CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS DEL INSTITUTO POLITÉCNICO NACIONAL

Departamento de Computación

Administración de Interrupciones en Sistemas Operativos de Tiempo Real

Tesis que presenta

Luis Eduardo Leyva del Foyo

Para obtener el Grado de

Doctor en Ciencias:

En la Especialidad de

Ingeniería Eléctrica

Codirectores de la Tesis:

Dr. Pedro Mejía Álvarez
Dr. Dionisio de Niz

México, D.F: Febrero 2008

DEDICATORIA

A mi hija Zoita,

para que siempre luche por sus sueños sin hacer concesión en sus valores.

A mi papá Luis,

que por sus valores, siempre será para mí el ejemplo a imitar.

A mi mamá María Antonia,

que me ha dado amor infinito.

A mi novia Liliana,

por esperarme.

A la memoria de mi abuela Aglae,

ejemplo de fuerza y valor.

AGRADECIMIENTOS

A mis asesores, en especial al Dr. Pedro Mejía Álvarez por aceptarme como su estudiante, por su invaluable apoyo y por su preocupación constante por dar solución a todas las dificultades que enfrentamos para la consecución de estos estudios de doctorado. Todo ello, a pesar de las innumerables dificultades que derivaron de las restricciones que impuso a mi régimen de estudios, las reglamentaciones de mi Universidad y mi país. Sin eso, este trabajo hubiese sido imposible. A él y al Dr. Dionisio de Niz por la confianza que siempre tuvieron en mi trabajo, por introducirme exitosamente en el camino de las publicaciones científicas de alto nivel y por la paciencia, dedicación y maestría con que lo hicieron. Sin ellos, los resultados científicos que exhibimos hoy serían impensables.

La validación de las ideas presentadas en este trabajo, necesitó de un extraordinario esfuerzo de programación en la realización de un micro-núcleo experimental. En este empeño conté con la ayuda de varios de mis estudiantes del Grupo Científico de Programación de Sistemas del Dpto. de Computación de la Universidad de Oriente. En especial quiero agradecer al Lic. William Martínez por las innumerables horas, días y meses que compartimos en la programación del componente de administración de interrupciones y la depuración del núcleo en general, al MSc. Alain Tamayo por la programación del Administrador de Memoria, al Lic. Pablo Rafael López por la ayuda en la programación del componente de comunicación y sincronización, al Lic. Antonio Díaz Tula, al Lic. Germán Mendoza, al Lic. Luís Enrique Rodríguez y al Lic. Yunior Pupo por su ayuda en la programación de otras partes del sistema.

Quiero dar un agradecimiento muy especial a mis padres, María Antonia y Luis Leyva, aunque no son especialistas en computación, nos brindaron a mi y a mis estudiantes un apoyo invaluable durante las largas horas de programación del núcleo. Tarea que se llevó a cabo casi en su totalidad en mi casa y bajo su atención. Sin su apoyo y atención este trabajo no hubiese sido posible. Les agradezco a ellos la ayuda y el amor que siempre me han dado, la educación y los valores que me han transmitido y sin los cuales el logro actual hubiese sido imposible.

Agradezco además a mi hija Zoe Leyva por la comprensión que ha tenido de la necesidad de los muchos meses de separación cuando he estado en México y de las horas de trabajo programando frente a una computadora mientras estoy en Cuba.

Agradezco a mi hermana Rosa María Leyva, por su apoyo a mi carrera y en especial porque, a pesar de sus muchas responsabilidades profesionales y familiares, se encuentre aquí para apoyarme durante la defensa de esta tesis y representando a toda mi familia, a la que le es imposible estar. A mis primas Mapy y Gaby, a mi tía Motin y mi familia en general .

A mi gran amigo el MsC. Rolando Menchaca Méndez, por hacer posible el encuentro con mi asesor el Dr. Pedro Mejía Álvarez en primer lugar y por ofrecerme, de forma desinteresada su casa durante mis estancias en México. Al Dr. Felipe Rolando Menchaca García y su esposa Reina, por acogerme en su familia. Su ayuda fue determinante para que mis estancias en México fuesen posibles.

Al Dr. Bárbaro Ferro del Tecnológico de Monterrey por su invaluable amistad y su apoyo durante todo el desarrollo de mis estudios y mi estancia en México en general. Estas estancias no estuvieron ausentes de problemas de salud. Agradezco a los Doctores Alejandro Benhumea (Odontólogo) y Mariano Sotomayor (Urólogo) por su invaluable ayuda medica.

Al Dr. Carlos Coello Coello porque, a pesar de estar alejado de su área de investigación y pese a sus múltiples ocupaciones científicas y administrativas, siempre se dio el tiempo para seguir mi trabajo. Por la comprensión que mostró ante las dificultades que enfrentamos y su apoyo para superarlas desde su posición como Jefe del Dpto. de Computación del CINVESTAV.

Al Dr. Hugo Cesar Coyote del CIC-IPN, por permitirse el tiempo de revisar mí trabajo y asistir al seminario de investigación de doctorado a pesar de sus responsabilidades administrativas. Tanto a él como al Dr. Héctor Benítez del IIMAS-UNAM y al Dr. Daniel Mosse de la Universidad de Pittsburg por aceptar ser sinodales de este trabajo. A todos ellos y al Dr. José Guadalupe Rodríguez y el Dr. Matías Alvarado por sus valiosos comentarios como sinodales.

A la Dra. Xiaoou Lee, al Dr. Jorge Buenabad, a la Dra. Ana María, al Dr. Adriano de Luca, y al Dr. Sergio Chiapa por su apoyo durante mis estancias en el Dpto. de Computación del CINVESTAV. Al Dr. Alberto Soria, del Dpto. de Control por su apoyo académico y administrativo. Al CONACyT por su apoyo económico. Al CINVESTAV como institución, y a todo su personal, en especial al Dr. Santiago Domínguez por su apoyo como Administrador de la Red. Al Ing. Antonio López Morales Jefe de Relaciones Publicas del CINVESTAV por su apoyo en la tramitación de las visas. A la Secretaria Flor, por su apoyo diario, a ella y a Sofy por su apoyo en los trámites administrativos. A mis compañeros de estudio del Dpto. de computación del CINVESTAV.

A Hector Carmenaty, por su ejemplo de integridad y de amistad y por el valioso apoyo para el logro de estos estudios. A todos aquellos que en Cuba de una forma u otra han ayudado a que pudiera lograr este Doctorado en un centro como el CINVESTAV.

Agradezco además a la Universidad de Oriente y en general a todos los profesores que han contribuido a mi educación en Cuba y en México.

RESUMEN

Los requerimientos de diseño de las aplicaciones de tiempo real son radicalmente diferentes de aquellos de las aplicaciones de propósito general (cómputo científico y de escritorio en entornos de red). Esto demanda de sistemas operativos de tiempo real con características y mecanismos específicos para este segmento de la computación. En particular, en un sistema operativo de tiempo real es necesario poder predecir en todo momento qué actividad está ejecutando el procesador y por cuanto tiempo lo hace. Con este propósito la comunidad de tiempo real ha diseñado una arquitectura de software en donde las actividades a llevar a cabo por las aplicaciones (tareas) se activan por tiempos determinados y según esquemas de planificación predecibles. Sin embargo, en los sistemas operativos de tiempo real actuales, la comunicación entre los dispositivos externos y el procesador no ha cambiado y se sigue dando a través de señales emitidas por el hardware (denominadas interrupciones) que interrumpen a las aplicaciones de forma impredecible para ejecutar rutinas de servicio de interrupción (ISR -- "Interrupt Service Routine"). Este mecanismo, diseñado décadas atrás para los sistemas operativos de propósito general, introduce grandes dificultades para satisfacer los requerimientos de predecibilidad temporal y confiabilidad que demanda el cómputo en tiempo real.

En este trabajo se hace un análisis de las dificultades que presenta el modelo de manejo de interrupciones actualmente en uso en los sistemas operativos de tiempo real. Para dar solución a estas dificultades, se propone un nuevo modelo consistente en una unificación total de todos los tipos de actividades en el sistema (ISRs y Tareas) bajo un esquema común de planificación y sincronización. El análisis de la factibilidad de este modelo integrado pone de manifiesto bajo qué circunstancias es superior al modelo tradicional. Adicionalmente se presenta el diseño de un subsistema de administración de interrupciones de bajo nivel que soporta el modelo integrado y que puede ser configurado para usar diferentes modos de operación sobre las arquitecturas de hardware convencionales. El modelo ha sido implementado dentro del núcleo de un sistema operativo experimental desarrollado como parte de este trabajo y que ha permitido mostrar la viabilidad de su realización; así como, la obtención de evidencias experimentales de sus ventajas.

ABSTACT

The design requirements of real-time applications are radically different from those of general purpose systems. This difference in turn demands of real-time operating systems characteristics and specific mechanisms for this computing segment. Despite this, the mechanisms for handling interruptions of actual real-time operating systems are not more than adjustments to the mechanisms designed decades ago for general purpose operating systems. As a result, real-time operating systems nowadays face great difficulties to meet the requirements of temporal predictability and reliability of these applications.

This thesis provides an analysis of the challenges posed by the interrupt management model currently used in actual real-time operating systems. To provide a solution to these difficulties, we proposed a new model which unifies all types of computing activities in the system (ISRs and tasks) into a joint scheme of scheduling and synchronization. A feasibility analysis was designed for this integrated model to show under which circumstances it is superior to the traditional model.

Additionally, in our work we present the low level design of an interrupt management subsystem that supports the integrated model and that can be configured to be used under different modes of emulation on conventional hardware architectures. The model has been implemented on a custom designed operating system kernel developed as part of our work, to demonstrate the feasibility of its implementation and to obtain experimental evidence of its benefits.

1 Introducción

Denominamos sistemas (o aplicaciones) de tiempo real a aquellos sistemas (o aplicaciones) de cómputo que tienen que satisfacer requerimientos de tiempo de respuesta explícitos. Supóngase, por ejemplo, el sistema de software encargado de ordenar el inflado de la bolsa de aire frontal de un automóvil. En este caso el requerimiento de tiempo de respuesta está dado porque dicha bolsa de aire tiene que estar completamente inflada antes de que transcurran los 20 milisegundos posteriores a la detección del choque. De incumplirse esto, se elimina cualquier utilidad de la bolsa (y del software que la controla) dado que el pasajero ya se habrá impactado contra el tablero.

En la actualidad, los sistemas de tiempo real juegan un papel vital y cada vez más creciente en nuestra sociedad. Los mismos se pueden encontrar en muchos sistemas desde los muy simples, hasta los más complejos. Por ejemplo, pueden hallarse con facilidad en el control de experimentos de laboratorio, control de los motores de autos, sistemas de mando y control automáticos, plantas nucleares, sistemas de control de vuelos, plataformas de lanzamiento espacial y en la robótica. Los sistemas de tiempo real más complejos son muy costosos por lo que en la actualidad los gobiernos e industrias en los países desarrollados emplean miles de millones de dólares en diseñarlos, construirlos y probarlos.

La necesidad de satisfacer requerimientos de tiempo da como resultado que los sistemas de tiempo real demanden, de la plataforma de cómputo que los soportan, requerimientos que difieren radicalmente de aquellos de los sistemas de "propósito general" (estaciones de trabajo, sistemas de escritorio, servidores de red).

Los sistemas de tiempo real también tienen que operar con un alto grado de confiabilidad. Dados los requerimientos de respuesta a tiempo de estos sistemas, un aspecto fundamental de esta confiabilidad es la necesidad de establecer garantías del cumplimiento de sus restricciones de tiempo (plazos de respuesta). Estas garantías sólo se consiguen mediante la realización a priori de cálculos de factibilidad (de planificación) que permiten determinar si se cumplen los plazos de todas las tareas críticas en tiempo. Estos cálculos toman como base la frecuencia de ocurrencia de todos los eventos en el sistema, los algoritmos de planificación de la atención a dichos eventos y la cantidad de tiempo que toma el servirlos (si acaso estos pueden ser servidos). En algunos sistemas, estos cálculos de factibilidad se hacen fuera de línea; mientras que en otros, destinados a ambiente más dinámicos, se hacen en línea conforme varía la carga de trabajo del sistema. En este último caso, si los cálculos arrojan que es imposible dar servicio a tiempo a los sucesos, el sistema debe decidir sobre un plan de acción que garantice la respuesta de las tareas críticas en detrimento de las menos críticas.

Todo el esquema anterior se sostiene bajo dos premisas fundamentales: la capacidad de caracterizar a priori el comportamiento de la carga de trabajo del sistema y la predecibilidad temporal de la plataforma de cómputo en general y del sistema operativo en particular. En

consecuencia, la característica distintiva de un sistema operativo de tiempo real es la predecibilidad. Para el logro de esta predecibilidad, este tipo de sistemas operativos tienen que emplear algoritmos y mecanismos cuyas características temporales puedan ser predicables y verificables (por ejemplo, el conocimiento de los tiempos de ejecución en el peor caso de todos los servios del sistema, o de la disponibilidad de memoria física).

Un aspecto determinante para el logro del requerimiento de predecibilidad del sistema operativo, es el mecanismo mediante el cual se administran los eventos externos que se presentan durante la ejecución de la aplicación. Para el tratamiento de estos eventos externos, la generalidad de los sistemas operativos de tiempo real actuales emplea esquemas de administración de interrupciones que fueron diseñados décadas atrás para sistemas operativos de "propósito general". Estos esquemas tienen como propósito fundamental la respuesta rápida a eventos externos (minimizar la latencia de interrupción) o el caudal de procesamiento[1]; sin embargo, presentan serios inconvenientes para el caso de sistemas de tiempo real confiables. Algunos de los inconvenientes más importantes son:

- La generalidad de los desarrollos teóricos para los análisis de factibilidad de planificación de los sistemas de tiempo real, consideran sólo un único espacio de prioridades para todas las actividades en el sistema. Esta suposición contrasta con el modelo real soportado por el sistema operativo, en el cual las Rutinas de Servicio de Interrupción o ISRs ("*Interrupt Service Routine*") y las tareas poseen espacios de prioridades y algoritmos de planificación independientes. Como consecuencia, el empleo de dos espacios de prioridades independientes afecta severamente la capacidad de predecir el comportamiento temporal del sistema. En los casos en que las ecuaciones de factibilidad incluyen el efecto de estos dos espacios, se deteriora significativamente la cota de utilización que garantiza la factibilidad de planificación del sistema.

- La sincronización entre ISRs y tareas se lleva a cabo mediante la inhabilitación de interrupciones. En consecuencia, los sistemas operativos de tiempo real son incapaces de ofrecer garantías de la latencia de interrupción para el peor caso.

- Existen severas restricciones en cuanto a los servicios del sistema que se pueden invocar dentro de las ISRs. Esto a su vez, trae como consecuencia un aumento de la complejidad de diseño e implementación, que afecta negativamente la confiabilidad del software resultante.

Por lo anterior podemos afirmar que, aunque estos esquemas tradicionales de administración de interrupciones son adecuados para sistemas que demandan una alta capacidad (o caudal) de procesamiento, como por ejemplo los sistemas operativos de red y de bases de datos; así mismo, presentan severas restricciones para el caso de sistemas que requieren un alto grado de confiabilidad y, peor aún, carecen del determinismo necesario para establecer las garantías de respuesta temporal que demandan las aplicaciones de tiempo real. Estas dificultades son de un grado tal, que incluso algunos investigadores han optado por eliminar completamente el uso de las interrupciones en los sistemas de tiempo real.

[1] Número de trabajos procesados por unidad de tiempo.

El propósito de este trabajo es **diseñar nuevos esquemas de tratamiento de interrupciones que sean más adecuados para los sistemas operativos destinados a aplicaciones de tiempo real**, fundamentalmente en el área del logro del determinismo temporal y la confiabilidad.

Para la consecución de nuestro objetivo, partimos de la hipótesis de que, a pesar de que los esquemas y arquitecturas de administración de interrupciones en los sistemas operativos actuales dan como resultado la existencia de dos tipos de actividades asíncronas ejecutables denominadas ISRs y tareas (procesos o hilos), cada una de ellas con esquemas propios de planificación y sincronización, conceptualmente ambas son actividades asíncronas muy similares que se ejecutan como consecuencia de un evento asíncrono (señal de software, vencimiento de tiempo, o señal de petición de interrupción). Es por esto que un mecanismo completamente integrado de administración (sincronización y planificación) de interrupciones y tareas es más adecuado para el logro de los objetivos de determinismo temporal y confiabilidad propios de los sistemas de tiempo real. Como consecuencia de esto, en este trabajo:

- Analizamos las razones por las cuales los esquemas tradicionales de administración de interrupciones y tareas no son adecuados para el caso de sistemas operativos de tiempo real.

- Argumentamos las razones por las cuales, una estrategia completamente integrada para la administración de interrupciones y tareas es más adecuada para el caso de sistemas operativos de tiempo real.

- Hacemos una evaluación del esquema completamente integrado propuesto por nosotros desde el punto de vista de la utilización de la CPU y el tiempo de respuesta a los eventos externos, e cual permite poner de manifiesto bajo que condiciones pudiera considerarse más adecuado para el caso de las aplicaciones de tiempo real.

- Presentamos el diseño de un subsistema de administración de interrupciones transportable a diversas arquitecturas y sistemas operativos que utiliza este modelo y que fue implementado como parte de nuestro micro-núcleo.

- Proponemos estrategias para la implementación de nuestro esquema integrado sobre un hardware de interrupciones PC convencional y presentamos los algoritmos de emulación para estas estrategias.

- Presentamos evidencias experimentales que ponen de manifiesto la viabilidad del esquema integrado y sus ventajas sobre el esquema de administración tradicional.

El resto de este trabajo se organiza como sigue:

En el Capítulo 2 se da una introducción a los sistemas de tiempo real y el análisis de factibilidad de planificación seguido de una panorámica del esquema tradicional de

administración de interrupciones para luego exponer las dificultades que este presenta para la realización de sistemas de tiempo real confiables.

En el Capítulo 3 hacemos una exposición del trabajo relacionado en el área. Se da una panorámica de cómo los distintos sistemas operativos y la comunidad de investigación han estado resolviendo los diferentes inconvenientes que se han presentado en el manejo de interrupciones. Esta exposición además tiene el propósito de poner de manifiesto la esfera de aplicación de estas propuestas y como ellas no han estado específicamente orientadas a lograr el determinismo temporal que demandan las aplicaciones de tiempo real.

En el Capítulo 4 se presenta el mecanismo completamente integrado que se propone en este trabajo; así como, se fundamentan las ventajas que para el caso de sistemas de tiempo real presenta el empleo de este nuevo modelo. Se presenta el diseño de un subsistema de interrupciones de bajo nivel portable que puede ser utilizado como apoyo para la incorporación de este modelo en el núcleo de un sistema operativo de tiempo real. Por último, se hace un contraste del modelo integrado propuesto con las alternativas existentes para el manejo de interrupciones y la evitación de las interrupciones. Este análisis revela como el modelo integrado combina las ventajas de estas alternativas y evita sus inconvenientes.

En el Capítulo 5 se presentan diferentes esquemas de emulación que permiten la realización de nuestro esquema integrado sobre el hardware de interrupciones convencionales de las PC. Se presenta el análisis desde el punto de vista de la factibilidad de planificación de cada una de las estrategias de emulación. Este análisis pone de manifiesto los compromiso entre respuesta temporal en el peor caso y eficiencia de cada una de las variantes de emulación. Adicionalmente se presentan los algoritmos que permiten implementar el diseño del subsistema de interrupciones del capítulo anterior utilizando estos esquemas de emulación.

En el Capítulo 6 se presentan los resultados experimentales recogidos a partir de la implementación del diseño y los algoritmos antes expuestos en un micro-núcleo experimental para aplicaciones de tiempo real. Estos resultados ponen de manifiesto la viabilidad de la implementación del esquema integrado incluso sobre un hardware convencional y su factibilidad desde el punto de vista de la predecibilidad temporal en el caso de las aplicaciones de tiempo real.

Por último, en el Capítulo 7 se ofrecen nuestras conclusiones.

2 Problemática del Manejo de Interrupciones

En este capítulo damos a conocer las dificultades que presentan los mecanismos actuales de manejo de interrupciones para el caso de los sistemas operativos de tiempo real. Primero se da una caracterización del contexto de la investigación en la cual se definen brevemente las características más importantes de los sistemas de tiempo real, se introduce el análisis de factibilidad de planificación y se exponen los requerimientos de los sistemas operativos destinados al soporte de este tipo de aplicaciones. Posteriormente se da una introducción al mecanismo de interrupciones del hardware (sección 2.4) y al soporte que generalmente brindan los sistemas operativos (sean o no de tiempo real) para la administración del mecanismo de interrupciones del hardware (sección 2.5). Todo esto sirve de marco de referencia para finalmente presentar las dificultades que presenta este esquema para el caso de las aplicaciones de tiempo real (sección 2.6).

2.1 Contexto de la Investigación (Sistemas Embebidos y de Tiempo Real)

Dos fuerzas significativas provocan que los sistemas de cómputo penetren cada vez más en la vida cotidiana de las nuevas generaciones de seres humanos: por un lado está la necesidad cada vez más creciente en la sociedad moderna de equipos con mayores prestaciones, más sofisticados e “inteligentes” y por el otro, el rápido avance en el hardware, la miniaturización y la disminución de su costo. Esto trae como consecuencia que cada día la computadora se utilice en nuevas esferas de aplicación como un componente destinado a interactuar y controlar los sistemas y equipos en los cuales se incorpora. Esto ha llevado a la aparición de los denominados sistemas embebidos o empotrados.

Un **sistema embebido** es un sistema de cómputo incluido en otro sistema y que forma parte esencial de él. Dicho en otras palabras, no se percibe como un sistema de cómputo, sinó como el sistema de aplicación en sí. Por ejemplo, un teléfono celular o el inyector de gasolina de un automóvil. Los sistemas embebidos incorporan un conjunto de características significativas que los diferencian de los sistemas de escritorio. Entre ellas, una de las más importantes y que está presente en muchos de ellos es la necesidad de operar en tiempo real.

Un **sistema de tiempo real** es cualquier actividad de procesamiento de información o sistema de cómputo que tiene que responder con restricciones de tiempo y de manera predecible a eventos externos. Stankovic lo define como aquel sistema en el cual la corrección no sólo depende del resultado lógico del cómputo, sino también del tiempo en el

cuál este se produce [88]. Es decir, cada resultado debe darse en un plazo de tiempo específico, no importa si es largo o pequeño, pero de incumplirse con estos plazos el resultado final se considera erróneo.

De manera general, las restricciones de tiempo en un sistema de tiempo real pueden ser arbitrariamente complicadas pero la más común es la existencia de plazos de cumplimiento que pueden clasificarse como **duros** ("*hard*"), **firmes** ("*firm*"), o **suaves** ("*soft*"). Un plazo se dice que es **duro** si la consecuencia de su incumplimiento puede ser catastrófica. Se dice que un plazo es **firme** si el resultado producido por la tarea correspondiente, deja de ser útil expirado el plazo, pero su incumplimiento no provoca consecuencias muy severas. Un plazo que no es ni duro ni firme se dice que es **suave**, esto es, el resultado tiene mayor utilidad si se produce a tiempo pero conserva un valor disminuido pasado el plazo de cumplimiento.

Una concepción errónea muy común y arraigada es la creencia de que un sistema de tiempo real sólo tiene que ser rápido. Básicamente, ser rápido generalmente es una condición necesaria, pero no suficiente. Un sistema de tiempo real necesita cumplir plazos explícitos y ser rápido en el caso promedio no garantiza este cumplimiento. En su lugar, la característica fundamental de un sistema de tiempo real es que su comportamiento debe ser predecible. Esto es, debiera ser posible demostrar en la fase de diseño que todas las restricciones de tiempo de la aplicación se cumplirán siempre que se cumplan ciertas condiciones (incluyendo suposiciones de fallos). Esto requiere el conocimiento previo de las cotas en los tiempos de ejecución y los plazos de todas las tareas de manera que puede ser analizado formalmente. De este modo, el diseñador puede tener una temprana advertencia de la inhabilidad del sistema de satisfacer sus requerimientos temporales y tomar así las acciones correctivas apropiadas. En otras palabras**, tiempo real no es sinónimo de rápido sino de predecible**.

2.2 Sistema Operativo para Sistemas Embebidos y de Tiempo Real

Los requerimientos antes descritos traen como consecuencia la necesidad de contar con sistemas operativos de propósito específicos sobre los cuales poder realizar este tipo de aplicaciones. A este tipo de sistemas operativos se les conoce como sistemas operativos de tiempo real.

La característica fundamental de un sistema operativo destinado a aplicaciones de tiempo real es la **predecibilidad** [90]; es decir que suministre la capacidad de demostrar o probar que se satisfacen los requerimientos, sujeto a las suposiciones que se hagan. En especial, en los sistemas de tiempo real la predecibilidad se refiere a la posibilidad de demostrar el cumplimiento de los requerimientos temporales aún bajo suposiciones del peor caso. Esta cualidad está ausente en los sistemas operativos de propósito general. Un sistema operativo para aplicaciones de tiempo real confiables tiene que satisfacer los siguientes requerimientos:

- Soporta esquemas de planificación y sincronización que limitan el no determinismo propio de los sistemas concurrentes con el objetivo de garantizar las restricciones de tiempo (incluyendo plazos de las tareas).

- Soporta las necesidades de dominios de aplicación altamente confiables (por ejemplo soporte para detección de errores y condiciones excepcionales; así como la supervisión de plazos).

- Suministra un alto grado de control a los programas de usuarios. En un sistema operativo convencional las aplicaciones de usuario tienen un control muy limitado sobre funciones del sistema operativo tales como planificación, administración de memoria, control de dispositivos de E/S. En un sistema en tiempo real, sin embargo, resulta esencial permitir al usuario un control preciso sobre la prioridad y características temporales de las tareas, sobre el uso de paginación o intercambio de procesos y sobre qué procesos deben estar siempre residentes en la memoria principal.

- Suministra al programador un entorno simple y completamente predecible (conjunto restringido de servicios del OS con tiempos de respuesta deterministas).

Un aspecto importante a destacar es que **un sistema operativo de tiempo real no es un sistema de tiempo real, un sistema operativo de tiempo real sólo permite el desarrollo de un sistema de tiempo real**, tener tal sistema operativo no evita que sobre el se desarrolle un sistema que no satisfaga sus requerimientos de tiempo. Si por ejemplo se construye un sistema que necesita satisfacer restricciones de tiempo utilizando un sistema operativo de tiempo real pero que tiene que responder a comunicaciones a través de una red Ethernet, nunca será un sistema de tiempo real duro porque Ethernet como tal no es predecible. Por supuesto si se decide construir una aplicación encima de un sistema operativo con multitarea cooperativa como Windows 3.11, el sistema tampoco será de tiempo real ya que el comportamiento del sistema operativo es impredecible.

En otras palabras, un sistema de tiempo real contiene todos los elementos, incluyendo el hardware, el sistema operativo y el software específico de la aplicación. Un sistema operativo de tiempo real es solo un elemento del sistema completo de tiempo real. La característica fundamental de todos los elementos que lo integran es la predecibilidad.

2.3 Planificación y Análisis de Factibilidad

Los sistemas de tiempo real están compuestos por un conjunto de actividades asíncronas o tareas que se ejecutan de forma concurrente. El planificador de tareas, es la parte del sistema operativo que se encarga de asignar el tiempo de CPU a las distintas actividades del sistema según un algoritmo de planificación. Una característica distintiva de los algoritmos de planificación de tiempo real con respecto a los algoritmos de planificación de los sistemas convencionales es que en los primeros, además de la heurística para la asignación del procesador, es necesario ofrecer un conjunto de modelos analíticos o ecuaciones que permitan determinar de antemano si el conjunto de tareas es capaz de cumplir sus restricciones de tiempo.

