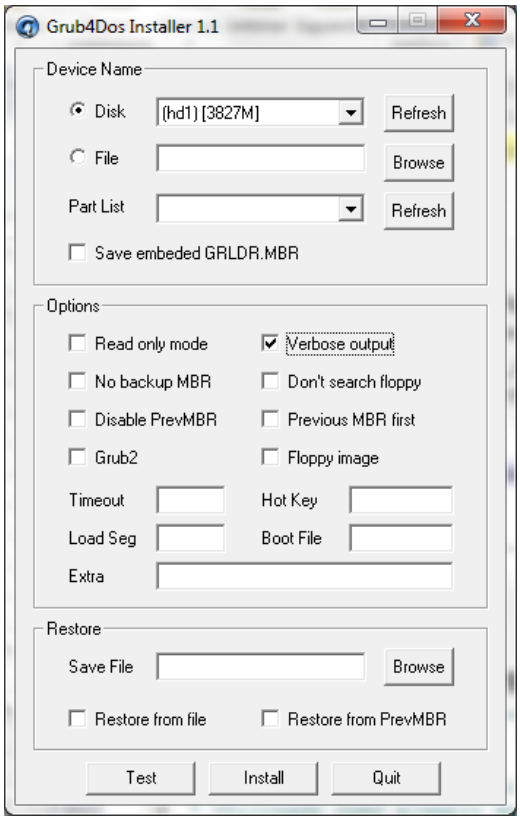


Figura B-2: Instalador de Grub4Dos.

[1] En caso de duda sobre que unidad usar el administrador de dispositivos de Windows brinda el número de unidad.

Anexo C
Código ejemplo de aplicación PARTEMOS

```
/*************************************************************************\
 *                                ball.h                                 *
 *                                ------                                 *
 * Ejemplo de Uso del Kernel PARTEMOS                                    *
 * Utilizado como ejemplo en tesis de Maestria                           *
 *                                                                       *
 * William Martinez Cortes                                               *
\*************************************************************************/
typedef struct {
    int x,y;
    int dx,dy;
    TASKID tarea;
} bolaStruct;

/* Usamos aritmetica de punto fijo para la pos y vel */
#define FracDig 8  /* Cantidad de digitos binarios de parte fraccionaria */
#define ClkFrec 33  /* Frecuencia de actualización de bola Hz            */
#define MaxVelInt 2  /* Velocidad Max entera de la bola (en cada eje)    */
#define MaxVel (MaxVelInt<<FracDig)  /* Velocidad Maxima en punto fijo   */
#define MaxDelta (MaxVel / ClkFrec)  /* DeltaPos Maxima en punto fijo    */
#define MaxX (80<<FracDig)     /* Posición X maxima en pantalla (pto fijo)*/
#define MaxY (25<<FracDig)     /* Posición Y maxima en pantalla (pto fijo)*/
#define IntPart(x) ((x)>>FracDig)   /* Convierte punto fijo a entero     */

extern SYNCID mutex1;   /* Objeto de sincronización para sección crítica */

/* Funciones exportadas */
void crearBola(bolaStruct * b);
void destruirBola(bolaStruct * b);

/*-------------------------< Fin de ball.h >---------------------------*/
```

```
/*************************************************************************\
 *                                ball.c                                 *
 *                                ------                                 *
 * Ejemplo de Uso del Kernel PARTEMOS                                    *
 * Utilizado como ejemplo en tesis de Maestria                           *
 *                                                                       *
 * William Martinez Cortes                                               *
\*************************************************************************/
#include <KERNEL.H>        // API del nucleo
#include <system.h>        // para rand()

#include "ball.h"