Los estándares de sistemas operativos de tiempo real actuales están basados casi en su totalidad en algoritmos de planificación con prioridades estáticas (ver por ejemplo: la

Especificación de Tiempo Real para Ada 95 [8][9][94]; la Especificación de Tiempo Real para Java o RTSJ ("*Real-Time Specification for Java*" [12][28][100], el Estándar POSIX – "*Portable Operating Systems Interface*" de Tiempo Real [32], o las especificaciones ITRON – "*Interfaz for The Real-time Operating system Nucleus*" y µITRON [78]). Para estos algoritmos básicamente existen dos enfoques para la determinación de la factibilidad de planificación:

- Modelos basados en la máxima utilización del procesador que garantiza que el conjunto de tareas ($\tau_1, \tau_2, \ldots, \tau_n$) satisfaga sus plazos. Se distinguen dos enfoques fundamentales:

  Basado en la utilización total del procesador: se apoya en la ecuación básica del Análisis de Tasa Monótona o RMA ("*Rate Monotonic Analysis*"), presentada por Liu y Layland [60]. Esta se aplica a un conjunto de tareas periódicas independientes que se planifican usando el algoritmo de Tasa Monótona (asigna prioridades proporcionales a los períodos – a menor período mayor prioridad). Según este modelo, el conjunto de tareas es factible de planificar si se cumple que:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(2^{1/n} - 1\right) = U(n)$$

  donde C_i y T_i son el tiempo de ejecución y el período respectivamente de la tarea τ_i y n es el número de tareas en el sistema.

  Basado en los puntos de planificación (utilización parcial): La condición anterior es suficiente pero no necesaria. Una condición suficiente y necesaria para el Análisis de Tasa Monótona fue presentada por Lehoczky, Sha y Ding en [52] y está basada en los puntos de planificación. Esta establece que un conjunto de n tareas periódicas independientes, es factible de planificar, para todas las fases entre tareas, si y sólo si:

$$\forall i,\ 1 \leq i \leq n,\ \min_{(k,l) \in R_i} \sum_{j=1}^{i} C_j \frac{1}{lT_k} \left\lceil \frac{lT_k}{T_j} \right\rceil \leq 1$$

  donde $R_i = \{ (k, l) \mid k = 1,..., i;\ l = 1,..., \lfloor T_i / T_k \rfloor \}$

- Modelos basados en la obtención del tiempo de respuesta de las tareas. Este método de análisis de sistemas de tiempo real es aplicable no sólo al caso en que se emplee el algoritmo de planificación de Tasa Monótona sino a cualquier algoritmo con prioridades estáticas.

  La ecuación básica de esta técnica, a la que se denomina **Análisis de Tiempos de Respuesta** [4][42], establece que un conjunto de tareas es factible de planificar para todos los desfases entre tareas, si y sólo si, el tiempo de respuesta de la tarea τ_i es menor o igual al plazo de dicha tarea ($R_i \leq D_i$) y el tiempo de respuesta puede calcularse como:

$$R_i = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

El soporte de los modelos anteriores por parte del planificador del sistema operativo de tiempo real es lo que permite que sobre este se puedan construir sistemas de tiempo real con la certidumbre de que el conjunto de tareas que conforman la aplicación cumpla con las restricciones temporales

2.4 Introducción al mecanismo de Interrupciones

Las **interrupciones** son cambios en el flujo de control, no ocasionados por el programa que se ejecuta, sino por algún otro suceso que necesita el servicio inmediato de la CPU por lo general relacionado con los dispositivos de E/S. Por ejemplo, un programa puede pedirle al controlador de disco que empiece a transferir información y que genere una interrupción cuando acabe la transferencia.

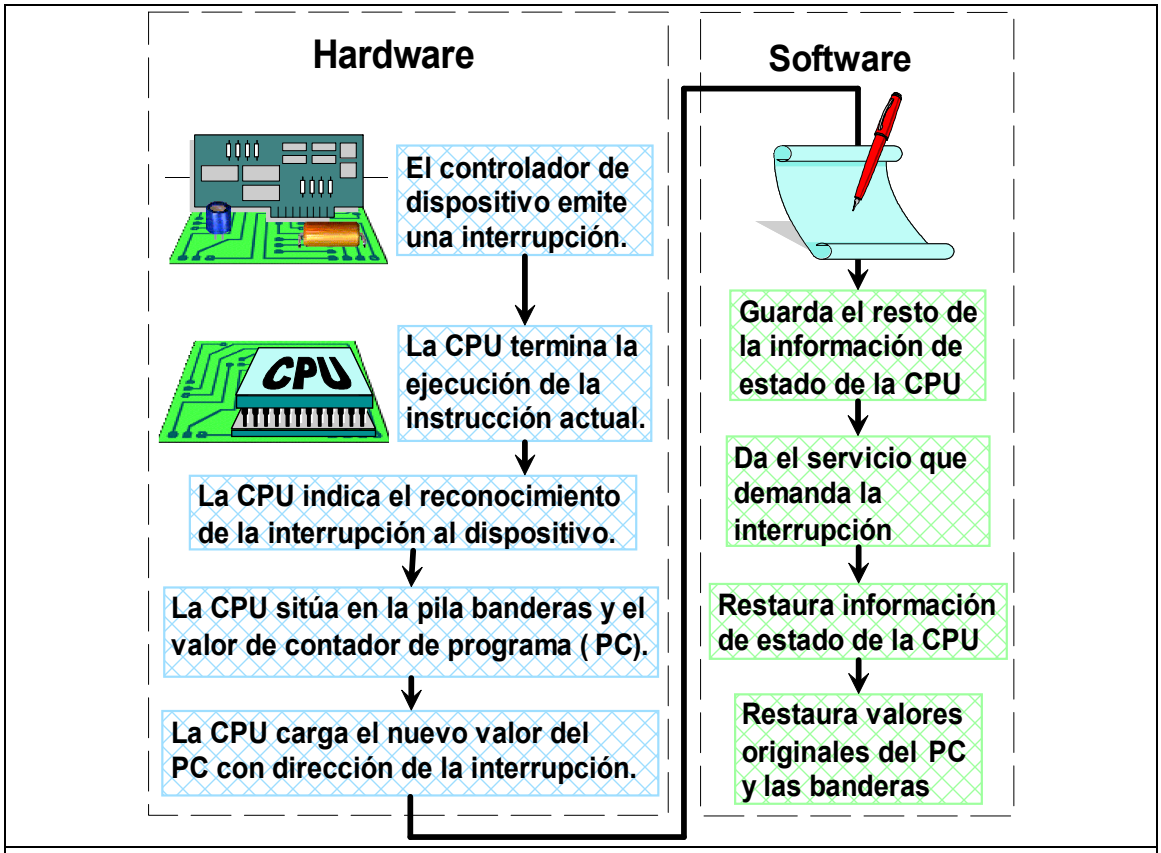


Figura 1 .Procesamiento de una Interrupción

La Figura 1 muestra, en forma muy simplificada, los pasos y los componentes involucrados en el manejo de una interrupción. La señal de petición de interrupción provoca que la CPU detenga el programa en curso, salve su estado (es decir, se guardan todos los contenidos de los registros de la CPU) y transfiera el control a una **Rutina de Servicio de Interrupción**, o **ISR** (del ingles "*Interrupt Service Routine*") la cual realiza alguna acción apropiada para

darle servicio a la petición. Al terminar el servicio de la interrupción, se debe continuar el código interrumpido exactamente en él mismo estado en que estaba cuando tuvo lugar la interrupción, lo cual se logra restaurando los registros internos al estado que tenían antes de la interrupción previamente salvado permitiendo continuar el flujo normal de procesamiento.

Como puede observarse, un concepto clave relacionado con las interrupciones es la **transparencia**. Cuando se produce una interrupción, tienen efecto algunas acciones y se ejecutan algunos códigos, pero cuando todo termina, la computadora se debe regresar exactamente al mismo estado en que se encontraba antes de la interrupción.

2.4.1 Panorámica del Hardware de Interrupciones

El hardware de un sistema de cómputo puede tener muchos controladores de dispositivos de E/S por tanto, el mecanismo de interrupción tiene que permitir identificar del origen de la petición de interrupción. Con ese propósito, por lo general se incluye un determinado número de líneas de petición de interrupción o IRQ ("*interrupt request line*"), cada una asociada con un controlador de dispositivo diferente. A su vez, asociado a cada línea de petición de interrupción existe un conjunto de localizaciones de memoria (vectores de interrupción) que mantienen la dirección de inicio de la rutina de manipulación de interrupción para dicha línea de petición. Cuando un dispositivo específico desea interrumpir, envía una señal por su línea de petición. Con este arreglo, el contador de programa de la CPU se modifica según el vector de interrupción que corresponda.

El hardware asocia cada línea de petición de interrupción con un nivel de prioridad de interrupción. La CPU incluye un registro en el que se almacena la prioridad actual (la prioridad del procesador). Si el nivel de prioridad actual del procesador es mayor o igual que la prioridad de la línea de petición de interrupción, se ignora la interrupción. Si no, la prioridad del procesador pasa a ser la de la línea de interrupción y se transfiere el control al manejador de interrupción correspondiente. Cuando finaliza el manejador de la interrupción, se disminuye el nivel de prioridad del procesador y se reanuda la ejecución de la actividad interrumpida. Observe que el proceso de manejo de interrupción puede ser interrumpido por interrupciones de mayor prioridad. Como las rutinas de atención interrupción están expuestas a la interferencia de las propias rutinas de interrupción, la mejor manera de mantener correcta la administración de interrupciones es asegurarse de que todas las interrupciones sean transparentes.

Para auxiliar al núcleo de la CPU ("*CPU core*") a administrar las diferentes fuentes, es común contar con un hardware de interrupciones auxiliar (el cual pudiera estar externo o integrado al propio chip) que comúnmente recibe el nombre de **Controlador de Interrupciones Programables** o PIC ("*programmable interrupt controller*"). El PIC contiene varias líneas de petición de interrupción IRQ por donde llegan las peticiones provenientes de los distintos dispositivos externos y una salida de interrupción que utiliza para solicitarle una interrupción al núcleo de la CPU.

2.4.1.1 Ciclo de reconocimiento de interrupción

La CPU responde a una petición de interrupción con un ciclo de reconocimiento de interrupción. En la mayoría de las CPUs la respuesta a una interrupción consta de los siguientes pasos:

1. El dispositivo de hardware genera el pulso o señal de petición de interrupción

2. El controlador de Interrupciones Programables prioriza la petición de interrupción en relación con las demás peticiones que podrían haberse emitido de forma simultánea (o estar pendientes) y emite la petición de interrupción al procesador.

3. Si las interrupciones están habilitadas, la CPU responde con un ciclo de bus de reconocimiento de interrupción.

4. En respuesta al reconocimiento de la CPU, el dispositivo externo (o el PIC si estuviese presente) sitúa un vector de interrupción en el bus de datos.

5. La CPU lee el vector y lo utiliza (posiblemente de forma indirecta) para obtener la dirección de la ISR.

6. Por último, la CPU sitúa en la pila el contexto actual, inhabilita las interrupciones, y salta a la ISR.

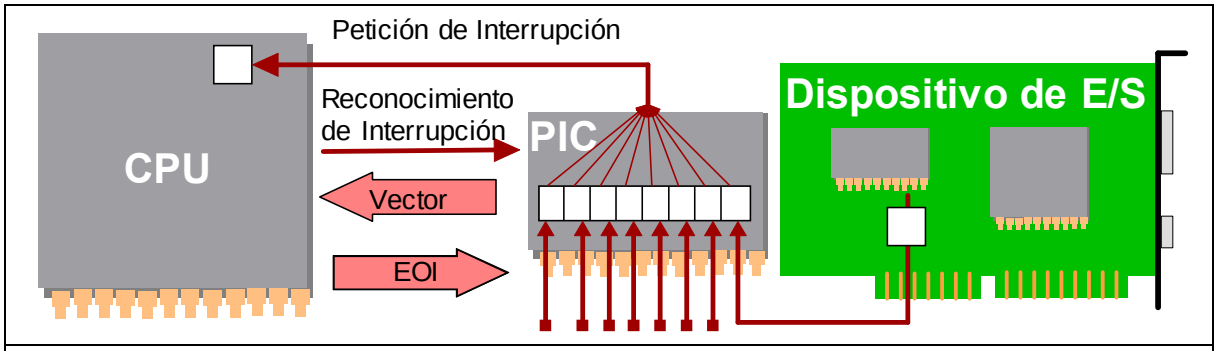


Figura 2. Hardware de Interrupciones

2.4.1.2 Niveles de Control de las Interrupciones

El PIC impone y hace cumplir un esquemas de prioridades a cada una de estas líneas de petición de interrupción (ver Figura 2). Como consecuencia de este arreglo, existen tres niveles de control de las interrupciones:

- Al nivel de CPU puede inhibirse/habilitarse globalmente la capacidad de la CPU de reconocer las interrupciones.

- Al nivel del PIC es posible enmascarar (inhibir) peticiones de interrupciones individualmente y/o sobre la base de sus prioridades. Típicamente, existen uno o más

registros de mascara de interrupción, con bits individuales que permiten o inhiben fuentes de interrupciones individuales.

Adicionalmente, para implementar el mecanismo de prioridades de las interrupciones (y abstenerse de solicitarle una interrupción a la CPU mientras se está dando servicio a una petición de mayor prioridad), el PIC tiene que llevar la pista de que ISR está procesando la CPU en todo momento. Con este propósito, cada vez que la CPU reconoce una petición de interrupción proveniente del PIC, además de enviarle a la CPU un vector de interrupción que identifica cual ISR debe ejecutar, el PIC también registra que dicha petición de interrupción se encuentra en servicio. A su vez, el PIC tiene que conocer cuando la CPU ha finalizado de servir una petición. Con este propósito, luego de que la ISR ejecuta el servicio apropiado (y antes de retornar de la interrupción) tiene que notificarlo al PIC enviándole un comando de Fin de Interrupción o **EOI** ("*End of Interrupt*"). En algunas arquitecturas (como por ejemplo la x86) este comando se lleva a cabo escribiendo de forma explícita un código apropiado a un registro del PIC. En otras arquitecturas (por ejemplo los procesadores Z80 y Z180) este comando lo suministra de forma implícita la misma ejecución de la instrucción de Retorno de Interrupción.

- A nivel de dispositivo, usualmente existe un registro de control de interrupción con bits para habilitar o inhabilitar las interrupciones que el dispositivo puede generar.

  Adicionalmente, muchos dispositivos requieren que la ISR le envíe un acuse de recibo explícito y lo "configuren" para una nueva petición (generalmente ambas cosas se consiguen con un solo comando de respuesta al dispositivo). El acuse de recibo que debe dar el software (la ISR) a la IRQ está dividido en dos partes: la primera parte es la que se acaba de mencionar y está dirigida al dispositivo que emitió la petición; mientras que la segunda, mencionada en el punto anterior, comprende el envío del EOI al PIC. La primera parte es dependiente de la interrupción mientras que la segunda parte es dependiente del PIC, por lo que es común para todos los dispositivos en un sistema.

2.5 Modelo Tradicional de Administración de Interrupciones por el Sistema Operativo

El mecanismo de interrupciones suministra un enlace entre los eventos asíncronos externos y las rutinas de software que le dan servicio. En otras palabras, las interrupciones señalan la llegada de eventos externos que provocan la ejecución de ISRs. Conceptualmente esto es lo mismo que el proceso de señalar cierto evento, como por ejemplo la finalización del uso de recursos o la disponibilidad de un lugar en el buffer que permite la ejecución de determinada tarea (que estaba esperando por el buffer).

A pesar de estas similitudes, con el afán de lograr una mayor eficiencia y una menor latencia en la respuesta a las interrupciones, en general los sistemas operativos ofrecen un conjunto de mecanismos para el tratamiento de interrupciones totalmente independiente de aquellos utilizados para la administración de las tareas o procesos concurrentes. Esta decisión proviene de las diferencias de implantación a nivel del sistema.

Las tareas son una abstracción propia del modelo de concurrencia soportado por el núcleo del sistema y la responsabilidad de su administración recae completamente en el mismo. Tradicionalmente, el núcleo brinda servicios para la creación, eliminación, comunicación y sincronización entre tareas [55].

Las interrupciones, por su parte, son una abstracción del hardware de la computadora y la responsabilidad de su administración recae fundamentalmente en el mecanismo de interrupciones del hardware. Este suministra un conjunto de funciones entre las que se encuentran: la asignación de ISRs suministradas por la aplicación a diferentes señales de petición de interrupción; conmutación de contexto mediante la salva y restaura automática de un contexto mínimo de la CPU; la habilitación/inhabilitación de peticiones de interrupción específicas mediante una máscara de interrupción; y la planificación de las interrupciones según un esquema de prioridad en hardware.

La mayoría de los sistemas operativos se limitan a suministrar un conjunto de servicios que permiten a las aplicaciones la ejecución de éstas y posiblemente otras operaciones a un nivel de abstracción ligeramente superior. En conjunto, estos servicios le dan al usuario la posibilidad de controlar de forma directa esta abstracción, quizás con un mayor nivel de seguridad. Sin embargo, ninguna de estas operaciones es necesaria para las tareas ordinarias no relacionadas con interrupciones.

Al estar fuertemente apoyado en el hardware, este enfoque tiene como ventaja un mejor desempeño del sistema y un menor costo operativo, razón por la cual ha sido el método utilizado en la mayoría de los sistemas operativos comerciales, estén estos destinados al mercado de propósito general o incluso al mercado de aplicaciones de tiempo real.

Sin embargo este enfoque da como resultado que estos sistemas operativos suministren en realidad dos formas de actividades asíncronas: los procesos o tareas y las ISRs. Asociado a esto, se suministra un conjunto exclusivo y restringido de primitivas con sintaxis y semántica diferentes que pueden utilizarse sólo para el código perteneciente a una forma específica; así como, un conjunto de políticas y algoritmos de administración independientes para cada caso. La Figura 3 muestra algunos ejemplos de esta dicotomía.

	Tareas	**Manejadores de Interrupción**
Iniciación	Creación de Tareas	Instalación de Rutina de Atención a Interrupción
Terminación	Eliminación de Tares	Desinstalación de Rutina de Atención a Interrupción.
Conmutación de Contexto	Automática y completamente transparente al usuario	Requiere ayuda del usuario para salvar y restaurar el contexto.
Sincronización	Explícita mediante mecanismos IPC (semáforos)	Semiautomática mediante Prioridades de Hardware
Planificación	Determinada por la política del sistema operativo.	Determinada por el Mecanismo de Interrupciones del Hardware

Figura 3. Diferentes primitivas para la misma función conceptual.

2.5.1 Esquema de Prioridades

Las ISRs son actividades iniciadas por una fuente de interrupción que demanda un servicio y su planificación corre a cargo del mecanismo de control de interrupciones del hardware de acuerdo a sus **prioridades de hardware**. Las tareas por su parte, se activan por eventos generados por software (que incluyen eventos periódicos) y su planificación la lleva a cabo el planificador del sistema operativo según sus **prioridades de software**. Este esquema con **niveles de prioridad de hardware** situados por encima de los **niveles de prioridad de software**, trae como consecuencia que estas últimas se ejecuten sólo cuando no existan ISRs listas para utilizar la CPU.

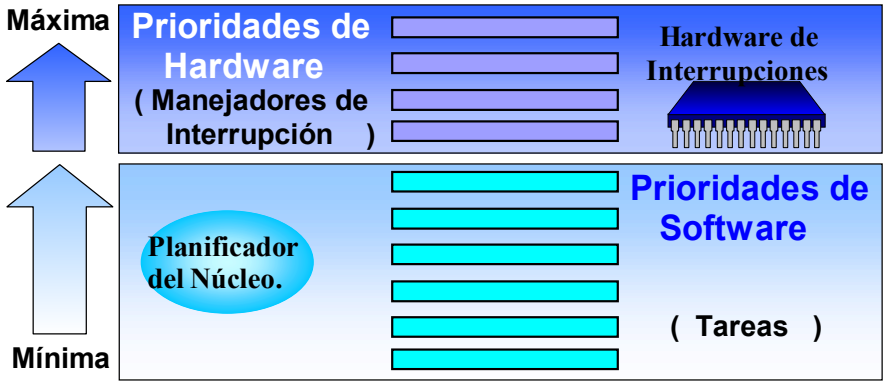


Figura 4. Prioridades en el esquema tradicional

En sistemas de propósito general, las tareas no tienen requerimientos estrictos de respuesta a tiempo y las únicas actividades con requerimientos de respuesta en tiempo real, son las ISRs. En consecuencia, esta disposición tiene sentido, pues logra una baja latencia en las respuestas a las peticiones de interrupción, permitiendo que no se pierdan los datos sólo por el hecho de que otra tarea se esté ejecutando.

2.5.2 Sincronización de Exclusión Mutua entre Actividades Asíncronas

Al analizar los mecanismos de sincronización entre las diferentes actividades asíncronas que existen en el sistema, hay que considerar tanto la sincronización entre actividades del mismo tipo, como entre actividades diferentes (ISRs y tareas). Según se muestra en la tabla de la Figura 5 lo cual se discute a continuación.

En el caso de las tareas, la sincronización entre ellas se realiza usando cualquiera de los mecanismos existentes para la sincronización entre tareas (semáforos, mutexes, mensajes, buzones, etc.) que brinde el modelo de sincronización del sistema operativo.

	Procesos o tareas	**Interrupciones**
Procesos o Tareas	Mecanismos de IPC (semáforos)	Inhabilitación/habilitación
Interrupciones	Nivel de Prioridad Superior.	Prioridades de hardware

Figura 5. Mecanismos de sincronización entre diferentes actividades asíncronas

Para el caso de las ISRs, la sincronización entre ellas se reduce a la sincronización de exclusión mutua y la consiguen apoyándose exclusivamente en su propio esquema de prioridades. Sin embargo, aún así existen alternativas:

- El caso más simple (por ejemplo en Windows CE versión 2.0 [95]) es que todos los niveles de interrupción están habilitados de forma tal que expropian a cualquier tarea. Sin embargo, al ejecutarse cualquier ISRs, el resto de las peticiones de interrupción están inhabilitadas. En esta configuración todas las ISRs pueden suponer exclusión mutua. Este arreglo es adecuado en sistemas en que las peticiones de interrupción requieren poco procesamiento, pero no es adecuado para sistemas de tiempo real.

- El diseño más utilizado consiste en asignarle a cada petición de interrupción una prioridad y permitir las peticiones de mayor prioridad durante la ejecución de una ISR. En este esquema, conocido como interrupciones anidadas, cada ISR se ejecuta como una sección crítica con respecto a las ISRs de menor prioridad y las tareas. Además, debido a que las interrupciones de la misma prioridad están inhabilitadas, una ISR no tiene que ejecutarse bajo exclusión mutua consigo misma.

Sin embargo, las dificultades cruciales en el aspecto de la sincronización no están en la sincronización entre entidades del mismo tipo, sino en la sincronización cruzada entre tareas e ISRs. Las ISR y las tareas se comunican a través de memoria compartida y todas las operaciones ejecutadas sobre ésta tienen que ser mutuamente exclusivas. Obsérvese que, aunque las ISRs son secciones críticas automáticas con respecto a las tareas, lo contrario no es cierto. Los mecanismos para garantizar el acceso exclusivo a las secciones críticas entre tareas, no garantizan acceso exclusivo contra las ISRs.

De ninguna forma, una ISR puede bloquear a la tarea interrumpida. Por tanto, la exclusión mutua entre ISRs y tareas se puede conseguir sólo inhabilitando las interrupciones mientras se accede al dato compartido. Sin embargo, no es conveniente inhabilitar de forma innecesaria aquellas interrupciones de mayor prioridad que no interfieren con la sección crítica de la tarea. Ello afectaría innecesariamente la sensibilidad de respuesta ("*responsiveness*") del sistema a las interrupciones urgentes. Por tanto, los segmentos de código que usan el nivel de interrupción de la CPU para sincronización deben situarlo sólo hasta el nivel de prioridad de la ISR con la que pudieran interferir.

En sistemas operativos tradicionales de tipo UNIX, este esquema de sincronización es adecuado. Ello se debe a que no es posible la ejecución de manejadores de interrupción en modo usuario, y a que las aplicaciones no pueden modificar el nivel de prioridad de la CPU (se ejecutan con todas las interrupciones habilitadas). En estos sistemas, sólo es posible actuar sobre el nivel de interrupción de la CPU cuando las tareas ejecutan el código del sistema operativo en modo núcleo. Debido a que el núcleo no es expropiable, no existe ningún peligro de que ocurra una conmutación de contexto mientras ha sido alterado el nivel de interrupción actual de la CPU. Cualquier elevación del nivel de interrupción de la CPU, será restaurada antes de que sea posible una conmutación de contexto restringiendo esta modificación al contexto de la tarea actual.

2.6 Dificultades al Usar el Modelo Tradicional en Núcleos para Sistemas Embebidos y de Tiempo Real.

Aunque muchos sistemas operativos destinados a aplicaciones embebidas y de tiempo real han optado por utilizar el modelo de administración de interrupciones antes descrito (sección 2.5), su empleo presenta serias dificultades en este entorno. En esta sección damos una panorámica de las dificultades que este esquema presenta en cuatro esferas fundamentales: la sincronización entre los diferentes tipos de actividades asíncronas, su integración con un mecanismo estructurado de tratamiento de situaciones excepcionales, las prioridades de planificación y el establecimiento de cotas en la latencia de interrupción.

2.6.1 Dificultades asociadas con el mecanismo de sincronización

En los sistemas embebidos, la diversidad de dispositivos periféricos para la interacción con el entorno hace inadecuado el modelo de manejadores de dispositivos al nivel del núcleo. Además, en el caso de los sistemas de tiempo real, un núcleo no expropiable puede posponer arbitrariamente la conmutación de contexto, dificultando seriamente la capacidad de predecir el comportamiento temporal. Si el núcleo se hace expropiable, o si se le permite a las tareas de usuarios actuar directamente sobre el nivel de interrupción de la CPU, es necesario suministrar protección contra dos errores comunes: el error del cierre de interrupción roto y el error de las interrupciones dobles/retardadas [97].

2.6.1.1 Errores Debido al Mecanismo de Sincronización de Exclusión Mutua

El **problema del cierre de interrupción roto** está asociado a la necesidad de rastrear los niveles de interrupción a través de escenarios con expropiación de tareas. Esta situación se ilustra en la Figura 6.

En la figura una tarea de alta prioridad T_H se bloquea esperando un evento externo que será señalado por la interrupción I_H (instante t_1). Ello provoca que comience a ejecutarse una tarea T_L de baja prioridad la cual elevará el nivel de interrupción de la CPU hasta el nivel correspondiente a I_L con el propósito de acceder a una sección crítica que comparte con dicha ISR (instante t_2). Si embargo, mientras se ejecuta en la sección crítica se produce una interrupción de mayor prioridad que provoca la ejecución de la ISR I_H. Esta ISR señala el evento que desbloquea a la tarea T_H (instante t_4) provocando que esta expropie a la tarea de baja prioridad T_L dentro de su sección crítica. Como parte de la conmutación de contexto hacia la tarea T_H, el núcleo establece el nivel de interrupción de la CPU asociado a la misma que pudiera ser inferior al nivel de interrupción de la tarea interrumpida (en este caso se habilitan todas las interrupciones). Mientras se ejecuta esta nueva tarea T_H, puede ocurrir una interrupción del nivel de prioridad que se quería evitar (I_L en el ejemplo). Para el ejemplo de la figura, de hecho en el instante t_4 se produce la interrupción I_L provocando que se ejecute su ISR asociada y violando el requerimiento de exclusión mutua establecido por la tarea T_L. En efecto, la conmutación de contexto ha eliminado el cierre sobre la sección crítica (entre los instantes t_4 y t_7 de la figura).