```

```
14 /*============================================================================*\
15  * Código de una tarea periódica sin lazo infinito                            *
16  * - La misma se activa con el servicio makeTaskPeriodic()                    *
17 \*============================================================================*/
18 KRNLTASK bolaCod(WORD_ arg)
19 {
20     int x,y;
21     bolaStruct *pBola = (bolaStruct *)arg;
22     wait(mutex1); /* Sección crítica para extraer posición */
23     x = pBola->x;
24     y = pBola->y;
25     signal(mutex1);
26 
27     x += pBola->dx;
28     y += pBola->dy;
29 
30     if(x>MaxX){
31         x= 2*MaxX-x;
32         pBola->dx = -pBola->dx;
33     }
34     if(y>MaxY){
35         y= 2*MaxY-y;
36         pBola->dy = -pBola->dy;
37     }
38     if(x<0){
39         x= -x;
40         pBola->dx = -pBola->dx;
41     }
42     if(y<0){
43         y= -y;
44         pBola->dy = -pBola->dy;
45     }
46 
47     wait(mutex1); /* Sección crítica para actualizar posición */
48     pBola->x = x;
49     pBola->y = y;
50     signal(mutex1);
51 }
52 
53 /*============================================================================*\
54  * Inicializa la estructura bola que recibe por parametro                     *
55  * - Incluye crear una tarea asociada con la bola                             *
56 \*============================================================================*/
57 void crearBola(bolaStruct * b)
58 {
59     /* Asignar posicion y velocidades aleatorias  */
60     b->x = rand() % MaxX;  /* X inicial aleatoria */
61     b->y = rand() % MaxY;  /* Y inicial aleatoria */
62     b->dx = rand()%(MaxDelta*2) - MaxDelta;
63     b->dy = rand()%(MaxDelta*2) - MaxDelta;
64     /* crear y ejecutar tarea */
65     b->tarea = createTask(bolaCod,  4, 0);
66     /* Período de Activación_________________       */
67     /* Primera activación_____________    |       */
68     /* Argumento_____________________  |  |       */
69     /* Tarea periódica __            |  |  |       */
70     /*                   |           |  |  |       */
71     /*                   v           v  v  v       */
72     makeTaskPeriodic(b->tarea, (WORD_)b, 3, 3);
73 }
```

```
74
75 /*=========================================================================*\
76  * Destructor de bola                                                      *
77  * Elimina la tarea asociada a la estructura bola                          *
78 \*=========================================================================*/
79 void destruirBola(bolaStruct * b) {
80     killTask(b->tarea);
81 }
82
83 /*-------------------------< Fin de ball.c >-----------------------------*/
```

```
1 /***************************************************************************\
2  *                                MainBall.c                               *
3  *                                ----------                               *
4  * Ejemplo de Uso del Kernel PARTEMOS                                      *
5  * Utilizado como ejemplo en tesis de Maestria                             *
6  *                                                                         *
7  * William Martinez Cortes                                                 *
8 \***************************************************************************/
9 #include <AAL/iKRNLAAL.H>  // para setupAsync
10 #include <scr/iScreen.h>   // para salida en pantalla
11 #include <KBDDRV.H>        // driver de teclado
12
13 #include "ball.h"
14
15 WinID winM;  /* Identificador de ventana modo texto */
16 SYNCID mutex1;   /* Mutex (Objeto de sincronización para sección crítica */
17
18 #define MaxBolas 10
19 bolaStruct bolas[MaxBolas];
20 int numBolas = 0;
21
22 /*=========================================================================*\
23  * MuestraPantalla Muestra todas las bolas en pantalla                     *
24 \*=========================================================================*/
25 void MuestraPantalla(void)
26 {
27     int i,x,y;
28     wclrscr(winM);
29     wait(mutex1);
30     for(i=0; i<numBolas; i++){
31         bolaStruct *b=bolas+i;
32         x = IntPart(b->x);
33         y = IntPart(b->y);
34         wgotoxy(winM, x, y);
35         wprintf(winM, "*");
36     }
37     signal(mutex1);
38 }
39
40 /*=========================================================================*\
41  * rutinaAsync Se ejecuta para actualizar pantalla si la tarea principal *
42  *             se bloquea por mucho tiempo                                 *
43 \*=========================================================================*/
44 void rutinaAsync(void)
45 {
46     MuestraPantalla();
47     beginWatch(7);   /* Fija nuevo límite al tiempo de espera por tecla  */
48 }
```

```
49
50 /*=========================================================================*\
51  * mainCode Código de la tarea principal, (primera que se ejecuta)       *
52 \*=========================================================================*/
53 KRNLTASK mainCode( )
54 {
55     int i;
56     BYTE c;
57
58     winM = CreateWindow(0,0,80,25);
59     mutex1 = createMutex();           /* Crea mutex para sección crítica */
60
61     wclrscr(winM);
62     wprintf(winM, "Uso: Presione [S] para crear bolas, ESC para salir");
63     wprintf(winM, "\r\npresione cualquier tecla para continuar");
64     getTcl();
65
66     setupAsync(rutinaAsync);  /* Establece manejador de señal asincrona  */
67     do {
68         beginWatch(7);    /* Fija un límite al tiempo de espera por tecla */
69         c = getTcl();     /* Obtiene el código de una tecla oprimida      */
70         endWatch();       /* Cancela la supervision de tiempo             */
71         if (c == 31) {    /* ¿Es la tecla [S]?                            */
72             if(numBolas<MaxBolas) {         /* Crear nueva bola         */
73                 crearBola(bolas+numBolas);
74                 numBolas++;
75             }
76         }
77         MuestraPantalla();
78     } while (c != 1);
79
80     //Borrar todas las bolas
81     for(i=0; i<numBolas; i++){
82         destruirBola(bolas+i);
83     }
84     // Destruir Mutex
85     deleteSync(mutex1);
86 }
87
88 /*-----------------------< Fin de MainBall.C >---------------------------*/
```