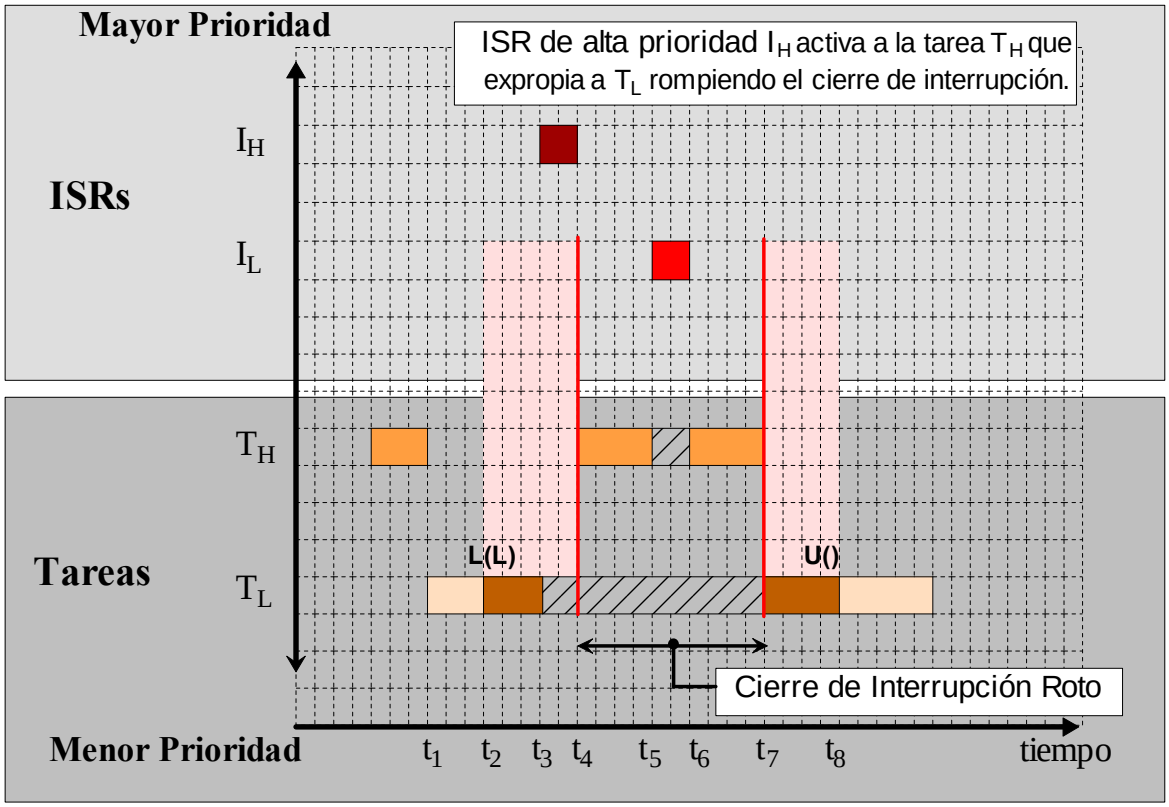


Figura 6. Cierre de Interrupción Roto

Una alternativa que se ha empleado, es mantener el estado de las interrupciones sin cambios al hacer una conmutación de contexto. Aunque esto resuelve los problemas de seguridad, se hace menos predecible el comportamiento global del sistema porque las tareas se ejecutarán con varios estados de interrupción, dependiendo de la tarea a la que expropiaron.

Para evitar este problema, muchos núcleos obligan a que las tareas siempre eleven el nivel de prioridad de la CPU al más alto posible. Esto inhabilita todas las interrupciones evitando que una conmutación de tareas destruya el cierre de interrupción.

2.6.1.2 Errores Debido al Mecanismo de Sincronización de Condición

Comúnmente una ISR hará al menos una llamada al núcleo para indicar la ocurrencia de algún evento. Las operaciones de este tipo (por ejemplo un ***signal*** sobre un semáforo) no pueden bloquear a la tarea interrumpida. Sin embargo, pueden provocar que se ponga lista una tarea de mayor prioridad. Si se ejecuta la conmutación de contexto, antes de que finalice la ISR, el resto de la misma no se ejecutará hasta que la tarea interrumpida (en el contexto de la cual se está ejecutando la ISR) sea ejecutada nuevamente en la CPU. Esto postergará la actualización de variables importantes del sistema, dejándolo en un estado inestable. En consecuencia, si estos servicios se invocan dentro de una ISR, el núcleo tiene que posponer cualquier conmutación de contexto hasta que la ISR haya finalizado, instante en que tiene que ejecutar cualquier conmutación de tarea pendiente. Por tanto estos

servicios se tienen que comportar de forma distinta si son invocados desde una ISR o desde una tarea. Existen dos alternativas para detectar esta diferencia:

1) **Protocolo EnterISR-LeaveISR**: requiere que la ISR de la aplicación salve los registros de la CPU y le notifique al núcleo (mediante un servicio como **enterISR()**) que se ha iniciado una ISR. El núcleo incrementará un contador que utiliza para determinar el nivel de anidamiento de ISRs. Justo antes de finalizar, el código de la ISR invoca a otro servicio para notificar su salida (por ejemplo **leaveISR()**). Este disminuye el contador de anidamiento y al llegar a cero, si hay alguna tarea de mayor prioridad lista, ejecuta la conmutación de contexto, de lo contrario reanuda la tarea interrumpida. La dificultad de este esquema es que pudiera ocurrir una interrupción de mayor prioridad antes de que se atienda la llamada **enterISR()**. Si esto sucede y se ha solicitado una conmutación de tarea, la primera ISR no se completará. Las consecuencias de esta implementación dependen del hardware del sistema:

   - Error de **doble interrupción**: se producirá en sistemas con interrupciones activadas por nivel y en los cuales el dispositivo que levantó la interrupción original la seguirá sosteniendo, por lo que se volverá a activar cuando la nueva tarea se comience a ejecutar (suponiendo que se ejecuta con interrupciones habilitadas).

   - Error de **demora excesiva**: se producirá en sistemas con interrupciones activadas por flanco. Ésta no se vuelve a activar, pero su servicio se reanudará sólo cuando la tarea que se estaba ejecutando al inicio de todo el proceso pase nuevamente a ser la tarea de mayor prioridad.

   Este esquema (protocolo EnterISR-LeaveISR) es utilizado en muchos sistemas operativos de tiempo real como por ejemplo OSEX [70] y μC/OS [48].

2) **Invocación indirecta de la ISR del usuario**: la interrupción transfiere el control directamente al núcleo del sistema. Esto permite que éste determine si está dentro de una ISR inspeccionando el indicador de estado de interrupciones de la CPU salvado automáticamente en la pila al ocurrir la interrupción. Hecho esto, el núcleo invoca a una rutina de manejo de interrupción suministrada por la aplicación para dar servicio a la interrupción la cual le devuelve el control (al núcleo) al finalizar. Al regreso, si hay tareas de mayor prioridad activas, se ejecutará la conmutación de contexto sólo si no quedan ISRs pendientes de terminar.

   La dificultad de este esquema está dada porque una tarea regular pudiera haber elevado el nivel de prioridad de interrupción para proteger una sección crítica. Si en el transcurso de ésta se produce una interrupción de mayor prioridad, que activa a una tarea de mayor prioridad, el código de salida de la ISR supondrá (erróneamente) que se estaba ejecutando una ISR (ya que se guía por el nivel de interrupción) y pospondrá cualquier conmutación de tarea. Obsérvese que, aunque este comportamiento resuelve también el error del cierre de interrupción roto (ver Figura 6) y por tanto es correcto desde el punto de vista lógico introduce el problema de la conmutación pendiente y por tanto si constituye un problema para el comportamiento (corrección) temporal del sistema.

Si la tarea restaurase directamente el nivel de interrupción de la CPU, no existiría forma de verificar si hay alguna conmutación pendiente y la tarea de alta prioridad se demoraría indefinidamente. El resultado es una forma extrema de inversión de prioridad que hace al sistema completamente impredecible temporalmente. Nuevamente, se han usado dos alternativas para evitar este error de "conmutación muy tarde":

- **Protocolo de habilitación/inhabilitación de interrupciones**: las tareas sólo sitúan el nivel de interrupción de la CPU al nivel más alto. Al no haber ISRs tampoco podrán ponerse listas tareas de mayor prioridad, por tanto no habrán conmutaciones pendientes.

- **Suministro de primitivas del núcleo**: las tareas no modifican directamente el nivel de interrupción de la CPU, sino que invocan un servicio del núcleo para restaurarlo. El núcleo puede entonces verificar si, al restituirse al nivel más bajo de prioridad, hay conmutación de tareas pendiente, y si es el caso ejecutarlas. Esta solución introduce una menor latencia en la conmutación de contexto y es la utilizada en SSX5 [97].

Sin embargo, independientemente de la magnitud de la latencia de conmutación de contexto que introduzcan cualquiera de las alternativas anteriores, el comportamiento temporal de las mismas es muy difícil de modelar y por tanto de predecir.

2.6.1.3 Dificultades asociadas a la diversidad de mecanismos de sincronización

Las diferencias existentes entre los mecanismos de sincronización utilizados según el tipo de actividad asíncrona, trae como consecuencia una gran diversidad de situaciones particulares para la cooperación entre éstas, en donde sólo debiera existir un número limitado. Como ejemplo, supóngase la sincronización de condición entre las ISRs y las tareas que se apoya en un esquema productor-consumidor pero sin posibilidad de bloqueo para el caso de las ISRs. Esta particularidad introduce todo un conjunto de variantes para manejar los casos que normalmente se tratan suspendiendo al productor o al consumidor y que ahora se tienen que tratar ya sea desechando el dato o situando datos más frescos o volviendo a obtener los datos anteriores (entre otras posibilidades). Esto, en dependencia de la aplicación. Como consecuencia el programador tiene que manejar una cantidad innecesariamente grande de patrones comunes de diseño [55]. Esta amplia diversidad, en última instancia hace más probable la ocurrencia de errores de diseño, afectando adversamente la confiabilidad del software.

2.6.2 Dificultades Asociadas con la Utilización de un Mecanismo Estructurado de Manejo de Excepciones

Otra dificultad asociada al modelo de dos tipos de actividades asíncronas, aparece al integrarlo con un mecanismo estructurado de manejo de excepciones [54], en donde las excepciones se propagan a través de la cadena de llamadas a subprogramas. Si ocurre una excepción dentro de una ISR, se propagaría al manejador de excepciones actual de la tarea

interrumpida, el cual no está preparado para tratar excepciones ocurridas en actividades que no tienen relación con éste.

La solución consiste en hacer que el mecanismo de propagación de excepciones verifique si la propagación va a salir de una ISR y si es el caso, detenerla y sólo abortar la ISR. Esto provoca la necesidad de situar y eliminar un marco de excepción como parte del protocolo de entrada y salida de la ISR, además de que dichas excepciones pasarían inadvertidas, lo que constituye una dificultad para la tolerancia a fallos.

2.6.3 Dificultades asociadas a la existencia de dos espacios de prioridades independientes.

La dificultad asociada al esquema de dos espacios de prioridades independientes en el caso de aplicaciones de tiempo real, radica en la suposición de que en ningún momento, los requerimientos de ejecución a tiempo de una tarea, tendrán mayor importancia que los de una ISR. Esta suposición, válida en sistemas de propósito general, no se sostiene en sistemas de tiempo real, en donde además, los requerimientos de respuesta a tiempo para algunas interrupciones pueden estar incluso en el mismo rango que los de las tareas con altas frecuencias de activación (por ejemplo en una medición periódica).

Ambos espacios de prioridades pueden interactuar de forma que interfieran el uno con el otro. En específico, las tareas de mayor prioridad del sistema quedan constantemente bajo la interferencia de los eventos de hardware necesarios sólo para tareas de baja prioridad. Es posible que las tareas de baja prioridad o incluso tareas con requerimientos de tiempo real suave que estén asociadas a interrupciones de E/S (por ejemplo la atención al operador) no puedan ejecutarse debido a sobrecargas temporales y; sin embargo, sus respectivas interrupciones se siguen produciendo. Esta situación afecta significativamente la capacidad de cumplir con los plazos de las tareas y se manifiesta como una disminución en la cota máxima permisible de utilización de la CPU para que el conjunto de tareas sea factible de planificar.

2.6.4 Dificultad asociada con el logro de latencias de interrupciones acotadas

Aunque los argumentos antes expuestos en contra del modelo tradicional de manejo de interrupciones son ya de por sí muy importantes, no son los únicos. Quizás el argumento más significativo contra de este modelo se puede encontrar en su misma razón de ser: disminuir al mínimo la latencia de las interrupciones. El único parámetro determinante en la latencia de interrupción sobre el cuál puede actuar el núcleo del sistema operativo es el tiempo en que se inhabilitan las interrupciones. Con el propósito de minimizar la latencia de interrupción estos sistemas hacen todo su esfuerzo por inhabilitar las interrupciones sólo por períodos de tiempo muy breves. Sin embargo, simultáneamente, este diseño no puede impedir que las aplicaciones inhabiliten las interrupciones ya que es la única forma posible de sincronización entre tareas e ISRs.

En realidad, la respuesta del sistema como un todo a las interrupciones no puede ser mejor que el tiempo máximo por el cual se inhiben las interrupciones en cualquier parte del sistema. Como la aplicación puede inhabilitar las interrupciones por más tiempo que el

núcleo, si ello ocurre, la latencia de interrupción en el peor caso será la suma de la latencia introducida por la CPU y el peor caso del tiempo con interrupciones inhabilitadas de la aplicación. En conclusión, lo más que puede hacer el núcleo es establecer una cota mínima en la latencia de interrupción, pero nunca garantizar la latencia de interrupción en el peor caso. Esta última, queda siempre en manos de aplicación.

2.7 Resumen

El modelo de administración de interrupciones actualmente en uso en los sistemas de tiempo real arroja una rápida respuesta a los eventos externos y un menor costo operativo. Sin embargo, presenta serias dificultades, las cuales exponemos a continuación.

Problemática asociada a los dos espacios de prioridades. En el modelo tradicional, las (peticiones de) interrupciones siempre tienen mayor prioridad que cualquier tarea o proceso de cómputo. Por el contrario, en los sistemas de tiempo real, los requerimientos de tiempo de respuesta de algunas ISRs pueden estar incluso por debajo que cualquier tarea del sistema. En este caso, ambos espacios de prioridades pueden interactuar de forma que interfieran el uno con el otro. Específicamente, bajo el modelo tradicional, las tareas de mayor prioridad podrían quedar bajo la interferencia de eventos de hardware necesarios sólo para tareas de baja prioridad. Por otro lado, tareas de baja prioridad asociadas a interrupciones (ejemplo, la atención al operador) podrían no ejecutarse debido a sobrecargas temporales y sin embargo, sus respectivas ISRs si se ejecutan. Este comportamiento afecta la capacidad de cumplir los requerimientos de tiempo real del sistema y se manifiesta como una disminución en la máxima utilización permisible de la CPU para que el sistema sea factible de planificar.

Problemática asociada a la latencia de interrupciones. Quizás el argumento más significativo en contra del modelo tradicional se puede encontrar en su propósito fundamental: *disminuir al mínimo la latencia de las interrupciones*. Con el propósito de minimizar esta latencia, el núcleo inhabilita las interrupciones sólo por períodos de tiempo muy breves. Sin embargo, este diseño no puede impedir que las aplicaciones inhabiliten las interrupciones, ya que es la única forma posible de sincronización entre tareas e ISRs. En realidad, el tiempo de respuesta del sistema a las interrupciones no puede ser menor que el tiempo máximo por el cual se inhiben las interrupciones en cualquier parte del sistema. Dado que la aplicación puede inhabilitar las interrupciones más tiempo que el núcleo, la latencia de interrupción en el peor caso será la suma de la latencia introducida por la CPU y el tiempo de inhabilitación de interrupciones en el peor caso de la aplicación. En conclusión, en el modelo tradicional lo más que puede hacer el núcleo es establecer una cota mínima en la latencia de interrupción, pero nunca garantizar su peor caso.

Problemática asociadas al mecanismo de exclusión mutua. Cuando una tarea de baja prioridad, para acceder a una sección critica que comparte con una ISR de nivel medio, eleva el nivel de interrupción hasta ese nivel, puede ocurrir una interrupción de nivel alto que active a una tarea de alta prioridad y expropie a la tarea de baja prioridad. Esto disminuirá el nivel de interrupción de la CPU, destruyendo el cierre de interrupción de la tarea de baja prioridad. Para evitar esta situación el núcleo podría mantener el estado de las interrupciones sin cambios al hacer una conmutación de contexto. Esto afecta la

predecibilidad del sistema porque las tareas se ejecutarán con varios estados de interrupción, dependiendo de cual haya sido la tarea que hubieran expropiado. La alternativa es obligar a que las tareas siempre sitúen el nivel de interrupción más alto posible, evitando así cualquier conmutación de contexto. Sin embargo, esta solución incrementa la latencia en la conmutación de contexto.

Problemática asociada a la sincronización de condición. Comúnmente una ISR hará al menos una llamada al núcleo para indicar la ocurrencia de algún evento. Esta llamada puede poner en listo a una tarea de mayor prioridad. Si se ejecuta la conmutación de contexto, antes de que finalice la ISR, el resto de la misma no se ejecutará hasta que la tarea interrumpida sea ejecutada, dejando al sistema en estado inestable. En consecuencia, si estos servicios se invocan dentro de una ISR, el núcleo tendrá que posponer cualquier conmutación de contexto hasta que la ISR finalice. Todas las soluciones existentes provistas para el modelo tradicional para resolver este problema y que garantizan la corrección lógica introducen una excesiva inversión de prioridad por demoras en la conmutación de contexto o exhiben un comportamiento temporal muy difícil de modelar y por tanto de predecir.

Problemática asociada a la diversidad de mecanismos de sincronización. Las diferencias existentes entre los mecanismos de sincronización utilizados según el tipo de actividad asíncrona, trae como resultado una gran diversidad de situaciones para la cooperación entre estas, donde solo debería existir un número limitado. Esta situación, produce un aumento en la complejidad de la solución de las interacciones entre estas. Esta situación hace más probable la ocurrencia de errores de diseño afectando adversamente la confiabilidad del software.

3 Antecedentes y Trabajos Relacionados

En este capítulo presentamos el modelo de manejo de interrupciones utilizado por varios sistemas operativos. Se comienza por un análisis del modelo clásico de manejo de interrupciones de Unix seguido por el esquema utilizado por los sistemas operativos más modernos destinados a entornos de red y aplicaciones multimedia. Durante la exposición se ponen de manifiesto las dificultades fundamentales que cada uno de esos modelos presentaron y cómo estas se han ido resolviendo en los sistemas más recientes y por la comunidad de investigación. Posteriormente se analizan los esquemas utilizados en la actualidad por los sistemas embebidos y de tiempo real. A lo largo de esta exposición se pone de manifiesto como los sistemas operativos de tiempo real actuales han adoptado modelos de manejo de interrupciones diseñados décadas atrás para los sistemas de propósito general. Por último, concluimos destacando que ninguna de estas soluciones resuelve todos los problemas planteados en el capítulo anterior (sección 2.6).

3.1 Tratamiento de las Interrupciones en los sistemas Unix Clásicos

La mayoría de los sistemas de tiempo compartido se basan en el concepto de interrupciones. Los mejores exponentes de este tipo de sistemas operativos son los sistemas Unix clásicos (Unix Sexta Edición AT&T [59] y versiones de Berkeley previas al BSD 4.2 [72]).

3.1.1 Núcleo dividido en dos mitades

En los sistemas Unix clásicos el software del sistema operativo (y de manejo de los dispositivos) estaba dividido en una **mitad superior** (“***top half***”) y una **mitad inferior** (“*bottom half*”)[2]. La mitad superior suministra servicios a los procesos y se ejecuta de forma procedural en el contexto del proceso actual en respuesta a llamadas al sistema. Por su parte, la mitad inferior incluye todo el código que se ejecuta como parte de las ISRs. Estas se ejecutan de forma asíncrona (“*asynchronous*”) con respecto a la mitad superior y en el contexto de cualquier proceso que se encuentre activo en el sistema cuando se recibe la petición de interrupción. Las mitades superior e inferior del núcleo se comunican a través de estructuras de datos, generalmente organizadas alrededor de colas de trabajo. Esta arquitectura ha sido ampliamente utilizada por muchos otros sistemas operativos, al punto

[2] No confundir esto con los manejadores de interrupción “*top half”* y “*bottom half*” de Linux. La terminología “*top halt*” y “*bottom half*” fue acuñada por el Unix BSD [64] para referirse al código de la mitad superior e inferior del núcleo respectivamente y es en este sentido (completamente diferente al significado Linux) que se usa aquí.

que Schmidth y Cranor acuñaron el término Half-Sync/Half-Async para referirse a este patrón arquitectónico [84].

3.1.2 Núcleo no expropiable

Para evitar el acceso concurrente a las estructuras de datos internas del núcleo por diferentes procesos (y los problemas de corrupción de datos derivados de ello), el núcleo de los sistemas Unix tradicionales es **no expropiable** ("*non-preemptible*"). Esto significa que nunca se le quita la CPU a un proceso mientras se ejecuta en la mitad superior del núcleo (dentro de una llamada al sistema) para darle el control a otro proceso. Esto no sucede ni siquiera cuando la interrupción del reloj detecta que ha vencido el cuanto de tiempo del proceso actual mientras está dentro del núcleo. Si esto sucede, la replanificación (o invocación del planificador para llevar a cabo la conmutación de contexto) no tiene lugar inmediatamente; en su lugar, la ISR del reloj simplemente activa una bandera interna del núcleo[3] para indicarle a éste que ejecute el planificador después de que se haya finalizado la llamada al sistema y se esté por devolver el control al modo usuario[4]. El resto de las interrupciones se comportan de igual modo y nunca provocan de forma directa una replanificación; en su lugar, sólo solicitan que esta se produzca (cuando sea posible) y siempre devuelven el control al mismo código del núcleo que se estaba ejecutando previo a la interrupción.

Dentro (de la mitad superior) del núcleo de Unix los procesos deben renunciar voluntariamente a la CPU. Típicamente esto sucede en dos situaciones: (1) Cuando un proceso ha finalizado sus actividades en modo núcleo y está en camino de retornar al modo usuario (si verifica que la bandera interna solicita una replanificación – según se dijo antes); y (2) cuando requiere esperar por algún recurso o evento externo.

3.1.3 Sincronización de condición dentro del núcleo (y entre mitad superior e inferior)

Es común que un proceso ejecutando el código (de la mitad superior) del núcleo necesite esperar a que finalice algún evento. El ejemplo más obvio es una E/S: cuando un proceso emite una petición de E/S, el manejador de dispositivo correspondiente inicia la operación de transferencia de E/S; sin embargo, puede pasar un largo tiempo antes de que ésta finalice. Cuando se completa la transferencia, el dispositivo emite una interrupción, de modo que es la ISR (dentro del manejador de dispositivo) se entere que la transferencia finalizó y le notifica al proceso. En Unix esta sincronización se lleva a cabo mediante un protocolo de dormirse/despertar mediante las funciones `sleep` y `wakeup`. La mitad superior del manejador invoca a `sleep` cuando quiere esperar por un evento, y la mitad inferior invoca a `wakeup` cuando ocurre el evento según se muestra en la Figura 7. En más detalle:

---

[3] El nombre original de esta bandera en la edición 6 de At&T en SVR3 y BSD 4.3 era **runrun** y en Linux, esta bandera tiene el nombre de **need_resched**.

[4] Mientras el proceso se ejecuta en modo usuario (corriendo el programa de la aplicación) es completamente expropiable. Usualmente, cuando se recibe una interrupción del reloj y el proceso se encuentra en modo usuario (ejecutando el programa de aplicación), la ISR del reloj dentro del núcleo invoca al planificador de forma que potencialmente puede suspender la ejecución el proceso actualmente en ejecución y reanudar la ejecución de otro – el tiempo compartido natural de cualquier sistema Unix.

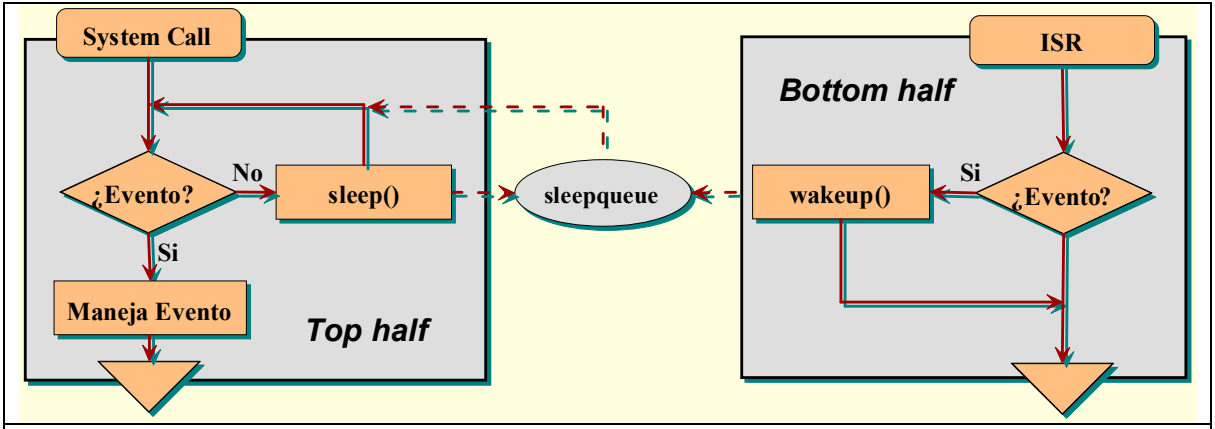


Figura 7 . Sincronización de condición sleep()/wakeUp() dentro del núcleo

- El proceso emite una llamada al sistema (por ejemplo `read()`), la cual lo lleva a modo núcleo.
- El servicio `read()` localiza el manejador de dispositivo asociado el dispositivo de E/S y lo invoca para iniciar la transferencia.
- A continuación `read()` invoca a `sleep`; pasándole la dirección de algún objeto único relacionado con la petición. `sleep` almacena la dirección en una estructura asociada al proceso, marca al proceso como durmiendo y libera al procesador. En este punto el proceso está durmiendo.
- En algún instante posterior, cuando la petición se completa, la ISR dentro del manejador de dispositivo invoca a `wakeup()` con la dirección que le fue pasada a `sleep()`. `Wakeup()` barre la lista de procesos durmiendo y despierta a todos los procesos esperando en esta dirección específica.

3.1.4 Sincronización de exclusión mutua entre la mitad superior y la inferior

El carácter no expropiable del núcleo garantiza que ninguna interrupción pueda afectar el orden de ejecución del código de la mitad superior del núcleo. Sin embargo, la ejecución de un proceso dentro del núcleo puede ser interrumpida temporalmente por (una ISR de) la mitad inferior del núcleo en dependencia del nivel de prioridad de interrupción IPL ("*interrupt priority level*") actual. Cada interrupción de dispositivo tiene asociado un IPL y sólo puede interrumpir al procesador si su IPL es mayor que el IPL actual. Mientras se ejecuta una ISR (código del la mitad inferior), el IPL actual se hace corresponder con el IPL de la interrupción correspondiente.

Para evitar situaciones de carrera durante el acceso a las estructuras de datos compartidas entre los códigos de la mitad superior e inferior; el código de la mitad superior eleva temporalmente el IPL (bloqueando las interrupciones correspondientes) mientras accede a dichas estructuras de datos compartidas. El valor al cual situar el IPL se elige sobre la base del nivel de prioridad del dispositivo que comparte las estructuras de datos que la mitad superior va a modificar. Este mecanismo asegura la consistencia de las colas de trabajos y otras estructuras de datos compartidas entre las mitades superior e inferior.

3.1.5 Inconvenientes de esta arquitectura

Este modelo de sincronización mediante la inhabilitación temporal de las interrupciones posee muchos inconvenientes que afectan el desempeño del sistema:

- Las interrupciones son eventos urgentes e importantes. Mientras estas permanecen inhabilitadas, quedan suspendidas todas las operaciones de E/S (por interrupción) y cualquier dispositivo que esté solicitando servicio tendrá que esperar hasta que las interrupciones se habiliten de nuevo (que puede ser un tiempo potencialmente largo). Esto puede traer consigo pérdida de datos por desbordamiento ("*overrum*") si se pierden interrupciones[5].

- En la mayoría de las arquitecturas de hardware, la habilitación e inhabilitación de las interrupciones o la elevación o disminución del IPL es una operación costosa que requiere varias instrucciones de máquina. Como consecuencia, existe una penalización por la inhabilitación de las interrupciones.

- En sistemas multiprocesadores, estos problemas se magnifican. El núcleo tiene que proteger mucho más objetos y generalmente tiene que bloquear las interrupciones en todos los procesadores.

3.2 Tratamiento de las Interrupciones en los Sistemas Operativos de Red

La solución adoptada por los Sistemas Operativos de Red (VMS 1.0+[43], BSD 4.2+[64], NT [85] y Linux [11]) consiste en dividir la atención a las interrupciones en dos niveles que puedan operar a diferentes IPLs:

- El primer nivel consiste en el manejo convencional de la interrupción (ISR) que se ejecuta al nivel IPL (alto) de la interrupción correspondiente. Este manejador es responsable de interactuar directamente con el dispositivo de hardware y de la administración de buffers; o sea, las funciones que tienen que llevarse a cabo antes de que sea posible volver a habilitar las interrupciones provenientes del dispositivo (disminuir el nivel IPL).

- El segundo nivel utiliza algún mecanismo de **interrupción de software** solicitada desde el manejador de primer nivel (que sitúa en cola la petición) y que permite postergar el procesamiento de más baja prioridad para que sea ejecutado por un pequeño planificador que se invoca de forma automática cuando terminan de ejecutarse las ISRs del primer nivel y baja el IPL. Estos manejadores se ejecutan entonces con todas las interrupciones de hardware habilitadas. Si se produce alguna otra interrupción de hardware mientras se está ejecutando el código de esta interrupción de software, esta última será interrumpida igual que ocurre con las demás tareas de baja prioridad. Entonces la ISR de primer nivel puede solicitar que se ejecute otra (o incluso la misma)

[5] La práctica común para "solucionar" este problema es a través de múltiples iteraciones de pruebas y errores. En sistemas comerciales es común que se emitan múltiples versiones con errores antes de atinar con la versión "correcta".

interrupción de software, pero su ejecución se diferirá hasta tanto la que se está ejecutando actualmente finalice. Cada ejecución de una interrupción de software de segundo nivel puede ser interrumpida por una ISR de primer nivel, pero nunca puede ser interrumpida por otra interrupción de software de segundo nivel similar. En otras palabras, los manejadores de este segundo nivel nunca pueden ser interrumpidos a si mismos (se ejecutan hasta terminar).

Esta estructura de dos niveles tiene como propósito reducir significativamente la cantidad de tiempo durante el cual están inhabilitadas las interrupciones de hardware (nivel IPL alto) y ello se logra por varios factores:

- El código de la parte del núcleo no dirigido por interrupción (mitad superior) sólo necesita inhabilitar las interrupciones en los casos en que tenga que acceder a estructuras de datos compartidas con las ISRs. Si se restringen los servicios que pueden ser invocados desde las ISRs de primer nivel a unos pocos que no acceden a datos compartidos con la parte del núcleo no dirigida por interrupción (por ejemplo, sólo a la petición de la interrupción de software de segundo nivel), entonces la parte del núcleo no dirigida por interrupción no necesitará inhabilitar las interrupciones de hardware.

- Aun en los casos en que el núcleo nunca inhabilitara las interrupciones para acceder a las secciones críticas, ello no significa que estas estén siempre habilitadas. Cada vez que se recibe una interrupción, el procesador hace una inhabilitación automática de las interrupciones al elevar el IPL hasta el nivel de la interrupción correspondiente para transferir el control a la ISR. Al permitir posponer el grueso del procesamiento de la interrupción a un manejador de segundo nivel con IPL bajo (interrupciones de hardware habilitadas) se logra mantener en un mínimo el tiempo de inhabilitación automática de las interrupciones.

Esta arquitectura de dos niveles de procesamiento de interrupción se completa con un mecanismo que permite inhabilitar (selectivamente) las interrupciones de software de segundo nivel mientras se ejecuta el código normal de la parte no dirigida por interrupción del núcleo. El núcleo puede proteger fácilmente sus secciones críticas con los manejadores de segundo nivel utilizando el siguiente sistema: los manejadores de interrupción siempre posponen la ejecución de cualquier código que pudiera implicar el acceso a datos compartidos con la parte no dirigida por interrupción del núcleo para que sea ejecutado en los manejadores de segundo nivel. Cuando la parte del código del núcleo no dirigida por interrupción desea entrar en una sección crítica compartida con (los manejadores de segundo nivel de) la parte dirigida por interrupción, puede inhabilitar a las interrupciones de software relevantes para evitar que estas los interrumpan. Al final de la sección crítica, el núcleo puede rehabilitar nuevamente las interrupciones de software y ejecutar cualquier manejador pendiente que haya sido situada en cola durante la sección crítica por las ISRs de primer nivel.

La Figura 8 resume los diferentes niveles de protección de interrupciones en el sistema. Cada nivel puede ser interrumpido por el código ejecutándose a un nivel superior, pero nunca será interrumpido por código ejecutándose al mismo nivel o a un nivel inferior

(excepto para el código de modo usuario que siempre será expropiado por otros procesos cuando ocurra una interrupción de planificación de tiempo compartido).

Niveles de Prioridad (IPL)	Funcionalidad del código
Alta	Manejadores de Interrupción de 1er Nivel (ISRs)
	Manejadores de Interrupción de 2do Nivel. Se ejecutan hasta terminar con todas las interrupciones habilitadas.
	Parte del núcleo no dirigido por interrupción (mitad superior). Comprende el planificador y las rutinas de servicio del núcleo. Este código es no expropiable.
Baja	Programas en modo usuarios. Siempre son expropiables

Figura 8. Niveles de protección de Interrupción

Aunque el propósito y el principio general de esta arquitectura de interrupciones de dos niveles es el mismo en todos los sistemas, los detalles de implementación y la denominación del manejador difieren en los distintos sistemas. En algunas arquitecturas (tales como la VAX [43]), estas interrupciones de software se implementan utilizando verdaderas trampas de hardware provocadas por instrucciones de máquina de la CPU. En otras arquitecturas, la misma funcionalidad se implementa completamente en software, supervisando banderas establecidas por el manejador de interrupción en los instantes de tiempo adecuados e invocando directamente a las funciones de procesamiento solicitadas (Llamadas a Procedimientos Diferidos o DPC – "*Deferred Procedure Call*" – en NT [85], Mitad inferior – "*bottom half*" – en Linux[11], Procedimiento de Bifurcación – "*Fork Procedure*" en VMS [43]). A continuación se da una panorámica de su implementación en NT y en Linux.

3.2.1 Manejo de Interrupciones en Windows NT

NT define un conjunto priorizado de Niveles de Petición de Interrupción IRQL ("*Interrupt Request level*") independiente de la arquitectura. Según se muestra en la Figura 9, estos niveles se hacen corresponder con los niveles de petición de interrupción de hardware; así como, con niveles de interrupciones de software predefinidos.

Windows NT posee una arquitectura interna muy diferente de la arquitectura monolítica[6] de los núcleos de Unix tradicionales. NT posee una arquitectura alternativa que combina la arquitectura clásica de micro-núcleo ("*micro-kernel*") (introducida por sistemas tales como Mach [3] y Chorus [5]) con la arquitectura estratificada (introducida por el sistema THE [30]). El equivalente al núcleo de Unix es lo que en NT recibe el nombre de Ejecutivo. Lo que se denomina núcleo en NT es sólo una parte muy pequeña del ejecutivo que sólo implementa un conjunto de mecanismos básicos.

Hasta la versión NT 4.0 la documentación oficial de Intel menciona que el núcleo de NT es no expropiable [85]; ello quiere decir que, aunque el código del núcleo puede ser

[6] La arquitectura monolítica se refiere a grandes rasgos a que todo el sistema operativo está contenido dentro de un gran núcleo que incluye todas las funciones del sistema operativo.

suspendido temporalmente por una interrupción para la ejecución de una ISR esta siempre devolverá el control al núcleo sin expropiar al hilo actualmente en ejecución (los hilos dentro del núcleo no son expropiados ni siquiera cuando se le vence su cuanto de tiempo). Cuando un hilo que se ejecuta dentro del núcleo (con un nivel IPL 0) necesita esperar por algún evento, este cede voluntariamente la CPU invocando directamente al planificador para realizar un cambio de contexto de forma inmediata.

	Nombre simbólico	Propósito	Nivel Intel	Nivel Alpha
hardware	Nivel Alto	Mayor Nivel de Interrupción	31	7
	Nivel de Energía	Falla de Suministro Eléctrico	30	7
	Nivel IPI	Señal Inter-Procesador	29	6
	Nivel del Reloj	Pulso de Reloj	28	5
	Nivel de Perfil	Monitoreo de Desempeño (historial)	27	3
	Nivel de Dispositivo	Interrupciones generales de los Dispositivos	3-26	3-4
Software	Nivel de Despacho	Operaciones del Planificador y Llamadas a Procedimientos Diferidos (DPCs)	2	2
	Nivel APC	Llamadas a Procedimientos Asícronos	1	1
	Nivel Pasivo	Todos los niveles de interrupción habilitados	0	0

Figura 9. Niveles de Interrupción independientes de la arquitectura de Windows NT

Si el núcleo detecta la necesidad de realizar una replanificación mientras se encuentra a un nivel de interrupción elevando (por ejemplo como parte de la ejecución de una Llamada a Procedimiento Diferida – ver Figura 9) entonces no invoca directamente al planificador sino que solicita una interrupción de nivel despacho/DPC para activar el planificador. Como el IRQL está en ese nivel, o en uno superior, el procesador pone la interrupción en espera. Cuando el núcleo termina su actividad actual, reduce el IRQL por debajo del nivel despacho/DPC y se activa la interrupción DPC que causa la replanificación.

Aunque los servicios de bajo nivel del núcleo no son expropiables estos son simples y se ejecutan muy rápidamente de manera que, el hecho de que el núcleo sea no expropiable ejerce un mínimo impacto en la latencia de expropiación. Adicionalmente, al no ser el núcleo expropiable se evita el problema de la inversión de prioridad no acotada (situación en la cual una tarea de alta prioridad ve impedida su ejecución debido a la ejecución de una tarea de menor prioridad y esto se mantiene por un tiempo no predecible). En NT, el grueso del código del sistema reside en el ejecutivo, quien implementa las políticas y los servicios más sofisticados en función de los servicios del núcleo. El ejecutivo se implementa mediante múltiples hilos y es completamente expropiable. Este diseño minimiza la latencia de replanificación a la vez que evita el problema de la inversión de prioridades no acotada.

Cuando el núcleo necesita sincronizar el acceso a estructuras internas compartidas que no son accedidas por las ISRs, no necesita inhabilitar las interrupciones de hardware; sino que, simplemente eleva el IRQL del procesador al nivel de despacho/DPC. Esto inhabilita todas las demás interrupciones de software incluyendo la replanificación de hilos. Sólo en el caso de que se necesite proteger el acceso a datos compartidos entre el núcleo y las ISRs, es que

el núcleo de NT eleva el IRQL del procesador hasta un nivel más alto inhabilitando cualquier fuente de interrupción cuya ISR pudiera acceder a los datos.

Esta estrategia de sincronización funciona en sistemas con un sólo procesador, pero resulta inadecuada para una configuración multiprocesador. Elevar el IRQL en un procesador no evita que ocurra una interrupción en otro procesador. El núcleo necesita garantizar también acceso mutuamente exclusivo a través de los diversos procesadores[7].

Para lograr exclusión mutua en un sistema multiprocesador, NT utiliza el mecanismo de **cierre de giro** ("*spin lock*") asociado a las estructuras de datos globales (compartidas entre varias CPUs). Mientras el cierre no está disponible la CPU se mantiene en un ciclo intentando continuamente (en espera ocupada) hasta tanto esté disponible.

Mientras un hilo retiene un cierre de giro, existe el peligro de que algún otro código activado por una interrupción en la misma CPU intente obtener el mismo cierre de giro. En esta situación el código activado por la interrupción permanecería girando indefinidamente debido a que precisamente él está impidiendo el progreso del código que pudiera liberar el cierre. Para impedir una posibilidad de bloqueo mutuo como esta, los cierres de giros tienen que primero elevar el nivel de prioridad de interrupción hasta un nivel lo suficientemente alto y mantenerlo en ese nivel mientras se retiene el cierre (se ejecuta la sección crítica). Una vez que se libere el cierre, el nivel de interrupción debe disminuirse a su valor previo.

En NT, todos los cierres de giro de modo núcleo poseen un IRQL asociado que siempre se sitúa a nivel de Despacho o superior. Por lo cual, cuando un hilo está intentando adquirir un cierre de giro, todas las demás actividades al nivel IRQ del cierre o a niveles inferiores cesan en ese procesador. Como la conmutación de hilos sucede al nivel de Despacho, un hilo que sostiene un cierre de giro nunca es expropiado (el IRQL enmascara al mecanismo de despacho). Esto permite que los hilos que retienen cierres de giro continúen ejecutándose para que el cierre se libere rápidamente. El núcleo usa los cierres de giro con mucho cuidado minimizando el número de instrucciones que ejecuta mientras retiene alguno[8].

NT implementa la arquitectura estándar de manejo de interrupciones en dos niveles, típica de los sistemas operativos de red. Primero, la interrupción es manejada por una ISR muy corta. Luego el trabajo se completa mediante la ejecución diferida de un Llamada a Procedimiento Diferido o DPC ("*Deferred Procedure Call*"). Las ISR pueden ser interrumpidas por ISR de mayor prioridad. Por su parte, todas las DPC se ejecutan, al nivel

[7] A partir de Windows 2000 (NT 5.0) la documentación oficial de Microsoft ha eliminado la alusión de que el núcleo de NT es no expropiable [86][77]. Esto hace suponer que a partir de estas versiones si el núcleo no está ejecutando código dentro de una sección crítica protegida por cierres de giros ("*spinlocks*") puede ser expropiado. De cualquier modo, en un sistema SMP el hecho de que el núcleo no sea expropiable no garantiza protección implícita de las regiones críticas contra secciones del núcleo que pudieran estar corriendo en otra CPU.

[8] Debido a que el IRQL es un mecanismo de sincronización efectiva en sistema uniprocesadoes, las funciones de adquisición y liberación de los cierres de giro en el HAL uniprocesador en realidad no los implementan sino que simplemente elevan o disminuyen el IRQL.

de prioridad de despacho el cual se encuentra por encima del nivel de prioridad de los hilos de usuario y del sistema (ver Figura 9). Esto significa que se ejecutan con todas las interrupciones de hardware habilitadas y con la planificación y los demás DPCs inhabilitados (todos están al mismo nivel de prioridad). Los DPC se sitúan en cola en este nivel y se planifican según una disciplina FIFO ("*First In - First Out*").

3.2.2 Manejo de Interrupciones en Linux

Al igual que los sistemas Unix tradicionales, las versiones del núcleo de Linux (previas a la versión 2.6) son no expropiables y también se pueden dividir en dos mitades: una **mitad no dirigida por interrupción** ("*non-interrupt half*") que es activada de forma procedural por las llamadas al sistema (que se corresponde con el "*top half*" de BCD) y la otra **mitad dirigida por interrupción** ("*interrupt half*") que contiene el código que se ejecuta como parte de las peticiones de interrupción (y se corresponde con el "*bottom half*" de BSD). Igual que en los sistemas Unix clásicos, ninguna interrupción que se reciba mientras un proceso (o hilo) está ejecutando el código de un servicio del núcleo, provoca una replanificación de forma directa; en su lugar, se activa la bandera del núcleo **need_resched** para solicitarle al núcleo que ejecute el planificador luego de que se haya completado la llamada al sistema y se esté por devolver el control al modo usuario. Los procesos (o hilos) utilizan el mismo mecanismo de sincronización con eventos basado en el esquema de dormirse/despertarse. La sincronización entre el código de la mitad no dirigida por interrupción y el código de las ISRs se realiza de igual modo mediante la inhabilitación temporal de las interrupciones durante el acceso a las estructuras de datos compartidas.

Al igual que otros sistemas operativos de red, Linux implementa una arquitectura estándar de manejo de interrupciones en dos niveles dividiendo el servicio a las interrupciones en dos secciones: la **mitad superior** ("*Top half*") constituida por la ISR que recibe la interrupción de hardware y la **mitad inferior** ("*Bottom half*") que hace el grueso del procesamiento de la petición de forma diferida con todas las interrupciones habilitadas [101].

La arquitectura de *bottom half* original se mantuvo sin modificaciones hasta la versión Linux 2.2. Sin embargo, debido a que el diseño original de Linux se hizo para máquinas con una sola CPU, esta arquitectura de *bottom half* se convirtió en un cuello de botella en arquitecturas con múltiples CPU. El problema era que aunque cada una de las CPU podía manejar una interrupción ("*top half*") a la vez, la capa de *bottom half* era de simple hilo, de modo que el procesamiento diferido por todas las ISRs no se podía distribuir entre todas las CPUs. En consecuencia, para la versión 2.3 se introdujo el soporte de multiprocesamiento simétrico o SMP ("*Symmetric Multiprocessors*") en los *bottom halves*. Esto se llevó a cabo reemplazando los *bottom halves* originales con los denominados "*softirq*" y "*tasklets*".

Una *softirq* representa una petición para que una función específica se ejecute en algún instante futuro. Si el mismo tipo de *softirq* se solicita múltiples veces entonces las invocaciones de esta se pueden ejecutar de forma concurrente en múltiples procesadores. Por el contrario, diferentes *tasklets* pueden ejecutarse simultáneamente en múltiples CPUs, pero las invocaciones de la misma *tasklet* son serializadas con respecto a si mismas. Por razones de compatibilidad, los *bottom halves* del viejo estilo se volvieron a implementar

utilizando un conjunto de *tasklets* que se ejecutaban reteniendo un cierre de giro ("*spinlock*") global dedicado de modo que cuando uno se está ejecutando en alguna CPU, ningún otro se puede ejecutar en alguna otra CPU.

Aunque el diseño anterior preservó la compatibilidad con los manejadores de dispositivos legados, todavía le imponía una fuerte restricción al desempeño de Linux 2.4 en sistemas multiprocesador. Para la versión Linux 2.5 los bottom halves del viejo estilo fueron eliminados y todo el código que lo usaba se modificó para usar ya sea *softirqs* o *tasklets*. Actualmente el término "Bottom Half" se usa para referirse a cualquiera de los código diferibles (sea *softirq* o un *tasklet*).

En Linux 2.6 se introdujo otro esquema para planificación de funciones diferidas al que se le denomina colas de trabajo ("*workqueues*") y que como diferencias más importantes con las funciones diferibles antes mencionadas se ejecutan en el contexto de hilos del núcleo. La Figura 10 muestra un resumen de estos mecanismos de ejecución diferida de Linux.

Bottom Half	Estado	Comentario
BH	Eliminado en 2.5	Su ejecución está globalmente serializada.
Task queues	Eliminado en 2.5	
Softirq	Disponible desde 2.3	Pueden ejecutarse en varias CPUs a la vez, incluso si son del mismo tipo.
Tasklet	Disponible desde 2.3	Las de diferentes tipos se pueden ejecutar concurrentemente, pero las del mismo tipo no.
Work queues	Disponible desde 2.5	

Figura 10. Niveles de protección de Interrupción

3.2.3 Problema del Encierro de Recepción ("Receive Livelock") y soluciones propuestas

Un problema potencial con la arquitectura de interrupciones en dos niveles es que, como las interrupciones del dispositivo se habilitan antes de finalizar su procesamiento (con el procesamiento diferido pendiente) no tienen protección contra la posibilidad de que el dispositivo genere peticiones de interrupción a una tasa sostenida superior a la capacidad de procesamiento. Esta situación provoca que, bajo cargas de trabajo extremadamente elevadas, pueda presentarse lo que se conoce como **encierro de recepción** ("*receive livelock*") [66] o condición en la cual el sistema de cómputo se ve abrumado ("*overwhelmed*") por la llegada de interrupciones y durante períodos de tiempos extremadamente largos gasta la mayor parte o todo su tiempo procesando las interrupciones (dejando en inanición los hilos de las aplicaciones). En casos extremos, el procesamiento diferido al IPL bajo nunca termina antes de que se reciba la siguiente interrupción, entregando más datos y requiriendo más procesamiento al IPL bajo. El empleo de buffers puede absorber pequeñas ráfagas de datos recibidos, pero en última instancia el espacio se puede agotar y los datos tienen que ser desechados. Si esta situación continua entonces no se hará trabajo útil porque el sistema gasta todo su tiempo recibiendo datos sólo para desecharlos más tarde.

Se han propuestos varias técnicas para enfrentar este problema. Todas ellas se apoyan en reducir la tasa a la cual pueden ocurrir las interrupciones con el propósito de evitar la llegada del encierro de recepción; si el sistema está sometido a una sobrecarga ("*overloaded*") entonces tiene sentido desechar los paquetes no procesados en el dispositivo/interfaz antes de que se invierta esfuerzo en el procesamiento parcial. Un enfoque simplista es inhabilitar las interrupciones provenientes de un dispositivo específico cuando los datos provenientes del mismo ocupan más de algún umbral en los buffers internos. Un segundo enfoque, propuesto en [66] por Mogul y Ramakrishnan, consiste en reducir el tamaño del manejador de interrupción a un mínimo absoluto, sólo una notificación de que ha sucedido un evento, y utilizar este para disparar la encuesta ("*polling*"). Tales interrupciones son idempotentes y pueden ser enmascaradas desde el instante entre la entrega de la primera desde el dispositivo y encuestar el estado del dispositivo.

3.2.4 Dificultades para aplicaciones de tiempo real y modificaciones propuestas

La arquitectura de procesamiento de interrupciones en dos niveles propia de los sistemas operativos de red (VMS, BSD 4.2+, NT o Linux) consigue descargar el procesamiento de interrupción de los manejadores de interrupción hacia una actividad planificada de forma independiente (AST en VMS, interrupción de software en BSD 4.2+, DPC en NT y mitad inferior – "*bottom half*" – en Linux). Al acortar las ISRs, el sistema es capaz de responder a las interrupciones subsecuentes más pronto, reduciendo la latencia de interrupción en el peor caso. Sin embargo, todas estas variantes de manejadores de interrupción diferidos se planifican antes que todas las tareas de usuario. Por tanto, todo el procesamiento de interrupciones en estos sistemas se ejecuta efectivamente a una mayor prioridad que todas las tareas de usuario, incluyendo los procesos de tiempo real. Para las aplicaciones de usuario críticas en tiempo que están impedidas de ejecutarse, hay poco beneficio en el hecho de que el trabajo se esté haciendo en manejadores diferidos en vez de en las ISRs propiamente. Varios investigadores ha propuestos distintas adecuaciones de esta arquitectura para enfrentar mejor los requerimientos de aplicaciones sensibles al tiempo (por ejemplo [40] y [105]).

Uno de los problemas de este diseño es el denominado tiempo robado ("*stolen-time*") a las aplicaciones por el procesamiento de interrupciones, principalmente por los manejadores de procesamiento diferido. Este tiempo está completamente fuera del control del planificador del sistema y fluctúa grandemente de modo que introduce una perturbación al planificador de procesos (o hilos) que afecta significativamente su capacidad para hacer cumplir la política de distribución del tiempo de la CPU entre las aplicaciones de usuario.

En [40] se propuso e implementó un esquema de planificación de los manejadores diferidos ("*botton half*") de Linux 2.4. Este consiste en supervisar el tiempo consumido por los manejadores diferidos y detener las subsiguientes ejecuciones de los mismos una vez que este tiempo ha sobrepasado un valor de umbral. Los manejadores pendientes son pospuestos hasta el siguiente instante en que el núcleo proceda a la ejecución de los mismos. De esta forma, el exceso de carga de trabajo por encima del tiempo de umbral establecido se procesa cuando las peticiones de manejadores diferidos están por debajo del tiempo de umbral. Un aspecto interesante de este esquema es que el valor de umbral no es

fijo sino que se obtiene dinámicamente usando una estimación basada en el tiempo previamente consumido por el procesamiento diferido. Esta característica de adaptación del umbral a la carga de E/S consigue evitar variaciones bruscas del tiempo consumido (robado) por los manejadores de procesamiento diferido logrando estabilizar el tiempo de ejecución de los procesos de usuario; todo ello sin introducir una demora en el servicio a los paquetes de red. Además, el método propuesto no necesita modificar el mecanismo de planificación o la arquitectura del subsistema de red.

Más recientemente en [105] Zhang y West propusieron e implementaron una modificación a la arquitectura de Linux 2.6 que añade un planificador de los manejadores de interrupción diferibles (conocidos como *bottom halves*, *softirqs* y *tasklet* – ver sección 3.2.2) luego de la ejecución de la ISR de primer nivel. Este planificador hace una preedición de la prioridad del proceso que hizo la solicitud (y que fué beneficiado por la interrupción) para determinar de forma más precisa cuando planificar la ejecución del manejador diferido. Además, luego de que se ejecuta el manejador diferido se añade un componente que contabiliza el tiempo de ejecución de los manejadores de interrupción y se los carga al proceso apropiado.

3.3 Manejo de Interrupciones como Hilos

Una alternativa al esquema de manejo de interrupciones en dos niveles discutido en la sección 3.2 es el tratamiento de las interrupciones dentro de actividades concurrentes asíncronas (hilos, tareas o procesos) dedicadas. La idea general de este esquema consiste en que el núcleo del sistema operativo se encarga de situar un manejador de interrupción de bajo nivel genérico el cual hace lo necesario para activar a una actividad asíncrona (hilo, tarea o proceso) que se ocupa de dar el tratamiento específico de la interrupción. Aunque la idea general es la misma, la motivación y los detalles de implementación permiten identificar dos variaciones: Señales de interrupciones como eventos de comunicación entre procesos o IPC ("*Inter-process comunication*") propia de los sistemas con arquitectura de micro-núcleo; e Interrupciones como hilos del núcleo. A su vez, en este último se presentan dos variaciones ejemplificadas por el modelo de interrupciones como hilos de Solaris 2.0 (ver sección 3.3.2) y el modelo de interrupciones como hilos de los sistemas de tiempo real (LynxOS o Linux de Tiempo Real – ver sección 3.3.3)

3.3.1 Interrupciones como IPC (arquitectura de micronúcleo)

Un **micro-núcleo** ("*micro-kernel*") es un pequeño núcleo de un sistema operativo que implementa las funciones fundamentales que sirven de base para extensiones modulares y transportables que implementan los servicios menos esenciales del sistema. Típicamente el micro-núcleo suministra la abstracción de tareas o procesos planificables y el mecanismo de comunicación entre éstos basado en mensajes. Servicios como son el sistema de archivos, sistema de ventanas, servicios de seguridad y otros se implementan como componentes encima del micro-núcleo y se comunican unos con los otros mediante el empleo de los servicios IPC que suministra el micro-núcleo. Es interesante que los conceptos que subyacen en la arquitectura de micro-núcleo (y la señalización de interrupciones a través de IPC) fueron introducidos por vez primera en 1969 con el sistema multiprogramado del RC 4000 [13][14] mucho antes de la introducción del término micro-

núcleo como tal. Este sistema fue diseñado por Brinch-Hansen para la computadora RC 4000 fabricada en Dinamarca por Regnecentralen..