Anexo D
Listado de archivos de PARTEMOS32 para análisis de portabilidad

no	archivo	zona	portable	AT16	IA32	tamaño	FAEP[1]	esfuerzo
1	SRC\syslib.c[2]	núcleo	si	no	si	1650		
2	SRC\AAL\KrnlAsy.c	núcleo	si	si	si	24617		
3	SRC\AAL\KrnlAAL.c	núcleo	si	si	si	57806		
4	SRC\CFG\configLi.c[3]	núcleo	si	si	si	14365		
5	SRC\CFG\KCfgLi.c[3]	núcleo	si	si	si	832		
6	SRC\CFG\KRNLCFG.C[4]	núcleo	no	si	no	1958	0.1	195.8
7	SRC\CFG\KRNLSIZE.C	núcleo	si	si	si	1829		
8	SRC\CFG\CONFIG.C[4]	núcleo	no	si	no	23446	0.1	2344.6
9	SRC\ClkDrv\LampClk.c[5]	núcleo	si	si	si	4858		
10	SRC\ClkDrv\ClkDrvLL.c[5]	núcleo	si	si	si	24697		
11	SRC\DBC\DBC.C	núcleo	si	si	si	2653		
12	SRC\HAL4AT16\IHAL16.C	HAL	no	si	no	4912	0.5	2456
13	SRC\HAL4AT16\CPUHAL16.C	HAL	no	si	no	5115	0.5	2557.5
14	SRC\HAL4AT16\FakeIHal\FakeIHAL.c[6]	HAL	no	si	no	26685	1	26685
15	SRC\HAL4AT16\MEM\MemHal.c	HAL	no	si	no	963	0.5	481.5
16	SRC\HAL4AT16\ATINTHAL.ASM	HAL	no	si	no	77645	0.5	38822.5
17	SRC\HAL4AT16\cpuhal_.ASM	HAL	no	si	no	13510	0.5	6755
18	SRC\HAL4AT16\ExcHal.asm	HAL	no	si	no	4660	0.5	2330
19	SRC\HAL4AT16\STKCHK.ASM	HAL	no	si	no	2190	0.5	1095
20	SRC\HAL4IA32\cpuhal32.c	HAL	no	no	si	5430	0.5	2715
21	SRC\HAL4IA32\ExcHal.c	HAL	no	no	si	6294	0.5	3147
22	SRC\HAL4IA32\iHal32.c	HAL	no	no	si	20621	0.5	10310.5
23	SRC\HAL4IA32\MEM\tmpMemHal.c	HAL	no	no	si	880	0.5	440
24	SRC\HAL4IA32\c0hal.asm[7]	HAL	no	no	si	2346	0.8	1876.8
25	SRC\HAL4IA32\cpuhal.asm	HAL	no	no	si	12974	0.5	6487

[1] FAEP: Factor de Aporte al Esfuerzo de Portado.
[2] Funciones de biblioteca estándar, no usado en 16 bits por motivos de eficiencia.
[3] Versión limitada del módulo de configuración.
[4] Módulo de configuración para AT16, FAEP muy bajo (0.1) porque no se necesita para portar PARTEMOS.
[5] Archivos del driver de reloj.
[6] El FAEP es 1 para estimar el esfuerzo de hacer otro INTHAL falso.
[7] No existe en 16 bits y es muy probable que se requiera algo parecido en otra plataforma, por eso tiene FAEP 0.8.