El núcleo del sistema de la RC 4000 suministró los mecanismos básicos para crear un árbol de procesos concurrentes que (aunque podían compartir memoria) interactuaban entre si mediante un mecanismo de comunicación basado en mensajes según un protocolo de petición y respuesta entre dos procesos. Según el propio Brinch Hansen, la elección de este esquema estuvo condicionada por una decisión temprana de tratar a los dispositivos de E/S como procesos, que reciben comandos de E/S como mensajes y devuelven acuses de recibos o respuestas [15]. Los manejadores de dispositivos se codificaban de forma tal que convertían las interrupciones del dispositivo y los registros en mensajes. Así, un proceso escribiría a una Terminal enviándole a dicha Terminal un mensaje. El manejador de dispositivo recibiría el mensaje y sacaría el carácter a la Terminal. Un carácter de entrada interrumpiría al sistema y transferiría el control al manejador de dispositivo. El manejador de dispositivo crearía un mensaje a partir del carácter de entrada y lo enviaría al que está esperando por él.

El sistema Mach [3], se basó en los mismos principios, acuñando el término micro-núcleo ("*microkernel*"). Otro micro-núcleo contemporáneo con Mach fue Chorus [75], ambos fueron críticos para la evaluación en el "mundo real" y la investigación del diseño de micro-núcleo. Mach, Chorus y otros muchos seguidores de finales de la década de 1980 son exponentes de lo que se conoció como primera generación de micro-núcleos.

3.3.1.1 Manejo de interrupciones a nivel de usuario

A tono con la filosofía de micro-núcleo es la implementación de todos los manejadores de dispositivo ("*device drivers*") como servidores a nivel de usuario fuera del núcleo. Esto tiene la ventaja de que los manejadores de dispositivos pueden ser reemplazados, eliminados, o añadidos dinámicamente – sin enlazar un nuevo núcleo e iniciar el sistema. Así los manejadores pueden distribuirse a los usuarios finales independientemente del núcleo. Motivados por esto, los micro-núcleos se ven en la necesidad de suministrar mecanismos que les permitan a los manejadores de dispositivo el acceso a los dispositivos y la implementación de manejadores de interrupciones a nivel de usuario.

La idea más natural de permitir manejadores de interrupción a nivel de usuario en un micro-núcleo es interpretando las interrupciones de hardware como mensajes de comunicación entre procesos o IPC ("*inter-process communication*") [57]. El micro-núcleo captura todas las interrupciones pero no se involucra en el manejo específico del dispositivo (no necesita saber nada de la semántica de la interrupción ni la política de manejo), en su lugar sólo genera un mensaje para el proceso o hilo de nivel de usuario asociado con la interrupción (mecanismo). Así, consistente con el principio de separación de mecanismo y política, el manejo específico de las interrupciones y de la E/S a los dispositivos se hace completamente fuera del núcleo en el contexto de una tarea hilo o proceso de la forma ilustrada en la Figura 11.

La transformación de las interrupciones en mensajes destinados al hilo o proceso asociado, tiene la ventaja adicional de que los manejadores de dispositivo se benefician del empleo de

los mecanismos suministrados por el micro-núcleo tales como hilos, espacios de direcciones y fundamentalmente los mecanismos de comunicación y sincronización entre hilos. Como consecuencia la sincronización de interrupciones se resuelve usando la sincronización de hilos ordinaria sin necesidad de algún mecanismo especial.

```
Driver thread:
     do
          wait for (msg, sender) ;
          if sender = mi interrupción de hardware
               then read/write i/o ports ;
                    limpia la interrupción de hardware
               else . . .

     while (TRUE);
```

Figura 11. Interrupciones como IPC

3.3.2 Modelo de manejo de interrupciones como Hilos del Núcleo en Solaris 2.0.

Sun Microsystems introdujo soporte de hilos en el núcleo en su sistema operativo Solaris 2.x (SunOS 5.0) [31]. En Solaris un hilo de núcleo es la entidad fundamental que es planificada y despachada en las CPUs del sistema. Los hilos de núcleo son muy ligeros, sólo poseen una pequeña estructura de datos y una pila. La conmutación entre hilos de núcleo es muy ligera ya que no requiere el cambio de espacio de direcciones de memoria virtual. Los hilos de núcleo (y el núcleo de Solaris) son completamente expropiables (los detalles de la arquitectura de Solaris pueden verse en [63]).

Solaris reemplaza el modelo tradicional de interrupción y sincronización con un esquema en el que las interrupciones son manejadas como hilos del núcleo. Estos hilos de interrupción pueden crearse dinámicamente y se les asigna una mayor prioridad que a todos los demás tipos de hilos. Ellos usan las mismas primitivas de sincronización que los demás hilos y por tanto se pueden bloquear si necesitan un recurso retenido por otro hilo. En consecuencia, el núcleo no necesita utilizar el IPL del procesador para protegerse de las interrupciones. El núcleo bloquea las interrupciones sólo en unas pocas situaciones excepcionales [45].

Aunque la creación de hilos del núcleo es relativamente ligera, todavía es demasiado costoso crear un nuevo hilo para cada interrupción. El núcleo mantiene un banco de hilos de interrupción, preasignados y parcialmente inicializados. Por omisión, este banco contiene un hilo por nivel de interrupción por cada CPU, más un único hilo en todo el sistema para el reloj.

El mecanismo es el siguiente: cuando llega la interrupción, un manejador de interrupción dentro del núcleo eleva el IPL para evitar interrupciones adicionales del mismo nivel o de nivel inferior. Luego asigna un hilo de interrupción desde el banco y conmuta el contexto al mismo. Mientras se ejecuta el hilo de interrupción, el hilo interrumpido está **clavado** ("*pinned*"), lo que significa que no puede ejecutarse en otra CPU. Cuando el hilo de

interrupción devuelve el control, se conmuta el contexto nuevamente al hilo interrumpido, el cuál reanuda la ejecución.

Con el propósito de disminuir la sobrecarga de la conmutación de contexto hacia el hilo de interrupción, este se ejecuta sin ser inicializado completamente. Esto significa que no es un hilo "con todas las de la ley" y no puede ser desplanificado. La inicialización se completa sólo si el hilo se bloquea. En este instante, se desclava el hilo interrumpido y se salva el estado completo del hilo de interrupción convirtiéndose en un hilo capaz de ser planificado como cualquier otro. Posteriormente, se devuelve el control al hilo previamente interrumpido. De esta forma, la sobrecarga de una inicialización completa del hilo se restringe a los casos en los que el hilo de interrupción se debe bloquear.

La implementación de las interrupciones como hilos añade un costo operativo al manejo de las interrupciones. Sin embargo, se evita la necesidad de manipular el IPL del procesador para bloquear las interrupciones en cada operación sobre un objeto de sincronización (y dejarlas bloqueadas mientras se está dentro de una sección crítica del núcleo)[9]. Debido a que las operaciones de sincronización son mucho más frecuentes que las interrupciones, el resultado es una mejora del desempeño, siempre y cuando las interrupciones no se bloqueen demasiado frecuentemente. En la práctica, menos del 0.5% de las interrupciones se bloquean. El trabajo de convertir una interrupción en un hilo "real" se realiza sólo cuando existe contención por cierre. El compromiso aquí es que el hilo interrumpido queda clavado imposibilitando su ejecución (incluso en otra CPU) hasta que el manejador de interrupción termine o se bloquee lo cual puede introducir una inversión de prioridad temporal. Sin embargo, este esquema ayuda a mejorar el desempeño y reducir la latencia de interrupción, particularmente en el caso de multiprocesadores.

Es importante destacar que la optimización de despachar al hilo de interrupción de forma inmediata (sin pasar por la trayectoria normal del planificador) es posible en Solaris por dos razones:

(1) los hilos de interrupción poseen una prioridad mayor que el resto de los hilos en el sistema la cual además es la que se corresponde con la prioridad de la interrupción de hardware. Esto garantiza que siempre que se recibe una interrupción no puede haber otro hilo (no asociado a interrupción) de mayor prioridad listo para ser ejecutado.

(2) porque su núcleo es completamente expropiable; o sea, que permite que ocurra una conmutación de hilo en cualquier momento. Este hecho es lo que le permite despachar la ejecución del hilo de interrupción en el mismo instante en que se produce la interrupción (sin necesidad de tener que esperar a que el hilo actual abandone la ejecución de alguna sección de código dentro del núcleo).

[9] En [45] se documenta que la carga operativa adicional de recibir una interrupción (respecto al método tradicional) es de unas 40 instrucciones SPARC. El ahorro en la ruta cierre/apertura del mutex es de unas 12 instrucciones las cuales elevan y disminuyen la prioridad de interrupción del procesador. Debe considerarse además que el SPARC tiene integrado dentro de la CPU la lógica para controlar el nivel de interrupción del procesador integrado. Los procesadores que usan controladores de interrupción externos (como el Pentium) pudieran probablemente incurrir en un gasto mayor.

Obsérvese que en un sistema operativo no expropiable la replanificación sólo puede tener lugar en momentos específicos en que el núcleo está en un estado consistente. En este caso si se transfiriere el control de forma inmediata al manejador de interrupción (como sucede en Solaris) entonces dicho manejador no se podría bloquear ya que la interrupción se pudo haber producido en cualquier instante, incluso en momentos en que el núcleo no se encuentra en un estado consistente. En esta situación el bloqueo provocará que se entre al núcleo y este haga una replanificación en un momento inoportuno provocando el colapso del sistema. En consecuencia, en sistemas con núcleos no expropiables, los hilos de interrupción tienen que esperar porque el hilo interrumpido termine su trayectoria de ejecución dentro del núcleo, por lo que se afecta significativamente la latencia de interrupción.

3.3.3 Interrupciones manejadas como hilos en los sistemas Linux para Tiempo Real

Motivados por la necesidad de hacer que el núcleo de Linux fuese más sensible ("*responsive*") a los eventos externos de modo que fuese adecuado para aplicaciones con requerimientos de tiempo, muchos trabajos le han realizado modificaciones para introducirle el tratamiento de interrupciones (exceptuando la ISR del Reloj) en el contexto de hilos del núcleo (Manas Saksena – TimeSys. – [79], Steven-Thorsten Dietrich et. al. – Montavista – [29], Heursch et. al [36] y Yang et. al. [103]). Estas modificaciones han tenido el propósito de reducir la latencia de expropiación, la cual puede ser muy elevada en Linux (superior a los 100 ms [35][102]).

Estos trabajos estuvieron precedidos por los primeros enfoques para introducir la expropiación al núcleo de Linux: los parches de expropiación [5][96] y los parches de baja latencia [104]. Las primeras implementaciones de estos parches protegían las secciones críticas dentro del núcleo mediante cierres de expropiación ("*preemption locks*") que inhabilitaban la expropiación durante las mismas [5]. El siguiente paso fue sustituir los cierres de expropiación por mutexes de modo que la expropiación fuese posible incluso mientras el núcleo está dentro de una sección crítica [96]. Estas técnicas lograron reducir significativamente la latencia de expropiación con respecto al núcleo de Linux convencional; sin embargo todavía no logran obtener valores suficientemente bajos (ver [26][102][2]). La razón de ello es que con estas técnicas no es posible la expropiación mientras se está ejecutando una ISR o incluso los manejadores de segundo nivel.

Un problema aún mayor es que, a pesar de que la arquitectura de interrupciones en dos niveles (sección 3.2.2) permite posponer el grueso del procesamiento de una interrupción a los manejadores diferidos. Todavía los tiempos de ejecución de los manejadores de primer nivel o ISRs difieren significativamente de una interrupción a otra. Como las ISRs se ejecutan con las interrupciones inhabilitadas todavía se hace muy difícil predecir el tiempo máximo durante el cual las interrupciones están inhabilitadas.

La solución a estas dificultades consistió en ejecutar los manejadores de interrupción en su propio contexto de hilos del núcleo. Bajo este esquema, todas las interrupciones (excepto la del temporizador) se dirigen hacia un manejador de bajo nivel en el núcleo cuyo único propósito es despertar a un hilo del núcleo correspondiente a la interrupción que previamente está durmiendo. Este hilo puede entonces ser ejecutado posteriormente con

todas las interrupciones habilitadas y bajo el control del planificador de hilos. Los hilos del núcleo dedicados al manejo de interrupción se pueden planificar en la clase de planificación de tiempo real (SCHED_FIFO) permitiendo además asignarles prioridades inferiores a las de los hilos convencionales de tiempo real.

Este esquema de interrupciones manejadas como hilos logra reducir la latencia de expropiación de tres formas:

1) Al permitir que los manejadores de interrupción se duerman, es posible reemplazar los cierres combinados de cierres de giro e interrupciones convencionales de Linux por mutexes que implementan el protocolo de herencia de prioridad permitiendo la expropiación de estas regiones críticas.
2) Como los hilos del núcleo destinados al manejo de interrupciones son expropiables, si llega una interrupción de mayor prioridad mientras se está ejecutando el hilo del núcleo, el hilo de ISR con mayor prioridad puede expropiar al de menor prioridad.
3) Todas las interrupciones ejecutan una ISR común que da el mismo servicio y consume el mismo tiempo de ejecución para todas las interrupciones. De este modo, se puede restringir la latencia de interrupción a un período fijo y corto.

La ejecución de los manejadores de interrupción como hilos en todas estas versiones de Linux posee varias diferencias significativas con respecto a la realización en Solaris.

1) En Solaris 2.x+, las prioridades de los hilos de interrupción se corresponden con las mismas prioridades de las interrupciones de hardware sólo que mapeadas a la parte más alta (de mayor prioridad) del espacio de prioridades de los hilos convencionales (sección 3.3.2). En Linux, los hilos de interrupción se ejecutan con interrupciones habilitadas a cualquier prioridad que se le asigne.
2) En Linux, la ISR general que se ejecuta cuando ocurren las interrupciones no esquiva la trayectoria de planificación normal del núcleo (como ocurre en Solaris 2.x+ – sección 3.3.2), sino que simplemente despierta al hilo de interrupción correspondiente el cual será entonces planificado del mismo modo que los demás hilos.

3.4 Tratamiento de Interrupciones en sistemas Embebidos y de Tiempo real

Las aplicaciones de cómputo tradicional propias de los sistemas de tiempo compartido aceptan una entrada, la procesan, y producen algún tipo de salida. A este tipo de sistemas de cómputo se les ha denominado **sistemas transformacionales** ("*transformational systems*") [34]. A diferencia de estos, la mayoría de los sistemas embebidos y de tiempo real se caracterizan por ser **sistemas reactivos** ("*reactive systems*") [34] o dirigidos por eventos ("*event-driven*") que pasan una gran parte del tiempo en un estado de espera por eventos o estímulos ante los cuales reaccionar. Una vez terminadas las acciones de manejo de estos eventos (reacción), tales sistemas regresan al estado de espera por el siguiente evento [81].

Los **estímulos primarios** son **generados por el hardware** a partir de eventos internos (provenientes del propio sistema de cómputo) o externos (provenientes del entorno exterior) y comunicados al software del sistema mediante el mecanismo de interrupciones

de hardware. El estimulo primario interno más común es el tic del reloj del sistema (que marca el paso del tiempo). Los estímulos primarios externos comprenden una amplia variedad dependiendo de la aplicación de que se trate. Ejemplos de este último tipo son la depresión de un botón (o tecla), un clic del ratón, la llegada de un paquete de datos, etc.

Al reconocer un evento, estos sistemas reaccionan llevando a cabo el cómputo apropiado para manejarlo. Esta reacción puede ser la emisión una acción de respuesta al entorno exterior (mediante la manipulación del hardware de salida del sistema) o la generación de otros **eventos secundarios** o de software. Ejemplo de eventos secundarios de software son el vencimiento de un temporizador ("*timeout*") lógico establecido por el software, la emisión de una solicitud de interrupción de software o la activación (arribo a la cola de tareas listas) de una tarea (hilo o proceso) que previamente no era ejecutable. Estos eventos de software provocan la ejecución de otros elementos de software. Por ejemplo, una petición de interrupción de software provoca la ejecución de un manejador de interrupción; el vencimiento de un temporizador provoca la ejecución de un manejador de vencimiento de tiempo ("*timeout handler*") y la activación de una tarea, trae como consecuencia la ejecución de la misma.

No importa que estos eventos sean internos o externos, una característica importante de los mismos es la impredecibilidad de su ocurrencia. La impredecibilidad de los eventos primarios se hace más evidente dado que, en el caso de los eventos externos, son emitidos fuera del control del software. En el caso de los eventos de software internos, un análisis ingenuo pudiera llevar a pensar que al ser generados por el propio software, tienen un carácter predecible. Sin embargo, aunque es cierto que son emitidos por el software del sistema, este lo hace sólo en respuesta a los eventos primarios (que están fuera de su control) lo cual lo convierte en impredecibles. Por ejemplo, los instantes en que se activa (desbloquea) o se desactiva un tarea a cargo de procesar los mensajes provenientes de la red dependen de la tasa con que estos mensajes se reciben y la longitud de los mismos, aspectos ambos fuera del control del sistema receptor.

3.4.1 Planificación del Tratamiento de los eventos

Un aspecto importante del entorno exterior es que este está compuesto por una variedad de elementos del mundo real (por ejemplo, servo mecanismos, actuadores, magnitudes físicas que caracterizan el estado de algún proceso externo controlado obtenidas a través de sensores) que operan o evolucionan en paralelo e independientemente los unos de los otros (y del sistema de cómputo). En consecuencia, el software encargado de interactuar con estos sistemas es concurrente por naturaleza. En cada instante pueden existir múltiples actividades de cómputo a ser ejecutadas. El núcleo del sistema operativo de tiempo real utiliza el mecanismo de multiprogramación para entrelazar la ejecución de las mismas según un algoritmo de planificación.

La ejecución de los elementos de software encargados de manejar los distintos eventos que suceden en el sistema (posiblemente de forma simultánea) consume recursos del sistema de cómputo (en un sistema de tiempo real, el tiempo de CPU es el recurso más importante). Debido a que estos recursos de cómputo son limitados y además debido al conflicto potencial en las demandas y el acceso a los mismos por parte de las distintas actividades de

cómputo, es posible que en el preciso instante en que se produjo el evento no estén disponibles los recursos necesarios para la ejecución del software encargado de su manejo (por ejemplo el tiempo de CPU se tiene que dedicar a ejecutar otra actividad cuyo plazo de respuesta es más urgente en ese instante). Como consecuencia, las actividades de software no pueden ser ejecutadas de forma inmediata a la ocurrencia del evento correspondiente; sino que, es necesario diferir su ejecución según algún algoritmo de planificación que permita satisfacer los requerimientos de todas las actividades de cómputo bajo las restricciones de recursos del sistema.

Dado que en los sistemas de tiempo real el recurso más importante es el tiempo, el algoritmo de planificación debe ser tal que posibilite el establecimiento de garantías del cumplimiento de los plazos de respuesta a los eventos externos. Esto se consigue mediante las pruebas de factibilidad de planificación ("*schedulability*") propias del algoritmo (ver sección 2.3). Como consecuencia, un algoritmo de planificación de tiempo real cumple la función de asignar el tiempo de cómputo de la CPU de forma tal que (bajo ciertas suposiciones de diseño y determinada caracterización del sistema) sea posible conocer por anticipado (o sea predecir) los tiempos de respuesta a eventos cuyo instante o patrón de llegada no se conoce con exactitud (o sea son impredecibles). En otras palabras, un planificador de tiempo real transforma las demandas asíncronas no predecibles en procesamiento síncrono planificado (según criterios preestablecidos) y predecibles[10]. En este sentido podemos concebir la planificación de tiempo real como un mecanismo que le "introduce" la predecibilidad al sistema y que, de alguna forma, funciona como una barrera ("*firewall*") a la impredecibilidad.

3.4.2 Entidades Planificables vs. Entidades No planifcables

A las actividades de cómputo que se ejecutan bajo el control del planificador de tiempo real se les denomina **entidades planificables** ("*schedulable entities*") y en su conjunto constituyen el **dominio de predecibilidad** del sistema. Idealmente (para poder establecer garantías de cumplimiento de los plazos), en un sistema de tiempo real, el planificador de tiempo real debería planificar todas las actividades dentro del sistema; sin embargo, en los sistemas reales algunas cosas están fuera de su control. Ejemplo de ello son: el servicio a las interrupciones (ejecución de las ISRs); los accesos directos a memoria o DMA; los manejadores de vencimiento de tiempos ("*timeout*") y algunas otras actividades que por alguna razón son ejecutadas con una "prioridad" mayor que el propio planificador del sistema[11]. Todas estas actividades utilizan una fracción de los recursos del sistema (como tiempo de CPU) pero no están controladas por el planificador de tiempo real por lo que se les denominan **entidades no planificables** ("*non-schedulable entities*"). Estas entidades no planificables le hacen estragos a la secuencia de ejecución preestablecida por el planificador y en conjunto constituyen un **dominio de impredecibilidad** lo cual constituye

[10] Aquí planificado se entiende como controlado

[11] En realidad estas actividades son planificadas por planificadores que poseen precedencia sobre planificador de tiempo real del núcleo del sistema. Por ejemplo, las actividades de DMA son planificadas por el controlador del bus del sistema y en muchos esquemas le "roban" tiempo a la CPU principal en los accesos al bus. Las ISRs son planificadas por el controlador de interrupciones del hardware según sus prioridades de hardware (como se vio en la sección 2.5.1). Los manejadores de vencimiento de tiempo son planificados por el manejador de interrupción del tic de reloj del sistema según sus instantes de vencimiento.

el anatema de un planificador de tiempo real[12]. Actualmente, los diseñadores y programadores de sistemas de tiempo real llevan a cabo grandes esfuerzos para reducir el número de entidades no planificables y la cantidad de tiempo de CPU que estas consumen (por ejemplo reducir al mínimo posible el tiempo de ejecución de las ISRs).

3.4.3 Utilización de variaciones de los modelos de Interrupciones tradicionales

A pesar de las diferencias de requerimiento existentes entre los sistemas operativos de propósito general y los sistemas operativos de tiempo real, la gran mayoría de los núcleos o sistemas operativos de tiempo real utilizan esquemas de manejo de interrupciones que cuando más son ligeras adaptaciones de los utilizados en los sistemas de propósito general.

3.4.3.1 Arquitectura simple de Manejador Unificado (o Sincronización por hardware)

El método usado con mayor frecuencia en sistemas operativos pequeños y ligeros consiste en darle tratamiento a las interrupciones directamente en las ISRs. Este es un método simple en el cual el sistema operativo no crea ninguna nueva abstracción de alto nivel para la administración de las interrupciones; sino que, hace uso de las abstracciones brindadas directamente por el hardware. En este aspecto, esta arquitectura es similar al esquema utilizado en los Unix tradicionales (ver sección 3.1); sin embargo, existen tres diferencias fundamentales con respecto a la arquitectura Unix tradicional:

1. En la arquitectura Unix tradicional básicamente las ISRs sólo pueden utilizar un servicio interno especial del núcleo para desbloquear al proceso en espera del evento (según el protocolo de dormir/despertar descrito en la sección 3.1.3). En la arquitectura de manejador unificado, a las ISR les está permitido invocar a cualquier servicio del sistema operativo para interactuar con las tareas o hilos de aplicación (o solicitar recursos del sistema) con la única restricción de que este servicio no tenga la posibilidad de bloquear a la tarea actual (ello podría paralizar el sistema).

2. Dado que en la arquitectura Unix tradicional, las ISRs no invocan servicios del núcleo, éstos no necesitan inhabilitar las interrupciones para protegerse de las ISRs[13]. En la arquitectura de manejador unificado, los servicios invocados dentro de una ISR pueden modificar estructuras de datos del núcleo. En consecuencia es necesario sincronizar de forma explícita el acceso a estas estructuras de datos internas. Como el código invocado desde una ISR no puede bloquearse para esperar a que estas estructuras sean liberadas, se necesita de algún mecanismo para demorar o postergar el inicio de la ejecución de la ISR, hasta tanto el recurso esté disponible. Esto se consigue inhabilitando brevemente las interrupciones mientras algún servicio del sistema está modificando estructuras de

[12] Aunque existen técnicas de análisis de factibilidad que le dan cabida a la existencia de entidades no planificables (ver sección 3.4.5), en el mejor de los casos, su presencia disminuye la eficiencia del planificador

[13] Aunque los servicios del núcleo no necesitan inhabilitar las interrupciones, si lo hacen algunas funciones internas del núcleo que pueden ser invocadas desde las ISRs como por ejemplo las funciones de sincronización que implementan el protocolo de dormir despertar. Además, los desarrolladores de manejadores de dispositivo tienen que inhabilitar las interrupciones para acceder las estructuras de datos compartidas entre las partes superior e inferior del manejador de dispositivo (ver sección 3.1.4).

datos críticas dentro del núcleo. Esto evita que cualquier otro programa o ISR haga cambios no coordinados a los datos críticos que están siendo utilizados por el código en ejecución.

3. En la arquitectura Unix tradicional, el núcleo es no expropiable (ver sección 3.1.2) lo que significa que si el código de la ISR provoca que se active una tarea de más prioridad que la tarea actual mientras la tarea actual está ejecutando algún código del núcleo, la conmutación de contexto (expropiación) se pospondrá hasta tanto la tarea actual abandone el código del núcleo. El carácter no expropiable del núcleo evita el acceso concurrente (en diferentes tareas) de las estructuras de datos internas. En la arquitectura de manejador unificado el núcleo del sistema operativo es expropiable. Esto significa que si, producto de una llamada a servicio realizada dentro de una ISR, se activa una tarea de mayor prioridad, la conmutación de contexto (expropiación) se producirá inmediatamente a la salida de la ISR Obsérvese que en este caso, no se requiere que el núcleo sea no expropiable ya que las secciones críticas del núcleo se protegen de forma explícita mediante la inhabilitación de interrupciones.

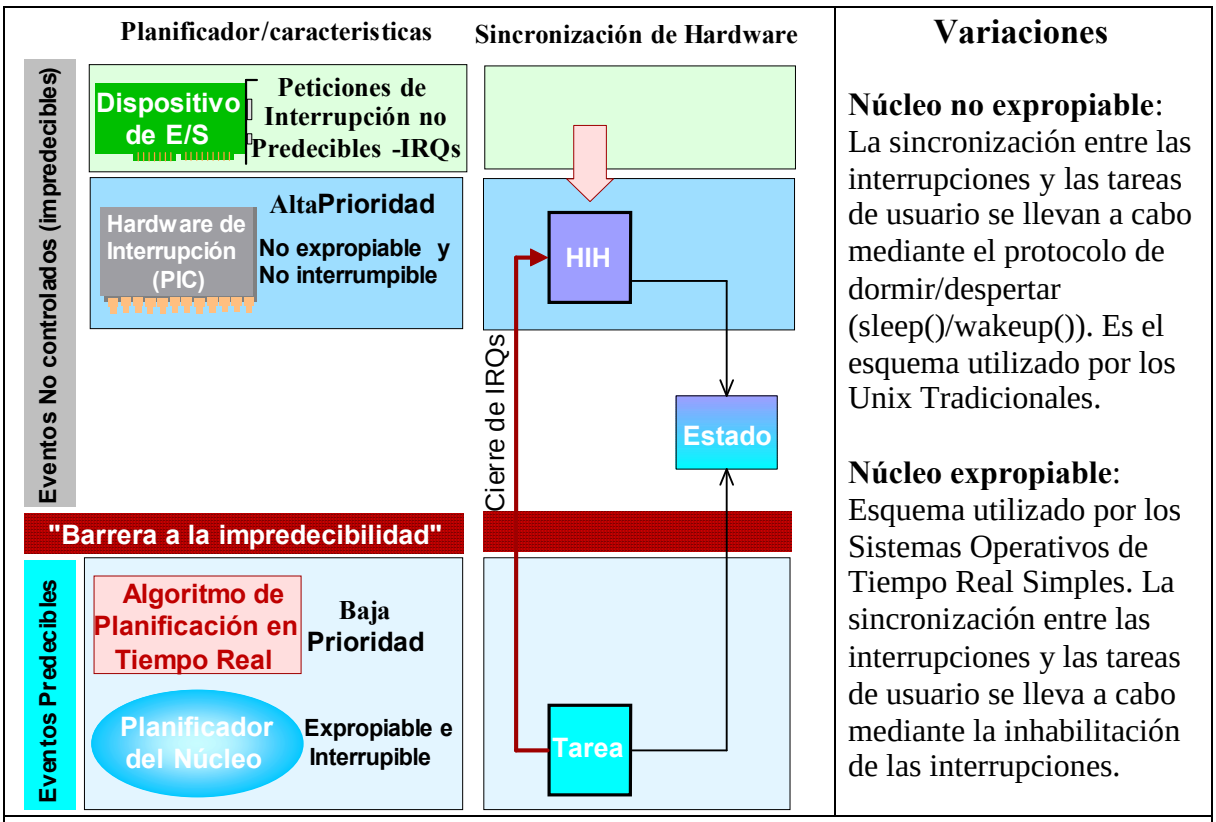


Figura 12. Arquitectura de Interrupción Simple Unificada (con y sin expropiación)

La Figura 12 muestra esquemáticamente esta arquitectura. En el extremo derecho de la figura se muestran los elementos que intervienen en el manejo de interrupciones en el sistema. En el extremo superior de la figura se muestra el dispositivo de E/S que genera las interrupciones (o estímulos primarios). Estas peticiones de interrupción llegan al controlador de interrupciones del hardware o PIC el cual se encarga de planificarlas de acuerdo a sus prioridades de hardware. La característica importante aquí es que, tanto la

emisión de las interrupciones por los dispositivos como la planificación de esta por parte del PIC está fuera del control del planificador del núcleo del sistema y por tanto son eventos no predecibles. En el extremo inferior de la figura aparece el planificador del núcleo del sistema el cual (en un sistema de tiempo real) ejecuta un algoritmo de planificación de tiempo real y por tanto los eventos (de software) que este emite constituyen eventos predecibles. En efecto, el planificador de tiempo real funciona como una barrera a la impredecibilidad. De este modo, la parte superior de la figura (dispositivos de E/S y controlador de interrupciones) constituyen el dominio de los eventos asíncronos no predecibles mientras que la parte inferior (bajo el control del planificador de tiempo real) constituye el dominio de los eventos síncronos predecibles.

En la parte central de la figura aparecen las distintas entidades de ejecución dentro del sistema. En la parte central superior se representa la emisión de una interrupción que provoca la ejecución de la ISR o Manejador de Interrupción de Hardware (HIH – "*Hardware Interrupt Handler*"). Este HIH está bajo el control del PIC y por tanto es una entidad no planificable. En la parte inferior se representan las tareas de la aplicación que se ejecutan bajo el control del planificador de tiempo real y por tanto son entidades planificables. El acceso al área de datos común entre tareas e ISRs (estado) se sincroniza mediante el cierre o inhabilitación de las interrupciones.

Al extremo derecho de la figura se describen dos variaciones de esta arquitectura: La arquitectura con núcleo expropiable y la arquitectura con núcleo no expropiable (típica de los sistemas Unix tradicionales). Observe que esta última puede verse como una variación de la arquitectura unificada en la cual se restringe el número de servicios que se pueden invocar dentro de la ISR a aquellos necesarios para implementar el protocolo de dormir/despertar a una tarea de usuario.

La sincronización del acceso a las secciones críticas dentro del núcleo mediante la inhabilitación de interrupciones tiene el inconveniente de que desactiva la capacidad de expropiación del sistema lo cual afecta adversamente el desempeño del sistema. Mientras más tiempo se pase en una sección crítica con las interrupciones desactivadas, mayor la degradación en la latencia de expropiación del sistema. De hecho, este esquema no brinda ninguna ventaja significativa con respecto al esquema de núcleo no expropiable de Unix a no ser que los períodos durante los cuales el sistema operativo inhabilita las interrupciones sean muy cortos.

A este método se le denomina la "Arquitectura de Interrupción Unificada" [50] porque todo el procesamiento de la interrupción se lleva a cabo en una única y "unificada" rutina de servicio de interrupción (ISR) o arquitectura de sincronización por hardware ("*hard synchronization*") [61] ya que la sincronización se consigue mediante la inhabilitación temporal de las interrupciones de hardware. Algunos Unix (por ejemplo Xinus [21]) y muchos núcleos comerciales de tiempo real (por ejemplo µC/OS-II [48], ThreadX [49], RTEMS [69], y la especificación OSEX/VDX [70]) utilizan esta arquitectura.

3.4.3.2 Arquitectura de Manejador Segmentado (o Sincronización por Software)

La idea general de esta arquitectura es dividir explícitamente el código de manejo de una interrupción en una parte crítica y una parte no crítica (ver Figura 13).

1. La parte crítica se ejecuta dentro del manejador de la interrupción de hardware o HIH con muy baja latencia y bajo el control del hardware de interrupciones. Esta debería llevar acabo sólo el procesamiento más crítico en tiempo y no hace uso de ningún servicio del núcleo. Antes de terminar, el HIH puede solicitar la ejecución de la parte que no es crítica mediante la solicitud de interrupciones de software.

2. Los Manejadores de Interrupciones de Software o SIH ("*Software Interrupt Handlers*") son planificados por el núcleo o, en algunas arquitecturas por el mecanismo de interrupciones de software de la misma CPU (lo cual se representa en la parte derecha de la Figura 13). La ejecución de estos manejadores se aplaza hasta tanto hayan terminado todos los manejadores de interrupciones de hardware anidados y ocurre antes de que se active al planificador de tareas (de tiempo real) del núcleo. Por tanto, estos SIH tienen prioridad sobre los hilos, pero son interrumpibles por los HIH si llegan nuevas señales de IRQ.

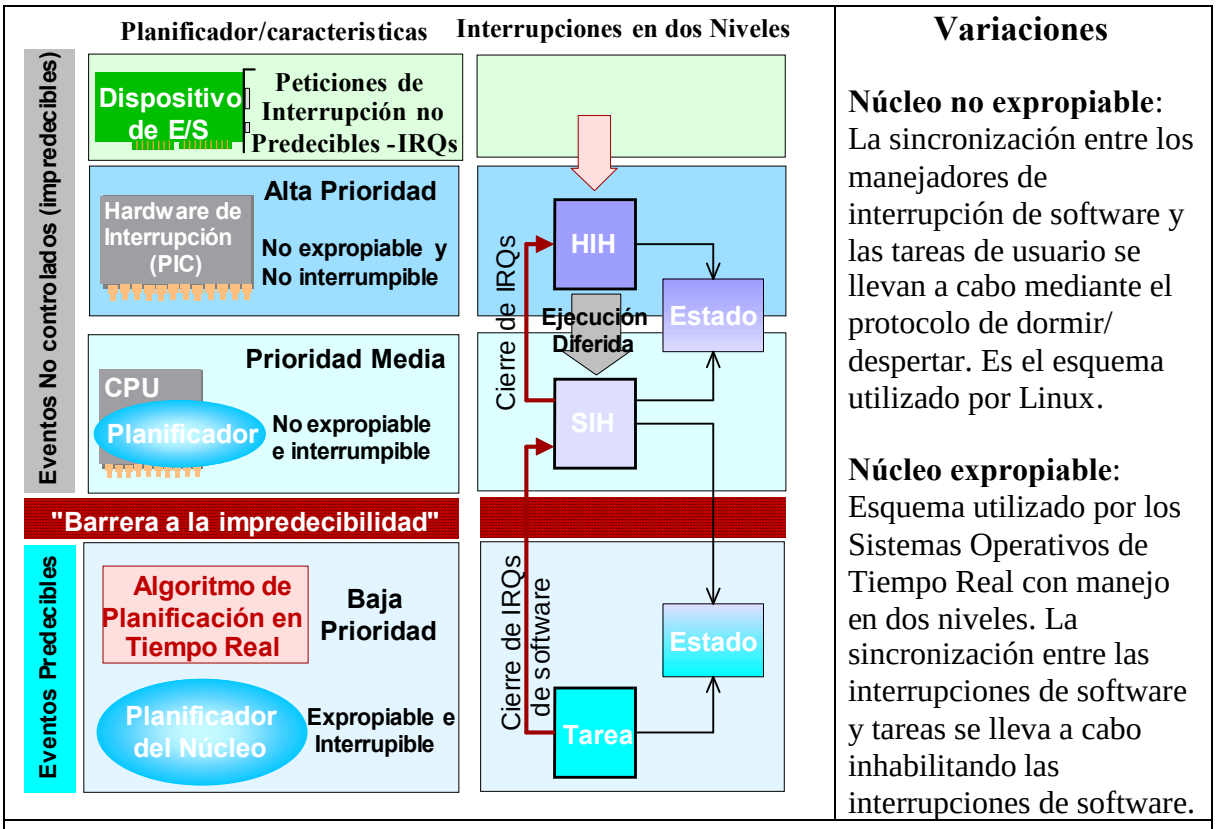


Figura 13. Arquitectura de Interrupción Segmentada (con y sin expropiación)

Esta arquitectura es muy similar al esquema utilizado en los sistemas operativos de red (ver sección 3.2) con la diferencia de que muchos de los sistemas operativos de red poseen un

núcleo no expropiable y los manejadores de interrupción de segundo nivel están restringidos a invocar sólo los servicios que implementan el protocolo de dormir/despertar (por ejemplo el esquema de "*top half/bottom half*" de Linux). Los sistemas operativos de tiempo real que utilizan este esquema poseen núcleos expropiables que sincronizan el acceso a sus estructuras internas mediante la inhabilitación temporal de la ejecución y los manejadores de segundo nivel. Además, los manejadores de segundo nivel pueden invocar cualquier servicio del núcleo no bloqueante.

Como ya se analizó en la sección 3.2, la arquitectura segmentada consigue reducir la latencia de interrupción mediante la reducción (o eliminación) de la necesidad de inhabilitar las interrupciones y la reducción de la duración de los manejadores de interrupción de primer nivel. Sin embargo, es importante destacar que, al igual que cualquier otra, esta arquitectura tiene que impedir el acceso concurrente a las estructuras de datos internas del núcleo. En consecuencia, aunque logran minimizar la latencia de interrupción evitando proteger sus secciones críticas mediante la inhabilitación de interrupciones, en su lugar lo hacen "demorando" o "inhabilitando" la invocación al planificador (y la conmutación de las tareas de aplicación) con lo cual mantienen una latencia de expropiación. En consecuencia, esta arquitectura logra minimizar la latencia de interrupción pero no así la latencia de expropiación que sigue ejerciendo un impacto negativo en el desempeño del sistema.

Además, como se aprecia en la Figura 13, los manejadores de interrupciones de software siguen siendo entidades no planificables fuera del control del planificador de tiempo real. El resultado de esto es que esta arquitectura sólo consigue disminuir la latencia de interrupción (y como se dijo antes no la de expropiación) sin embargo, no introduce ninguna mejora a la predecibilidad del sistema (ver sección 3.2.4).

Esta arquitectura de manejador segmentado es utilizada por muchos sistemas operativos embebidos y de tiempo real. Ejemplos de estos junto con la denominación que dan a cada uno de los niveles son: ISR/Rutina de Servicio Diferida – ISR/DSR o "*deferred service routine*" en eCos [65]; Prólogo/Epílogo – "*Prologue/Epilogues"* – en PEACE [82] y PURE [83]; ISR/Rutina de Servicio de Enlace – ISR/LSR o "*Link Sevice Routine*" en smx[67]; ISR/Llamada a función Demorada – ISR/DFC o "*Delayed Function Call*" – en Symbian OS [71].

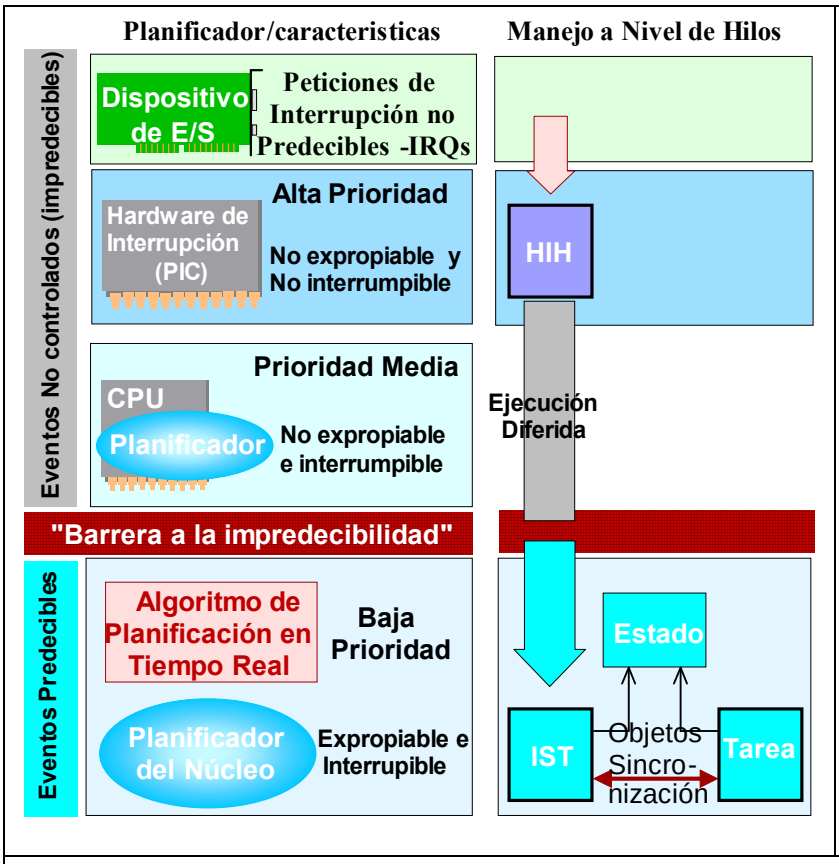
3.4.3.3 Manejo de Interrupciones a nivel de Hilos en sistemas de Tiempo Real

Con el propósito de disminuir la interferencia provocada por las interrupciones, la mayoría de los núcleos de tiempo real modernos realizan el segundo nivel de procesamiento de las interrupciones en una Tarea de Servicio de Interrupción o IST ("*Interrupt Service Task*") según se muestra en la Figura 14.

Se mantiene una pequeña ISR (denotada como HIH en la Figura 14) que lleva a cabo el procesamiento mínimo necesario para evitar la pérdida de datos y ejecuta un servicio especial suministrado por el núcleo que permite activar a una IST que se encuentra esperando mediante la ejecución del servicio de espera complementario. Esta IST se encarga entonces de llevar a cabo el servicio adicional que necesite la interrupción. Una vez activada, esta tarea se ejecutará (igual que cualquier otra) bajo el control del planificador

del núcleo y a una prioridad acorde con los requerimientos de la aplicación. ART [98][99], HARTIK [1] [19] [20], SPRING [89], Windows CE 3+ [33], Nemesis [53], QNX [37], Timesys Linux [96] son ejemplos de sistemas que utilizan esta estrategia de procesamiento.

Aunque esta estrategia consigue minimizar la perturbación producida por las ISRs la ejecución de las mismas todavía está fuera del control del planificador de tiempo real y por tanto no eliminan la impredecibilidad en el manejo de las interrupciones. La perturbación de las ISRs se vuelve significativa cuando la frecuencia de las interrupciones es alta, ya que esta frecuencia de interrupción tampoco es controlada por el planificador sino que depende de los eventos externos.



Variaciones

Interrupciones como IPC: Un manejador de bajo nivel en el núcleo transforma las peticiones de interrupción en IPC. Típico de la arquitectura de micro-núcleo.

Interrupciones como hilos: El tratamiento a las interrupciones se hace en hilos del núcleo.

Interrupciones como seudo-hilos: Las interrupciones se manejan en el contexto de hilos de núcleo, pero para agilizar la conmutación se posterga la conversión a hilo en el punto en que el manejador se bloquea

Figura 14. Arquitectura de Manejo de Interrupción a nivel de Hilo

3.4.4 Eliminación de las Interrupciones

Con el propósito de evitar las dificultades del modelo tradicional para aplicaciones en tiempo real (sección 2.6), se han propuesto varias alternativas. La más radical de ellas es evitar completamente el empleo de las interrupciones [41]. De hecho, Stewart cataloga el empleo indiscriminado de interrupciones como uno de los errores más comunes entre los programadores cuando desarrollan aplicaciones de tiempo real [91]. En su trabajo se recomienda que las ISRs de dispositivos que interrumpen periódicamente se conviertan en tareas periódicas debido a que se pueden planificar con algoritmos de planificación de tiempo real [60]. Siguiendo este enfoque se destaca la **arquitectura disparada por tiempo**

("*time-triggered architecture*") propuesta por Hermann Kopetz [46] la cual aboga por evitar las interrupciones a favor del enfoque basado en encuesta ("*polling-based*") para interactuar con los dispositivos. Varios sistemas operativos de tiempo real han optado por inhabilitar todas las interrupciones externas, excepto aquellas que provienen del temporizador (necesarias para las aplicaciones básicas del sistema). En este caso, todos los dispositivos periféricos se manejan ya sea por tareas de la aplicación, las cuales poseen acceso directo a los registros de las tarjetas de interfaz como sucede en el núcleo RK [51] en donde un proceso puede solicitarle al núcleo que convierta las interrupciones de dispositivo en eventos de tiempo; o por rutinas dedicadas dentro del núcleo, activadas periódicamente por la interrupción del temporizador y que se encargan de encuestar de forma periódica a los dispositivos como ocurre en el sistema MARS [25]. Cualquiera que sea el caso, debido a que no se generan interrupciones, las transferencias de datos tienen lugar mediante encuesta.

Aunque esta solución evita completamente el no-determinismo asociado a las interrupciones, tiene como desventaja fundamental una baja eficiencia en el uso de la CPU en operaciones de E/S, ya sea debido a la espera ocupada de las tareas mientras acceden a los registros del dispositivo así como al empleo de tareas de encuesta periódicas (y el compromiso entre velocidad de respuesta y sobrecarga a la hora de elegir el período adecuado de la encuesta).

3.4.5 Incorporación del costo de las interrupciones al análisis de Factibilidad.

Otra dirección menos radical consiste en utilizar las interrupciones pero modelar su efecto de forma que se pueda tener en cuenta su perturbación en las ecuaciones de factibilidad de planificación. En esta dirección se destaca el trabajo pionero de Jeffay y Stone [39] donde, utilizando el enfoque de demanda del procesador ("*processor demand*") descrito inicialmente en [10], los autores proponen una ecuación de recurrencia para obtener la cota superior del costo del manejo de interrupciones en ISRs durante cualquier intervalo de tiempo dado y suponiendo que se conocen los tiempos mínimos entre llegadas de las peticiones de interrupción (y los tiempos de cómputo de las ISRs asociadas) Su análisis es válido para esquemas de planificación tanto con prioridades estáticas como prioridades dinámicas. Más recientemente, varios trabajos han propuesto otros métodos de análisis que tienen en cuenta las interrupciones como las actividades de mayor prioridad en el sistema. Sandström, Eriksson y Fohler han propuesto un método de análisis de factibilidad de planificación que integra las técnicas de planificación estática, con técnicas de cálculo de tiempo de respuesta y reportan su aplicación industrial de forma satisfactoria [80]. Posteriormente, Mäki-Turja, Fohler y Sandström presentaron una modificación del análisis exacto de tiempo de respuesta que utiliza información acerca de los instantes de liberación y los plazos para obtener tiempos de respuesta más ajustados [39].

El trabajo de Brylow, Damgaard y Palsberg [17] presenta técnicas de análisis estático para el análisis de software dirigido por interrupciones al nivel del código ensamblador. Estas técnicas permiten obtener entre otros aspectos cotas máximas en la latencia de interrupción.

Stewart y Arora ampliaron la ecuación exacta de factibilidad de planificación de Lehoczky [52] para incluir la sobrecarga de las interrupciones en sistemas de prioridades estáticas.

Además se extiende el modelo de planificación presentado en [87] para tener en cuenta la sobrecarga del tratamiento de las interrupciones. La ecuación obtenida permite evaluar los compromisos entre realizar todo el tratamiento de la interrupción dentro de una ISR o posponer el grueso de su tratamiento a un servidor esporádico.

La convención actual para el uso de interrupciones en sistemas de tiempo real es programar las ISRs de modo tal que su única función sea enviar una señal a un servidor aperiódico [87]. Entonces se utilizan los métodos de análisis en tiempo real que tengan en cuenta la sobrecarga del manejo de interrupciones aquí presentados (por ejemplo [33] y[18])

Aunque los trabajos anteriores permiten cuantificar la interferencia de las interrupciones y tenerlo en cuenta a la hora de aplicar los análisis de factibilidad por si mismos no consiguen reducir (y menos evitar) la inversión de prioridad debido a las interrupciones. En la práctica los peores casos de esta inversión de prioridad arrojan resultados excesivamente pesimistas que demanda niveles de utilización media de la CPU excesivamente bajos aumentando considerablemente los costos del sistema para poder garantizar las respuestas a tiempo en los peores casos.

3.4.6 Manejo de Sobrecarga de Interrupciones

Aunque los trabajos anteriores permiten cuantificar la interferencia de las interrupciones en la práctica su empleo se ve muchas veces limitados por la imposibilidad de conocer con certidumbre el comportamiento temporal de las interrupciones (fundamentalmente ante situaciones no previstas del entorno). En esta situación los sistemas dirigidos por interrupción son vulnerables a la sobrecarga de interrupciones ("*interrupt overload*"): la condición en que las interrupciones externas son señaladas con la suficiente frecuencia como para que las demás actividades que se ejecutan en el procesador queden en inanición.

En un trabajo reciente Regehr [74] propone el empleo de un planificador de interrupciones que no es más que una pequeña pieza de hardware o software que limita la tasa de llegada de una fuente de interrupción. La implementación de software trabaja conmutando entre el comportamiento por interrupción cuando la tasa de llegada es baja y comportamiento tipo encuesta cuando la tasa de llegada es alta. Una técnica similar también fue implementada recientemente por Coutinho, Rufino y Almeida en RTEMS utilizando filtros digitales para modelar la tasa de llegada de las interrupciones y controlar la conmutación entre el tratamiento por interrupciones y por encuesta [23].

3.5 Resumen

En este capítulo hemos visto las diferentes estrategias utilizadas por los sistemas operativos para la administración de las interrupciones. Se ha puesto de manifiesto como los mecanismos que se han ido introduciendo han tenido como propósito fundamental aumentar el caudal de procesamiento y disminuir las latencias de interrupción y de planificación. En otras palabras, todas estas estrategias han logrado un mejor comportamiento en el caso promedio y una mayor rapidez de respuesta. Sin embargo, como se destacó en la sección 2.1, tiempo real no es sinónimo de rápido sino de predecible y desafortunadamente estas técnicas no han contribuido mucho a la predecibilidad del

sistema (operativo) en general ni tampoco resuelven todos las dificultades analizadas en el capítulo precedente (sección 2.6). En la comunidad de tiempo real podemos identificar dos enfoques fundamentales, el primero consiste en eliminar completamente el empleo de interrupciones y el segundo en la ampliación de los modelos de factibilidad de planificación para tener en cuenta el costo de las interrupciones (que se administran básicamente por los mismos mecanismos diseñados para sistemas operativos de propósito general). El primer enfoque es completamente radical y tiene fuertes implicaciones negativas en el uso eficiente de los recursos del sistema (incluyendo el consumo de energía); mientras que el segundo, tiene que ser complementado con técnicas de manejo de sobrecargas para poder lograr predecibilidad (sección 3.4.6) y aún así trae consigo un uso ineficiente de los recursos ahora debido a que las garantías de factibilidad de planificación sólo pueden alcanzarse con niveles extremadamente bajos de utilización de los recursos.

4 Mecanismo Integrado de Interrupciones y Tareas

En este capítulo presentamos nuestra propuesta de un modelo integrado de administración de interrupciones y tareas. La idea de este esquema parte de la observación de que, aunque la administración de la ISRs y las tareas se diferencia en todos los detalles, a un nivel más conceptual esas son actividades asíncronas completamente idénticas. Como consecuencia, un modelo que haga desaparecer todas las diferencias entre estas podrá resolver los problemas previamente identificados en la sección 2.6 a la vez que, en virtud de su simplicidad será más adecuada para el desarrollo de sistemas más confiables y seguros.

En este capítulo presentamos este modelo completamente integrado; así como, las ecuaciones para los análisis de factibilidad de planificación del modelo tradicional y del modelo integrado. Estas ecuaciones ponen de manifiesto las ventajas del modelo integrado desde el punto de vista de predecibilidad temporal y utilización de los recursos en sistemas de tiempo real. Posteriormente, presentamos el diseño de un subsistema de interrupciones que puede ser utilizado para incorporar este esquema integrado en un núcleo de tiempo real. Por último, hacemos un análisis comparativo de las características del modelo integrado con respecto a las alternativas existentes para el manejo de eventos externos en sistemas de tiempo real. Este análisis pone de manifiesto las ventajas de este modelo para este tipo de sistemas.

4.1 Modelo Integrado de Tratamiento de Interrupciones y Tareas

El modelo integrado de interrupciones y tareas consiste en la unificación de dos aspectos importantes de la administración de las interrupciones y las tareas: El mecanismo de sincronización y el mecanismo de planificación. Como resultado de ello se logra la existencia de un único tipo de actividad asíncrona al que denominamos tarea. La diferencia entre las tareas convencionales (periódicas, esporádicas) y las tareas que dan servicio a las interrupciones estriba únicamente en la naturaleza de su activación. Las primeras son activadas por eventos de software (vencimiento de un temporizador, señal de sincronización emitida por otra tarea, etc.) por lo que les denominamos **Tareas Activadas por Software** o SAT ("*software activated tasks*"); mientras que las segundas son activadas por interrupciones de hardware o IRQ, por lo que las denominamos **Tareas Activadas por Hardware** o HAT ("*hardware activated task*").

En este modelo integrado las HAT son las tareas que dan servicio a las interrupciones de hardware y en consecuencia sustituyen a las ISRs. Usamos el término HAT en lugar de, por ejemplo, **Tarea de Servicio de Interrupción** o IST ("*Interrupt Service Task*") para destacar la novedad del concepto y diferenciarlo de otros sistemas operativos (como los

descritos en las secciones 3.3 y 3.4.3.3) que sin llevar a cabo una integración total, también dan servicio a las interrupciones a nivel (o en el contexto) de tareas y que pueden estar utilizando el término IST.

A continuación detallamos el significado de este modelo integrado en términos de sus dos componentes y valoramos sus consecuencias y el modo en el cual se resuelven los problemas previamente planteados en la sección 2.6.

4.1.1 Integración del Mecanismo de Sincronización

La integración del mecanismo de sincronización se logra manejando todas las interrupciones en un **Manejador de Interrupciones de Bajo Nivel** o LLIH ("*Low Level Interrupt Handler*") universal situado al nivel más bajo del núcleo. Este LLIH convierte las IRQ en eventos de sincronización utilizando las mismas abstracciones u objetos de comunicación y sincronización entre tareas que soporte el modelo de concurrencia del sistema operativo (por ejemplo, semáforos buzones, etc).

Con este modelo, lo que antes eran ISRs ahora son HAT que permanecen en algún estado no elegible para ejecución hasta tanto ocurra la interrupción. El estado específico en que permanecen las HAT hasta tanto ocurra la interrupción depende del modelo de tareas. Por ejemplo, para modelos de tareas que se ejecutan permanentemente, las HATs pudieran bloquearse ejecutando un ***wait***() sobre un semáforo o una variable de condición asociada a la interrupción, (para esquemas basados en comunicación mediante memoria compartida), o utilizando un ***receive***() en espera de un mensaje (para esquemas que utilizan paso de mensajes). Cuando sucede la interrupción, el LLIH universal, al nivel más bajo del núcleo, hará lo necesario para hacer ejecutable la HAT (un *signal*() sobre el semáforo o variable de condición o un *send*() al buzón según sea el caso). Cualquiera sea el caso, el efecto será que se ponga lista una tarea que antes no lo estaba.

Esta variante suministra una abstracción que delega al núcleo los detalles del tratamiento de bajo nivel de la interrupción, a la vez que elimina las diferencias entre las ISRs y las tareas. El servicio real de la interrupción descansa aún dentro de la HAT, suministrando total flexibilidad y haciendo innecesario que el núcleo se ocupe de las particularidades específicas del tratamiento de las distintas interrupciones.