no	archivo	zona	portable	AT16	IA32	tamaño	FAEP[1]	esfuerzo
26	SRC\HAL4IA32\ExcHal0.asm	HAL	no	no	si	4326	0.5	2163
27	SRC\HAL4IA32\irqhal32.asm	HAL	no	no	si	3020	0.5	1510
28	SRC\HAL4PC\CLK\8253HAL.c[2]	HAL	no	si	si	3181	1	3181
29	SRC\HPC\80x86\8086.C[3]	Hard[4]	no	si	si	2246	0.5	1123
30	SRC\HPC\8253\8253.C[5]	Hard	no	si	si	23484	1	23484
31	SRC\HPC\8259\8259.asm	Hard	no	si	no	14056	0.5	7028
32	SRC\HPC\IA32\descript.c[6]	Hard	no	no	si	5478	0.75	4108.5
33	SRC\HPC\IA32\HwDebug.c[6]	Hard	no	no	si	3316	0.75	2487
34	SRC\HPC\IA32\APIC.C	Hard	no	no	si	8215	0.5	4107.5
35	SRC\KRNL\KRNLDBC.C	núcleo	si	si	si	2000		
36	SRC\KRNL\KrnlSync.c	núcleo	si	si	si	73161		
37	SRC\KRNL\KrnlTask.c	núcleo	si	si	si	24527		
38	SRC\KRNL\KrnlTime.c	núcleo	si	si	si	54070		
39	SRC\KRNL\KRNLBOOT.c	núcleo	si	si	si	24262		
40	SRC\LIST\Pri1List.c	núcleo	si	si	si	4269		
41	SRC\LIST\Priority.c	núcleo	si	si	si	2045		
42	SRC\LIST\CList.c	núcleo	si	si	si	4405		
43	SRC\MEM\KrnlMem.c	núcleo	si	si	si	18541		
44	SRC\MEM\BMPPool\BmpPool.c	núcleo	si	si	si	19518		
45	SRC\MEM\ListMngr\ListMngr.c	núcleo	si	si	si	24894		
46	SRC\MEM\Utils\Dump.c	núcleo	si	si	si	2423		
47	SRC\MEM\ZoneMngr\ZoneMngr.c	núcleo	si	si	si	25658		
48	SRC\MIS\AuxTime.c	núcleo	si	si	si	3065		
49	SRC\Scr\SCREEN.C[7]	núcleo	multi[8]	si	si	22606	0.1	2260.6
50	SRC\Scr\SCRSMEM.C	núcleo	si	si	si	1061		
51	SRC\Scr\KrnlOut.c	núcleo	si	si	si	1137		
52	SRC\TAL\KRNLINT.c	núcleo	si	si	si	14591		
53	SRC\TAL\KRNLExc.c	núcleo	si	si	si	2023		
54	SRC\XCP\Excptn.c	núcleo	si	si	si	30080		

[1] FAEP: Factor de Aporte al Esfuerzo de Portado.
[2] Archivo no portable común a ambas plataformas (PC), por eso el FAEP es 1.
[3] Archivo no portable común a ambas plataformas, con compilación condicional, por eso FAEP 1.
[4] Representa archivos específicos de acceso al hardware, que no brindan la interfaz de la capa HAL.
[5] Archivo no portable común a ambas plataformas (PC), por eso el FAEP es 1.
[6] Archivos específicos de IA32, probablemente se necesite algo similar en otras plataformas, por eso su FAEP alto.
[7] Administrador de ventana, FAEP 0.1 porque posee un pequeño segmento con compilación condicional.
[8] Indica que el archivo es común a todas las plataformas, pero requiere modificación a la hora de portar el sistema. Usa compilación condicional.