La existencia de un tipo único de actividad asíncrona y mecanismos uniformes de sincronización y comunicación entre ellas ofrece, entre otras, las siguientes ventajas:

- Las HATs se ejecutan en un ambiente en donde pueden invocar sin restricción cualquier servicio del núcleo o de alguna biblioteca.

- Facilita el desarrollo y mantenimiento del sistema, por contar con un mecanismo único de sincronización y comunicación entre actividades cooperantes.

- Elimina completamente la necesidad de que las tareas de aplicación inhabiliten y habiliten las interrupciones lo cual le permite al núcleo garantizar el peor caso en el tiempo de respuesta a los eventos externos (sección 2.6.4).

- Al eliminar la necesidad de utilizar el nivel de interrupción como medio de sincronización, no es necesario que los componentes de software conozcan el nivel de interrupción más alto que lo invoca. Facilitando el desarrollo independiente de componentes y bibliotecas de software sin necesidad de, ante la imposibilidad de determinar exactamente dicho nivel, inhabilitar completamente las interrupciones.

- Se elimina la dificultad asociada al levantamiento de excepciones como parte de los manejadores de interrupción (sección 2.6.2) ya que estos se ejecutan en su propio contexto de tarea.

Observe que esta integración es más completa de la que se logra con la implementación de interrupciones como hilos en Solaris o Linux (sección 3.3) ya que en estos sistemas lo que sucede en realidad es que las ISRs se ejecutan en el contexto de hilos dedicados; sin embargo, la activación de las mismas no se hace utilizando los mecanismos de sincronización estándares. En otras palabras el tratamiento de interrupciones se sigue realizando mediante un conjunto de servicios específicos que permiten la instalación de ISRs. Solo que las ISRs ahora se ejecutan en su propio contexto de hilo y no en el contexto del hilo interrumpido.

4.1.2 Integración del Mecanismo de Planificación

La unificación del mecanismo de sincronización es sólo un paso necesario pero no suficiente. El mecanismo integrado incluye un espacio de prioridades dinámicas unificado y flexible para todas las actividades (sean activadas por eventos de software o de hardware). El planificador del núcleo y el hardware de interrupciones cooperan para planificar todas las actividades asíncronas sean SAT o HAT utilizando un espacio unificado de prioridades bajo un mismo algoritmo de planificación (ver Figura 15).

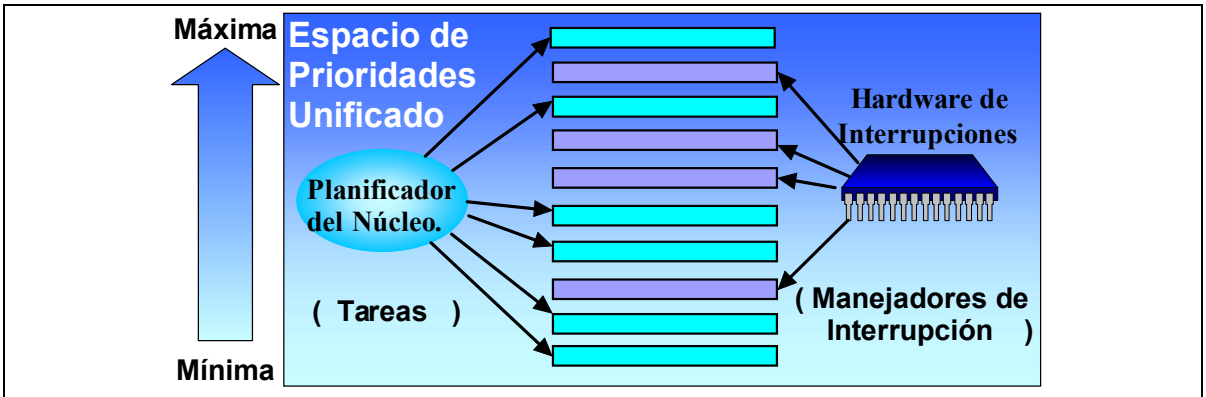


Figura 15. Prioridades en el esquema integrado

Es importante destacar que esta planificación integrada incluye la planificación de las IRQs (no sólo la HAT asociada). Esta es una diferencia fundamental con otras propuestas (como las descritas en las secciones 3.3 y 3.4.3.3) que integran la planificación de las actividades (hilos) de interrupción y en las que todavía las mismas IRQs quedan completamente fuera del control del planificador y por tanto pueden llegar en cualquier momento. La necesidad

de planificar las IRQ como tal es la razón por la cual es necesaria la participación del hardware de interrupciones en la planificación.

Este esquema de planificación unificado permite que a todas las actividades del sistema de tiempo real se le asignen prioridades en correspondencia con sus restricciones de tiempo. Con ello, se obtienen las siguientes ventajas:

- Se evita la implementación de un protocolo de entrada/salida (**EnterISR-LeaveISR**) para registrar las ISRs en el núcleo evitando los errores potenciales (sección 2.6.1.2).

- Se evita la inversión de prioridad asociada al espacio de prioridades independiente (sección 2.6.3).

- Se elimina el error del cierre de interrupción roto, producto de la conmutación de tareas (sección 2.6.1.1).

- Las situaciones de sobrecargas de interrupciones se pueden manejar utilizando cualquiera de las técnicas de planificación conocidas para el manejo de sobrecarga; como por ejemplo, el empleo de un servidor esporádico (ver [87]).

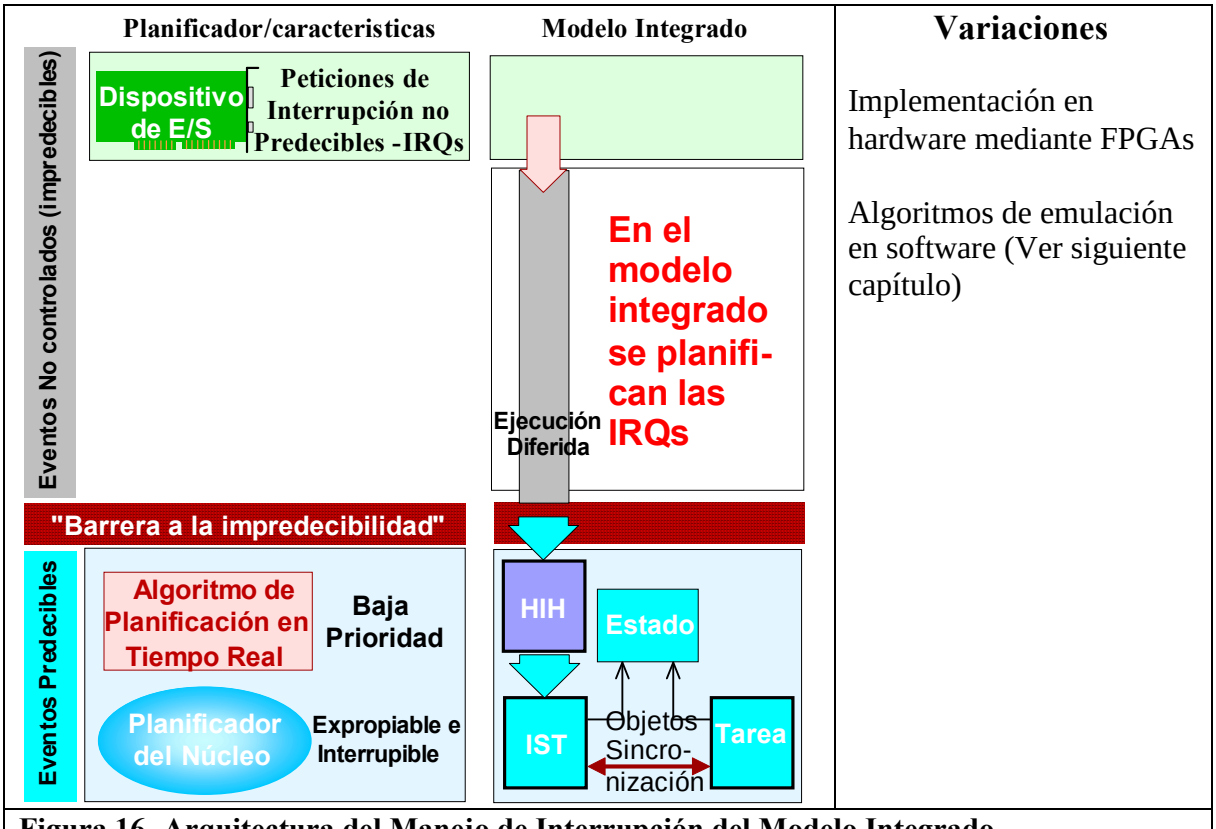


Figura 16. Arquitectura del Manejo de Interrupción del Modelo Integrado

4.1.3 Arquitectura de Interrupciones del Modelo Integrado

La Figura 16 muestra un esquema de la arquitectura del manejo de interrupciones del modelo integrado aquí propuesto y que puede ser contrastada con las figuras correspondientes a los esquemas tradicionales presentadas en la sección 3.4. Como puede apreciarse, la diferencia fundamental de nuestro esquema con respecto a los esquemas anteriores es que ahora las mismas IRQ están bajo el control del planificador de tiempo real del sistema. Por consiguiente, ahora el Manejador de Interrupciones de Hardware ejecuta el LLIH bajo el control del planificador de tiempo real convirtiéndose en una entidad planificable. Como consecuencia, nuestro esquema integrado ha eliminado completamente las entidades no planificables presentes en las arquitecturas tradicionales de manejo de interrupciones dando como resultado un esquema de manejo de interrupciones completamente predecible.

4.2 Análisis de la Factibilidad de Planificación en Ambos Modelos Incorporando las Interrupciones

Con el propósito de valorar la factibilidad del modelo integrado desde el punto de vista de la planificación en tiempo real, a continuación hacemos un análisis de la disminución en la cota de planificación y el aumento del tiempo de respuesta producto de la utilización de espacios de prioridades independientes en el modelo tradicional. Posteriormente analizamos la disminución de la cota de planificación introducida por la conmutación de contexto del modelo integrado. Estos resultados permitirán valorar bajo que condiciones es más apropiado un modelo u otro.

4.2.1 Disminución de la Cota de Utilización de Tasa Monótona

Según la teoría del Análisis de Tasa Monótona o RMA, una tarea t_i es factible de planificar utilizando la asignación de prioridades monótona en plazo si se cumple que:

$$U_{\text{lub}} \geq U_i \qquad (1)$$

dónde U_i es la **menor cota superior de utilización** ("*least upper utilization bound*") de la CPU, que para una asignación de prioridades estáticas como la Planificación de Tasa monótona ("*Rate Monotonic Scheduling*") es $i(2^{1/i}-1)$, o 1 si se utiliza una asignación de prioridades dinámico como Primero el Plazo más Próximo ("*Earliest Deadline First*"). Se supone que la U_i es la fracción de utilización de la CPU debido a la tarea t_i, más la fracción de utilización de la CPU producto de la **interferencias** de las tareas de mayor prioridad. Esto se puede expresar como:

$$U_i = \frac{C_i}{T_i} + \sum_{j \in P(i)} \frac{C_j}{T_j} \qquad (2)$$

donde C_i es el tiempo de cómputo de la tarea t_i y T_i es su período. $P(i)$ es el conjunto de tareas t_j de mayor prioridad que t_i con período $T_j < T_i$ y tiempos de cómputo C_j.

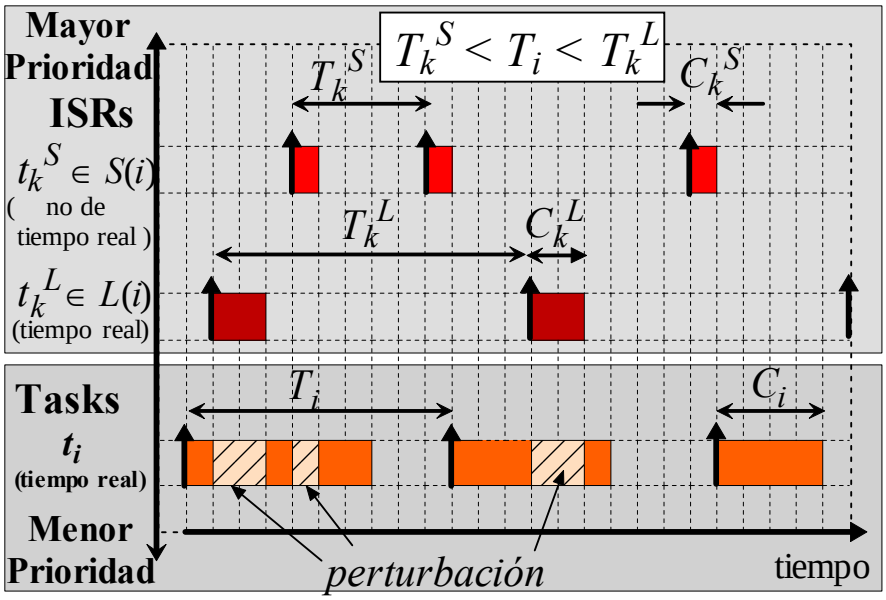


Figura 17. Perturbaciones Temporales debido al espacio prioridades de tareas e interrupciones independiente.

La **perturbación temporal** ("*timing disturbance*") que ejercen las *ISRs* sobre la planificación de una tarea t_i de tiempo real crítico se puede describir utilizando la Teoría de Planificación en Tiempo Real Generalizada [44]. Como se muestra en la Figura 17, en este caso tenemos dos tipos de perturbaciones:

- Aquella asociada a interrupciones con tiempos mínimos entre llegadas inferiores a los de la tarea de tiempo real t_i pero que no están asociadas a tareas sin requerimientos de tiempo real (o con requerimientos de tiempo real suave). A esta perturbación le llamaremos **perturbación debido a tareas no críticas**. Denotemos por $S(i)$ al conjunto de ISRs t_k^S con estas características, cada una con tiempos de cómputo C_k^S y períodos $T_k^S < T_i$. La utilización de una ISR t_k^S en el conjunto $S(i)$ está dada por C_k^S/T_k^S.

- Aquella asociada a ISRs con requerimientos de tiempo real crítico, pero que poseen un tiempo mínimo entre llegadas mayor que el de la tarea t_i. A esta perturbación se llama **inversión de prioridad monótona en tasa**. Denotemos por $L(i)$ el conjunto de ISRs t_k^L con estas características y por C_k^L el tiempo de cómputo de cada una de ellas. Como los tiempos entre llegadas T_k^L de estas interrupciones son mayores que T_i, sólo pueden expropiar a t_i una sola vez. En consecuencia, la utilización en el peor caso debido a una ISR en el conjunto $L(i)$ está dada por C_k^L/T_i.

La ecuación de la cota de utilización teniendo en cuenta estas dos afectaciones quedaría entonces como:

$$U_i = \left(\frac{C_i}{T_i} + \sum_{j \in P(i)} \frac{C_j}{T_j} \right) + \left(\sum_{k \in S(i)} \frac{C_k^S}{T_k^S} + \frac{1}{T_i} \sum_{k \in L(i)} C_k^L \right) \qquad (3)$$

Los dos primeros términos son idénticos a los de la ecuación (2). Por tanto, el tercer y el cuarto término se pueden interpretar como la disminución que se produce en la **menor cota superior de utilización** (U_{lub}) debido a la utilización de un espacio independiente de prioridades de interrupción. Sea esta disminución U_{is}, entonces la ecuación (1) se puede escribir:

$$U_{net} = U_{\text{lub}} - U_{is} \tag{4}$$

donde:

$$U_{is} = \sum_{k \in S(i)} \frac{C_k^S}{T_k^S} + \frac{1}{T_i} \sum_{k \in L(i)} C_k^L \tag{5}$$

Obsérvese sin embargo que la ecuación (3) no tiene en cuenta la afectación a la utilización debido a la necesidad de inhabilitar las interrupciones. Sea I_L el tiempo máximo durante el cual se inhabilitan las interrupciones en cualquier parte del sistema, entonces la ecuación (3) se puede extender fácilmente como:

$$U_i = \left(\frac{C_i}{T_i} + \sum_{j \in P(i)} \frac{C_j}{T_j} \right) + \left(\sum_{k \in S(i)} \frac{C_k^S}{T_k^S} + \frac{1}{T_i} \left(\sum_{k \in L(i)} C_k^L + I_L \right) \right)$$

Por tanto, la disminución de la cota de planificación teniendo en cuenta la inhabilitación de interrupciones U_{is}^* quedaría como:

$$U_{is}^* = \sum_{k \in S(i)} \frac{C_k^S}{T_k^S} + \frac{1}{T_i} \left(\sum_{k \in L(i)} C_k^L + I_L \right) \tag{6}$$

4.2.1.1 Solución tradicional

Con el propósito de minimizar U_{is}, el código de las ISRs (C_k^S, C_k^L) debe ser mínimo. De esta forma, una ISR sólo hará el procesamiento necesario para evitar la pérdida de datos y activar una tarea que llevará a cabo el servicio adicional que necesite la interrupción. Precisamente este es el fundamento para el esquema manejo de interrupciones a nivel de hilos previamente descrito en la sección 3.4.3.3 (y la Figura 14) del capítulo anterior en el cual el manejo de la interrupción se divide en una pequeña ISR o HIH y una Tarea de Servicio de Interrupción o IST. Con este esquema, la respuesta real a un evento ocurre entonces dentro de esta tarea con la cuál la ISR se comunica. Una vez activada, esta tarea se ejecutará, igual que cualquier otra, bajo el control del planificador del núcleo y se le puede asignar una prioridad según los requerimientos de la aplicación.

La Figura 18 muestra el comportamiento temporal del mismo conjunto de tareas mostrado en la Figura 17 pero ahora utilizando este esquema de manejo de interrupciones a nivel de hilos. Como se puede observar, ahora cada una de las ISRs (actividades en los conjuntos $S(i)$ y $L(i)$) hacen un procesamiento mínimo y provocan la activación de las correspondientes ISTs. Como puede observarse en la Figura 17, estas ISTs pertenecen al

conjunto $U(i)$ de tareas que poseen menor prioridad que la tarea de t_i y por tanto no interfieren con la ejecución de la tarea t_i

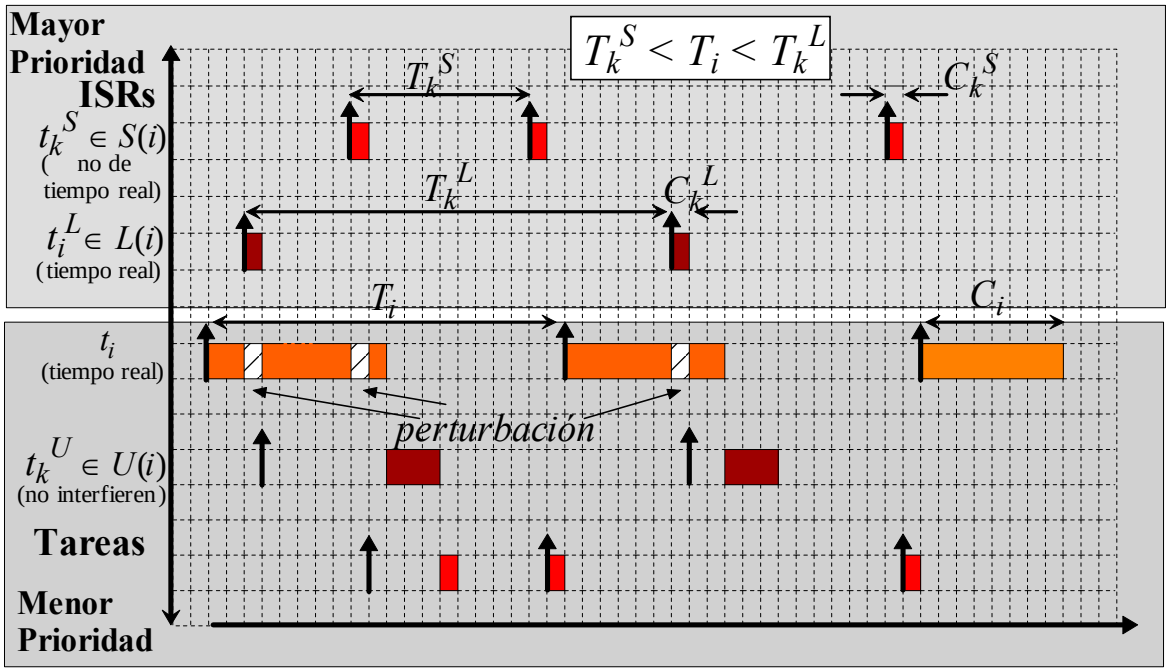


Figura 18. Perturbación temporal utilizando el esquema de manejo de interrupciones en dos niveles (ISR y tareas).

En realidad, aunque esta estrategia consigue minimizar la perturbación producida por las ISRs, no resuelve el problema de la predecibilidad. Este problema se origina por la incapacidad de predecir la frecuencia de las interrupciones provenientes de todos los dispositivos en el sistema (T_k^S). La ocurrencia de demasiadas interrupciones en un pequeño intervalo de tiempo hace al sistema no predecible y puede provocar que algunas tareas incumplan sus plazos. Con el propósito de abordar este problema algunos sistemas introducen mecanismos adicionales para limitar el número de interrupciones durante ciertos intervalos de tiempo [68], [74]. Sin embargo, está claro que estos mecanismos introducen una sobrecarga operativa adicional.

4.2.2 Incremento en el tiempo de respuesta de las tareas (a eventos externos)

Con el esquema anterior, el tiempo de respuesta de extremo a extremo real del evento coincide con el tiempo de respuesta en el peor caso de la tarea asociada con la cual la ISR se comunica. En [18] se ofrecen ecuaciones para obtener el tiempo de respuesta real a un evento externo utilizando varias estrategias para modelar las ISRs. Sin embargo, nosotros estamos interesados en obtener la afectación que ejerce la existencia de dos espacios de prioridades independientes sobre el tiempo de respuesta.

Como es de suponer la existencia de espacios de prioridades independientes se debe reflejar en un aumento en el tiempo de respuesta de las tareas. El tiempo de respuesta R_i de una

tarea t_i con tiempo de cómputo C_i y tiempo mínimo entre llegadas T_i se puede obtener mediante la siguiente ecuación de recurrencia [7]:

$$R_i^n = C_i + B_i + \sum_{j \in P(i)} \left\lceil \frac{R_i^{n-1}}{T_j} \right\rceil C_j \qquad (7)$$

Donde R_i^0 representa el valor inicial de R_i (puede tomarse como C_i) y R_i^n es el n-ésimo valor iterativo. $\boldsymbol{B_i}$ es el tiempo de bloqueo que experimenta la tarea t_i y $P(i)$ el conjunto de tareas con mayor prioridad que $\boldsymbol{t_i}$. El tercer término de esta ecuación representa la interferencia total que experimenta $\boldsymbol{t_i}$ debido a las tareas en el conjunto $P(i)$. Esta iteración termina satisfactoriamente cuando se encuentra un tiempo de respuesta $R_i = R_i^{n-1} = R_i^n$; o no satisfactoriamente, cuando $R_i^n > D_i$. Siendo D_i el plazo de la tarea t_i.

Para tener en cuenta el efecto que sobre el tiempo de respuesta de la tarea $\boldsymbol{t_i}$ ejerce la existencia de un espacio de interrupciones independientes, tenemos que añadirle a la ecuación (7) la interferencia que los conjuntos de ISRs $S(i)$ y $L(i)$ ejercen sobre el tiempo de respuesta. La ecuación quedaría como:

$$R_i^n = \left(C_i + B_i + \sum_{j \in P(i)} \left\lceil \frac{R_i^{n-1}}{T_j} \right\rceil C_j \right) + \left(\sum_{k \in S(i)} \left\lceil \frac{R_i^{n-1}}{T_k} \right\rceil C_k^S + \sum_{k \in L(i)} C_k^L \right) \qquad (8)$$

Para una mejor interpretación la ecuación está dividida en dos secciones delimitadas por paréntesis. La primera sección incluye tres términos idénticos a los de la ecuación (7). El primer término de la segunda sección introduce la afectación debida las ISRs en el conjunto $S(i)$ (que pueden expropiar múltiples veces a la tarea t_i) mientras que el segundo término de la segunda sección introduce la afectación debida a las ISRs en el conjunto $L(i)$ (que en el peor caso, sólo pueden expropiar una vez a la tarea t_i). Esta segunda sección se puede interpretar como la afectación en el tiempo de respuesta R_i debido a la utilización de un espacio de prioridades de interrupción independiente. Sin embargo, debido a que (8) es una ecuación de recurrencia, no podemos cuantificar ambas secciones de forma independiente como se hizo para el caso de la utilización. Mucho más importante es la observación de que un incremento muy pequeño en la contribución de la segunda sección de la ecuación puede dar como resultado un incremento muy grande en el tiempo de respuesta de la tarea que puede llevar al incumplimiento del plazo.

4.2.3 Sobrecarga por la introducción de la conmutación de contexto entre tareas.

El inconveniente de un tratamiento integrado como el propuesto aquí, está dado por la sobrecarga introducida en la conmutación de contexto hacia tareas que antes se trataban como ISRs. Esta sobrecarga se refleja, como es natural, en una disminución del límite superior de la cota de planificación.

Sea $H(i)$ el conjunto de todas las actividades t_j^H con un tiempo de ejecución C_j^H y tiempo mínimo entre llegadas T_j^H menor que el período T_i de la tarea t_i y cuyo tratamiento se hace en una ISRs en el modelo tradicional. Sea δ^I el tiempo total de CPU consumido por el

esquema de entrada/salida a la ISR necesario para salvar y restaurar el estado de la CPU y registrar el anidamiento de las ISRs (por ejemplo mediante el protocolo EnterISR-LeaveISR descrito en la sección 2.6.1.2). Sea c_j^H el tiempo de CPU del código de atención a la interrupción como tal. Entonces, bajo el esquema tradicional, el tiempo total de CPU de una ISR en el conjunto $H(i)$ se puede expresar como:

$$C_j^H = c_j^H + \delta^I$$

Entonces, en la ecuación (2), el segundo término puede descomponerse en dos partes: el consumo de CPU debido a las ISRs que, en virtud de sus períodos inferiores al período de la tarea t_i, les corresponde una prioridad de tasa monótona mayor a la prioridad de la tarea t_i (conjunto $H(i)$) y el resto de las tareas de mayor prioridad que no son ISRs (conjunto $P(i) - H(i)$). Haciendo esta descomposición, la ecuación (2) se puede escribir como:

$$U_i^I = \frac{C_i}{T_i} + \sum_{j \in (P(i)-H(i))} \frac{C_j}{T_j} + \sum_{j \in H(i)} \frac{c_j^H + \delta^I}{T_J^H} \qquad (9)$$

Por el contrario, en el modelo integrado todas las actividades en el conjunto $H(i)$ se tratan como tareas. Sea δ^P el tiempo de conmutación de contexto del sistema. Entonces, el consumo de CPU C_j^H de una tarea en el conjunto $H(i)$ se puede expresar como:

$$C_j^H = c_j^H + 2\delta^P$$

Donde $2\delta^P$ tiene el cuenta el tiempo de las dos conmutaciones de contextos asociadas a la expropiación para la ejecución de una tarea en H(i) y al regreso de esta a la tarea expropiada. En consecuencia, para el casi del modelo integrado, la ecuación (2) se puede escribir como:

$$U_i^P = \frac{C_i}{T_i} + \sum_{j \in (P(i)-H(i))} \frac{C_j}{T_j} + \sum_{j \in H(i)} \frac{c_j^H + 2\delta^p}{T_J^H} \qquad (10)$$

Por tanto, la disminución en la utilización U_i^{PI} debido a la sobrecarga asociada al tratamiento de las actividades en el conjunto $H(i)$ como tareas, en lugar de hacerlo como ISRs. Que es lo mismo que la disminución en la utilización debido al empleo de un modelo integrado de tratamiento de interrupciones y tareas, está dado por:

$$U_i^{PI} = U_i^p - U_i^I = \sum_{j \in H(i)} \frac{c_j^H + 2\delta^p}{T_j} - \sum_{j \in H(i)} \frac{c_j^H + \delta^I}{T_j} = \sum_{j \in H(i)} \frac{2\delta^p - \delta^I}{T_J^H}$$

$$U_i^{PI} = \sum_{j \in H(i)} \frac{\delta}{T_J^H} \qquad (11)$$

donde $\delta = 2\delta^p - \delta^I$ es la diferencia entre el tiempo de cómputo de dos conmutaciones de contexto y el tiempo de cómputo del protocolo de entrada/salida a la ISR.