no	archivo	zona	portable	AT16	IA32	tamaño	FAEP[1]	esfuerzo
55	include\CPUTYPE.H	núcleo	multi	si	si	3425	0.5	1712.5
56	include\dbgLog.h	núcleo	multi	si	si	2520	0.5	1260
57	include\global.h	núcleo	multi	si	si	3198	0.5	1599
58	include\KERNEL.H	núcleo	si	si	si	1948		
59	include\KRNLDEFN.H	núcleo	si	si	si	4180		
60	include\KRNLTASK.H	núcleo	si	si	si	356		
61	include\KRNLTIME.H	núcleo	si	si	si	1870		
62	include\KRNLTYPE.H	núcleo	multi	si	si	2620	0.5	1310
63	include\SIPCLTCB.H	núcleo	si	si	si	1414		
64	include\stack.h	núcleo	multi	si	si	2860	0.5	1430
65	include\Statist.h	núcleo	si	si	si	654		
66	include\AAL\PKRNLAAL.H	núcleo	si	si	si	6824		
67	include\AAL\IKRNLAAL.H	núcleo	si	si	si	9973		
68	include\CFG\iCONFIG.H	núcleo	si	si	si	432		
69	include\CFG\iKRNLCFG.H	núcleo	si	si	si	668		
70	include\CFG\iKRNSIZE.H	núcleo	si	si	si	1774		
71	include\CFG\CFGTYPES.H	núcleo	si	si	si	696		
72	include\CLKDRV\ICLKDRV.H	núcleo	si	si	si	2017		
73	include\CLKDRV\ISYSCLK.H	núcleo	si	si	si	972		
74	include\CLKDRV\CLOCKDRV.H	núcleo	si	si	si	2590		
75	include\DBC\iDBC.H	núcleo	si	si	si	4630		
76	include\exp\iXCtx.h	núcleo	si	si	si	830		
77	include\HAL\ICLKHAL.H	HAL	si	si	si	571		
78	include\HAL\ICPUHAL.H	HAL	multi	si	si	5538	0.5	2769
79	include\HAL\iINTHAL.H	HAL	multi	si	si	6097	0.5	3048.5
80	include\HAL\iMemHal.h	HAL	si	si	si	401		
81	include\HAL\intCPU.h	HAL	multi	si	si	3256	0.5	1628
82	include\HAL\sysHAL.h	HAL	multi	si	si	1441	0.5	720.5
83	include\HPC\80x86\I80X86.H[2]	Hard	no	si	si	2638	0.5	1319
84	include\HPC\8253\I8253.H[3]	Hard	no	si	si	1855	1	1855
85	include\HPC\IA32\descript.h[4]	Hard	no	no	si	267	0.75	200.25
86	include\HPC\IA32\HwDebug.h[4]	Hard	no	no	si	848	0.75	636

[1] FAEP: Factor de Aporte al Esfuerzo de Portado.
[2] Archivo no portable común a ambas plataformas, con compilación condicional, por eso FAEP 1.
[3] Archivo no portable común a ambas plataformas (PC), por eso el FAEP es 1.
[4] Archivos específicos de IA32, probablemente se necesite algo similar en otras plataformas, por eso su FAEP alto.

no	archivo	zona	portable	AT16	IA32	tamaño	FAEP[1]	esfuerzo
87	include\HPC\IA32\APIC.H	Hard	no	no	si	2721	0.5	1360.5
88	include\LIST\PRIOLIST.H	núcleo	si	si	si	1183		
89	include\LIST\PRIORITY.H	núcleo	si	si	si	2576		
90	include\LIST\SET.H	núcleo	si	si	si	2097		
91	include\LIST\CLIST.H	núcleo	si	si	si	2453		
92	include\MEM\pKRNLMEM.H	núcleo	si	si	si	397		
93	include\MEM\IKRNLMEM.H	núcleo	si	si	si	9799		
94	include\MEM\BMPPool\BMPPOOL.H	núcleo	si	si	si	4531		
95	include\MEM\ListMngr\LISTMNGR.H	núcleo	si	si	si	4117		
96	include\MEM\MemMngr\MEMMNGR.H	núcleo	si	si	si	3064		
97	include\MEM\Types\MEMTYPE.H	núcleo	si	si	si	1205		
98	include\MEM\Utils\DUMP.H	núcleo	si	si	si	228		
99	include\MEM\ZoneMngr\ZONEMNGR.H	núcleo	si	si	si	4397		
100	include\MIS\iAUXTIME.h	núcleo	si	si	si	990		
101	include\scr\iScreen.h	núcleo	si	si	si	6125		
102	include\scr\pScreen.h	núcleo	si	si	si	3466		
103	include\scr\SCRMEM.H	núcleo	si	si	si	374		
104	include\scr\iKrnlOut.h	núcleo	si	si	si	3257		
105	include\TAL\iKRNSYNC.H	núcleo	si	si	si	8059		
106	include\TAL\PKRNSYNC.H	núcleo	si	si	si	5347		
107	include\TAL\IKRNLINT.H	núcleo	si	si	si	2605		
108	include\XCP\PEXCPTN.H	núcleo	si	si	si	6807		
109	include\XCP\iEXCPTN.h	núcleo	si	si	si	11983		

[1] FAEP: Factor de Aporte al Esfuerzo de Portado.