La sobrecarga del modelo integrado será menor que el efecto de inversión de prioridad del modelo tradicional si se cumple la siguiente condición:

$$U_i^{PI} < U_{iS} \qquad (12)$$

A partir de la ecuación (12), si se compara la disminución en la cota de planificación debido al empleo del modelo tradicional de tratamiento de interrupciones U_{iS} (ecuación 5) y U_{iS}^* (ecuación 6), con la disminución que introduce el modelo integrado U_i^{PI} (ecuación 9) debido a la sobrecarga adicional en la conmutación de contexto, es posible observar que en la mayoría de los casos el ahorro que se obtendría mediante el tratamiento al nivel de ISR utilizando el modelo tradicional, es mucho menor que la reducción de la cota de factibilidad de planificación que este introduce, debido a los diferentes tipos de inversión de prioridad que trae aparejado.

En cualquier caso, pudiera ser posible diseñar un modelo híbrido con una configuración en la cual algunas actividades son tratadas como ISRs y otras como HATs para satisfacer la condición establecida en (12). Por ejemplo, cómo la interrupción del temporizador siempre tendrá la mayor prioridad en el sistema y nunca será manejada por una aplicación, podría ser considerada una ISR. Esto reduce el conjunto H(i) y por tanto disminuye U_i^{PI}.

4.3 Diseño del Subsistema de Administración de Interrupciones de Bajo Nivel para un micro-núcleo experimental.

En esta sección damos una breve descripción del diseño de un subsistema de administración de interrupciones de bajo nivel para la implementación del modelo integrado de administración de interrupciones y tareas anteriormente descrito. Hemos establecido como requerimiento que este subsistema suministre un nivel de abstracción tal que permita que el resto del núcleo sea completamente independiente del hardware de interrupciones específico de la plataforma.

El diagrama UML de la Figura 19, muestra las relaciones entre los diferentes componentes[14] del núcleo involucrados en la administración de interrupciones. Todos los componentes del núcleo que tienen que interactuar con el componente de administración de interrupciones lo hacen a través de la interfaz **iKRNLINT**. Por tanto, cualquier implementación del subsistema de administración de interrupciones tiene que suministrar una realización de esta interfaz.

[14] Independientemente de cualquier significación que se le de al término componente en el contexto del diseño de software basado en componentes, en este trabajo entendemos por componente una unidad de software en formato ejecutable que brinda una abstracción bien definida y a cuyos servicios; así como a su configuración, se pueden acceder a través de una interfaz de modo que puede ser: (1) intercambiado fácilmente por otro componente que implemente la misma interfaz o, reutilizado en otro proyecto de software que haga uso de la misma interfaz.

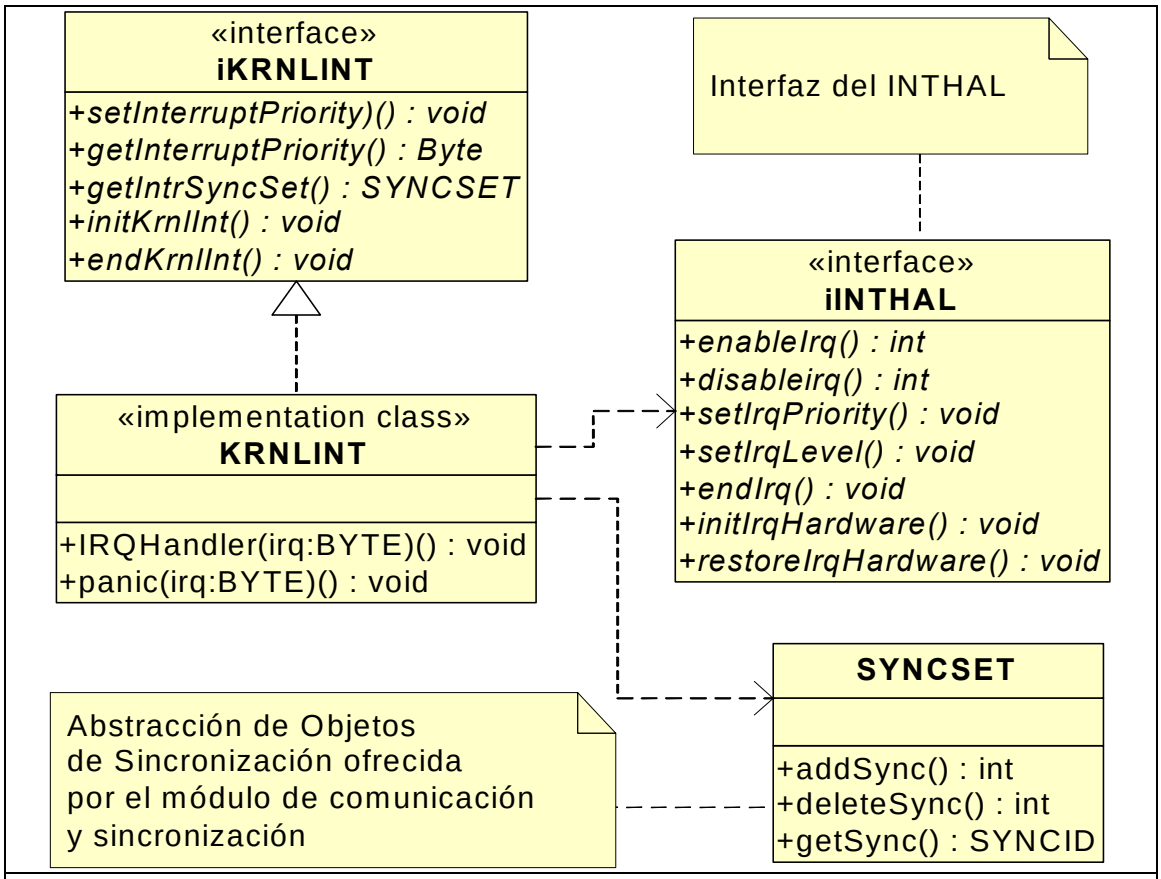


Figura 19. Diagrama UML que muestra los diferentes módulos del subsistema de administración de interrupciones de bajo nivel del núcleo.

A su vez, el componente administrador de interrupciones está subdividido en otros dos componentes. Uno denominado Módulo de Administración de Interrupciones del Núcleo (**KRNLINT**) que contiene el código de administración independiente del hardware, y otro que contiene el código dependiente del hardware y que recibe el nombre de Capa de Abstracción del Hardware de Interrupciones (**INTHAL**). Toda la comunicación entre el componente **KRNLINT** y el componente **INTHAL** se realiza a través de la interfaz **iINTHAL**.

4.3.1 Módulo de Administración de Interrupciones del Núcleo

La responsabilidad fundamental del módulo de administración de interrupciones del núcleo es el suministro de mecanismos de bajo nivel que le permitan al resto del sistema (en específico a los módulos de planificación y de comunicación y sincronización) tratar a las interrupciones con las mismas políticas de planificación o de sincronización de tiempo real que aplica a las tareas. Entre sus responsabilidades se encuentran:

- permitir la asociación de **objetos de sincronización** (abstracciones de sincronización de alto nivel que soporte el modelo de concurrencia del núcleo – semáforos, buzones, etc) todos ellos identificados por un **identificador de**

sincronización de tipo **SYNCID** único en el sistema, a cada una de las líneas de petición de interrupción del hardware;

- generar una señal sobre los objetos de sincronización cada vez que se produzca una petición de interrupción por la línea asociada; así como,

- suministrar los mecanismos que permitan la administración de las prioridades de las interrupciones de forma dinámica.

Para ello este módulo crea la abstracción de **Interrupción del Núcleo** cada una de las cuales se identifican por unos **identificadores de interrupción** de tipo **IRQID** predefinidos. A cada interrupción del núcleo se le puede asociar una **prioridad** dentro de un espacio unificado de prioridades para interrupciones y tareas). Como se puede observar en la Figura 19, la interfaz **iKRNLINT** brinda los servicios **setInterruptPriority**() y **getInterruptPriority**() para establecer y obtener la prioridad de una Interrupción del Núcleo (en la implementación actual este valor puede estar entre 0 y 255). Se ofrece tambien el servicio **getIntrSyncSet**() para obtener el conjunto de objetos de sincronización asociados a cada interrupción. A través de este servicio el resto del núcleo puede inspeccionar, añadir y eliminar objetos de sincronización asociados a una interrupción.

Adicionalmente (ver Figura 19), la interfaz **iKRNLINT** brinda el servicio **initKrnlInt**() que debe ser invocado como parte del arranque del sistema para inicializar el componente de administración de interrupción.

4.3.2 Capa de Abstracción del Hardware 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. Entre sus funciones se encuentran:

- Suministrar un conjunto de líneas de petición de interrupción del sistema independiente de la arquitectura. Estas líneas se denotan por IRQ ("*Interrupt Request*") y van desde IRQ0 hasta IRQn (donde el número n depende del sistema).

- Suministrar la capacidad de establecer dinámicamente las prioridades de cada una de las líneas IRQx, de forma arbitraria e independiente del sistema de prioridades del hardware de interrupciones real de la máquina.

- Suministrar la capacidad de establecer un nivel de interrupción por debajo del cual las interrupciones están inhabilitadas.

La prioridad de cada una de las interrupciones puede situarse en un valor arbitrario (e independiente de la arquitectura) que va desde 0 hasta 255, siendo 255 la mayor prioridad y 1 la menor (ver figura Figura 20). El valor 0 está reservado e indica que el nivel de

interrupción correspondiente está inhabilitado. Sin embargo, el HAL asigna a cada petición de interrupción una prioridad de hardware por defecto cuyo orden de prioridad (no los valores absolutos) coincide con el impuesto por el hardware de interrupción del sistema. El HAL de interrupciones sitúa estas prioridades por defecto en el rango configurable que se especifique mediante los valores **MaxDefault** y **MinDefault**.

Nivel de IRQ		Comentario
255		
		Niveles de prioridad de interrupción asignados explícitamente mediante los servicios *EnableIRQ()* y *SetIRQPriority()*
MaxDefault **MinDefault**		Rango de prioridades en que sitúan las prioridades por defecto de las interrupciones. Aquellas habilitadas con *EnableIRQ(IRQx, 0).*
IRQLevel		Nivel de interrupción por debajo del cual las interrupciones están inhabilitadas. El valor *IRQLevel* puede situarse en cualquier posición.
0		

Figura 20. Niveles de Prioridad del INTHAL

La Figura 21 muestra el diagrama de estados UML ("*statechart*") de las IRQs. En lo que concierne al núcleo, las IRQ pueden estar en dos estados: **capturadas** o **ignoradas**.

- **Interrupciones ignoradas**: Son todas aquellas interrupciones que el núcleo no ha solicitado atender. Si se produjera una interrupción ignorada entonces se considera un error y se invoca al servicio **panic**(). Cuando se inicia el sistema, todas las IRQs se encuentran en este estado.

- **Interrupciones capturadas**: Son aquellas que el núcleo ha solicitado atender de forma explícita mediante la invocación al servicio **enableIrqQ**().

Además, una interrupción capturada puede estar **habilitadas** o **inhabilitadas**.

- **Interrupciones habilitadas**: son aquellas cuyo nivel IRQ está por encima del nivel IRQ actual de la CPU (**IRQLevel**). La activación de las IRQ capturadas y habilitadas provocan la invocación de la rutina **IRQHandler**() del núcleo.

- **Interrupciones inhabilitadas**: son aquellas cuyo nivel IRQ es menor o igual que el nivel IRQ actual de la CPU (**IRQLevel**). Las interrupciones inhabilitadas, independientemente de que hayan sido capturadas, no provocan la invocación de **IRQHandler**().

El servicio **initIrqHardware**() debe ser invocado como parte del arranque del núcleo del sistema y sirve para inicializar el INTHAL. Como se dijo antes, al iniciar el sistema, todas las IRQ están inhabilitadas. Puede utilizarse el servicio **enableIrq(irq, priority**) para habilitar la captura de la petición de interrupción especificada en el parámetro **IRQ**. El parámetro **priority** especifica el nivel de prioridad deseado para la IRQ. Si dicho parámetro se sitúa en 0 se utiliza el nivel de prioridad por defecto. Este servicio puede devolver **ERROR** debido a que la IRQ especificada ya ha sido previamente capturada.

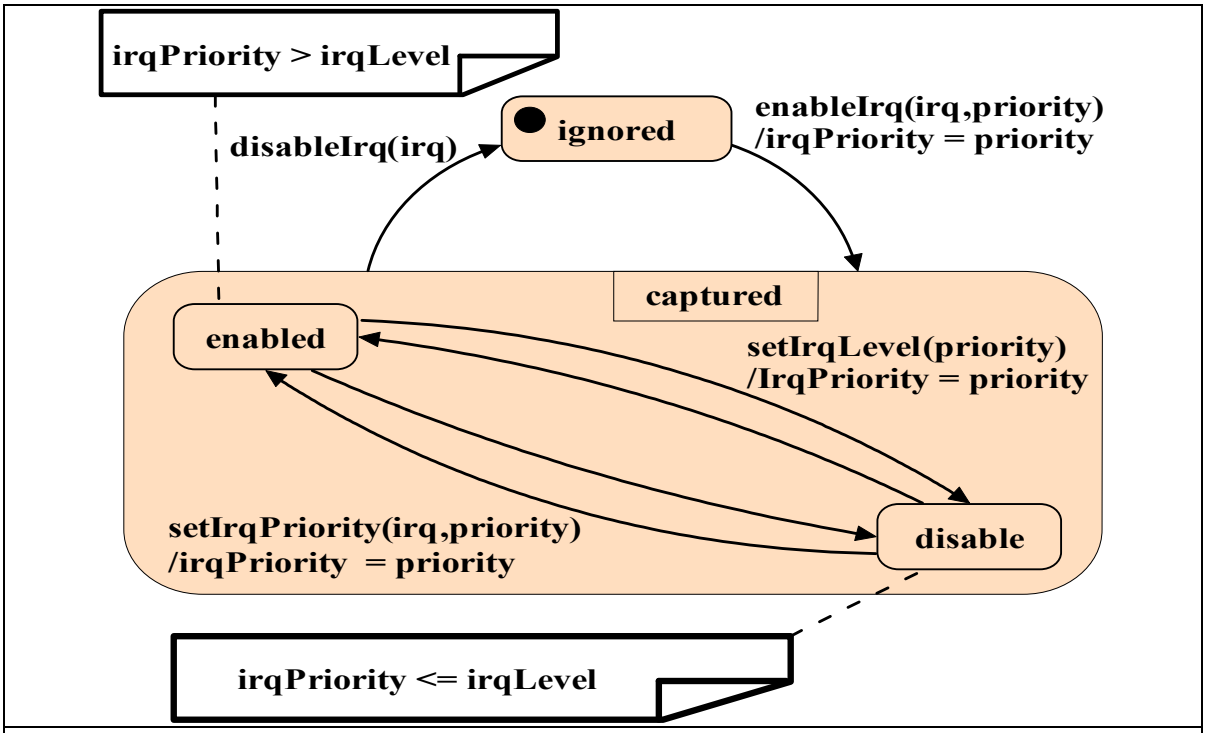


Figura 21. Diagrama de Estado de cada una de las líneas de petición de interrupción del INTHAL (IRQ)

El HAL suministra el servicio **setIrqPriority (IRQ , PRIORITY)** que puede ser utilizado para, una vez habilitada la captura de una IRQ determinada, modificar su nivel de prioridad en cualquier momento.

El servicio **setIrqLevel(PRIORITY)** establece el **nivel de interrupción actual del sistema** por debajo del cual las interrupciones están inhabilitadas. En otras palabras solamente se permiten las interrupciones que posean un nivel de prioridad mayor o igual que el valor pasado como parámetro en **PRIORITY**. Un valor de 0 indica que pueden ocurrir todas las IRQ que hayan sido habilitadas con **enableIrq()**.

Capturada la IRQ si su prioridad es mayor que el nivel de interrupción actual del sistema, cada vez que se produzca, se transfiere el control a **IRQHandler**(**IRQ**) pasándole como parámetro la IRQ correspondiente. El código de este manejador formará parte del núcleo y se ejecuta con todas las IRQ de igual o menor prioridad que la IRQ actual inhabilitadas. Esta rutina puede invocar al servicio **endIrq**(**IRQ**) para indicarle al **INTHAL** que restituya el nivel de interrupción actual del sistema, al valor que tenía antes de producirse la IRQ.

Alternativamente, pudiera situar explícitamente el nivel de interrupción actual invocando al servicio **setIrqLevel**(**PRIORITY**).

4.4 Análisis comparativo entre el Modelo integrado y las alternativas existentes

A manera de conclusión de este capítulo, en esta sección ponemos en perspectiva el modelo integrado de administración de interrupciones y tareas que se propone, contra el resto los esquemas que se utilizan actualmente en los sistemas de tiempo real (ya descritos en la sección 3.4 del capítulo precedente).

Las características de desempeño que son de importancia para el caso de sistemas de tiempo real son: la posibilidad de predecir el comportamiento temporal, la latencia en la respuesta a los eventos externos, la sobrecarga operativa y la posibilidad de control de la sobrecarga de trabajo. Primero se analizará el comportamiento de las estrategias utilizadas en la actualidad de acuerdo a estos parámetros para entonces analizar el comportamiento del modelo integrado aquí propuesto. Este análisis pondrá de manifiesto el verdadero significado de este modelo.

En general, todas las estrategias utilizadas en la actualidad se pueden agrupar en dos grandes grupos:

1. **Modelo tradicional de ISRs y Tareas**: En estos esquemas los eventos externos se manejan mediante interrupciones utilizando algún tipo de variante de los esquemas de tratamiento de interrupción utilizados en los sistemas operativos tradicionales (sistemas operativos de tiempo compartidos, y sistemas operativos de red – ver sección 3.4.3) quizás acompañado con la implementación de algún esquema de manejo de sobrecarga de interrupciones (ver sección 3.4.6) y complementado con alguna extensión a las ecuaciones de análisis de factibilidad de planificación para tener en cuanta el costo de las ISRs (ver sección 3.4.5).

2. **Eliminación de las Interrupciones**: Bajo este enfoque, el tratamiento por interrupción es considerado incompatible con el tiempo real (ver sección 3.4.4) debido a que posibilita la introducción de transferencia del control a las ISRs en cualquier punto del flujo de control, con lo cual éstas se convierten en entidades no planificables deteriorando la predecibilidad del sistema (ver sección 2.2).

4.4.1 Predecibilidad

Sin duda alguna, la posibilidad de predecir el comportamiento temporal del sistema es el requerimiento más importante que tiene que satisfacer cualquier sistema operativo de tiempo real.

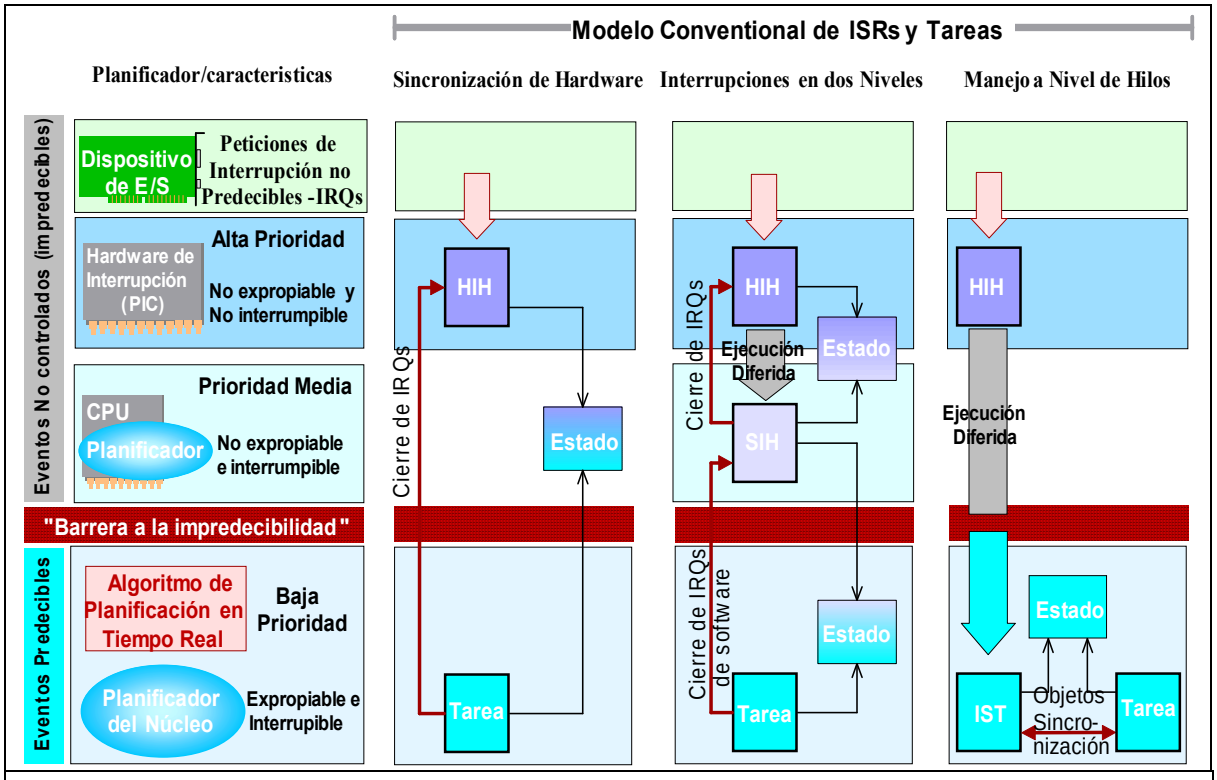


Figura 22. Modelo "tradicional" de ISRs y Tareas: Variaciones fundamentales

La Figura 22 muestra un diagrama con los diferentes elementos de hardware y de software involucrados en el manejo de las interrupciones y donde se resumen las diferentes variaciones de lo que hemos dado en denominar "Modelo Tradicional de Manejo de Interrupciones". Al observar el esquema se pone de manifiesto como, en todos los casos, permanecen (en mayor o menor medida) distintos tipos de ISRs (ya sean manejadores de interrupciones de hardware – denotadas por HIH – o de interrupciones de software – denotadas por SIH) que están fuera del control del planificador de tiempo real y en consecuencia constituyen entidades no planificables. En otras palabras, en todos estos esquemas el mecanismo de interrupciones le concede a un dispositivo de E/S externo que está fuera del control del planificador de tiempo real el poder para asignar arbitrariamente el recurso más importante en un sistema de tiempo real, el tiempo de CPU. Esto da como resultado que se comprometa significativamente la capacidad de predecir el comportamiento temporal del sistema.

La alternativa de eliminación de las interrupciones (con la excepción de la interrupción del reloj del sistema) es una solución radical que le quita a los dispositivos de E/S externos la posibilidad de comunicar expeditamente los eventos externos y con ello la capacidad de ejercer algún control sobre la asignación del tiempo de la CPU. Bajo este esquema, todos los eventos (o señales) de control de la planificación son producidos única y exclusivamente por el transcurso del tiempo (reloj del sistema). Al quedar eliminadas las ISRs asociadas a dispositivos externos, desaparecen las entidades no planificables del sistema de E/S y se hace posible predecir de forma confiable el comportamiento temporal del sistema. Al eliminar las interrupciones, todas las operaciones de E/S se llevan a cabo mediante encuesta ("*polling*") utilizando tareas periódicas dedicadas (por ejemplo las

denominadas tareas detonadoras – "*trigger task*" en la arquitectura disparada por tiempo - "*time-triggered architecture*" [47]).

4.4.2 Sobrecarga operativa

Aunque en los sistemas de tiempo real es aceptable hacer el compromiso de aceptar una mayor sobrecarga operativa para lograr el objetivo primordial de la predecibilidad temporal, la mayoría de las aplicaciones se beneficiarán de forma importante con una disminución de la sobrecarga del sistema. Aunque ambos enfoques implican una sobrecarga operativa, ésta difiere en cuanto a su magnitud, a los factores que la determinan y en el comportamiento con respecto a la carga de trabajo.

Los enfoques basados en interrupciones incurren en una sobrecarga de conmutación de contexto debido a la necesidad de salvar y restaurar el estado de la actividad en ejecución en el momento de la interrupción. En los procesadores actuales esta sobrecarga puede ser significativa debido al empleo de largas canalizaciones ("*pipelining*") y la ejecución especulativa; así como, debido a la necesidad potencial de contaminar/limpiar ("*pollute/flush*") los caches alrededor de la invocación de los manejadores de interrupción.

En el caso de la encuesta, la sobrecarga de conmutación de contexto es mínima o puede evitarse ya que el sistema puede encuestar al dispositivo una vez que está ya en el contexto correcto (por ejemplo, manejando la interrupción del reloj del sistema, o una trampa en el caso que se requiera estar en modo supervisor). Sin embargo, en el caso de la encuesta, se incurre en un costo debido a la asincronía existente entre la emisión de la encuesta y la ocurrencia del evento, de modo que es posible que la encuesta se emita en un instante en que no hay evento disponible (encuesta baldía). En el caso de las interrupciones este costo no existe ya que las mismas se producen única y exclusivamente ante la ocurrencia del evento en cuestión (bajo suposiciones de ausencia de defectos de hardware).

Una consecuencia de esta diferencia en el origen de la sobrecarga operativa ("*overload*") es que la sobrecarga real de ambos enfoques está en función de la frecuencia y la distribución de la ocurrencia de los eventos:

- En el caso de las interrupciones, el costo de la conmutación de contexto se añade a cada uno de los eventos procesados por lo que la sobrecarga crece (presumiblemente de forma lineal) con el aumento de la frecuencia de ocurrencia de los eventos. El tratamiento por interrupción puede ser particularmente costoso en situaciones de elevada frecuencia de los eventos, cuando la CPU es interrumpida frecuentemente (con toda la sobrecarga asociada) para procesar sólo un evento a la vez.

- En el caso de la encuesta, la sobrecarga operativa es menor cuando el tráfico es alto, pero la sobrecarga operativa por paquetes crece cuando disminuye la tasa de llegada de los paquetes ya que no son necesarias algunas encuestas.

4.4.3 Latencia de atención a los eventos

El enfoque de atención a los eventos externos mediante encuesta tiene como inconveniente respecto a los esquemas basados en interrupciones, que por regla general se sacrifica la sensibilidad del sistema o rapidez de respuesta a los eventos externos. Ello se debe a que en el servicio a los eventos externos mediante interrupciones se hace cuando el dispositivo lo requiere (por ejemplo cuando llega un nuevo paquete de datos por el dispositivo de comunicaciones); mientras que en el tratamiento mediante encuesta, se hace cuando el sistema lo desea no cuando el dispositivo lo requiere.

Con la encuesta el tiempo de respuesta a los eventos externos está determinado por la frecuencia de muestreo de la tarea de encuesta. Como consecuencia, sólo es posible reducir la latencia a valores razonablemente bajos aumentando la frecuencia de las encuestas (período de muestreo) pero ello sólo se consigue a costa de una sobrecarga operativa significativa.

4.4.4 Comportamiento en presencia de sobrecargas

En determinadas situaciones es posible que se presente (de forma temporal o permanente) una situación en la cual se origina una cantidad o frecuencia de los eventos superior a aquella que las capacidades del sistema de cómputo es capaz de manejar. Estas situaciones de sobrecarga pueden ocurrir debido a la presencia de cargas no anticipadas en el diseño del sistema o debido a escenarios de fallos. En estas situaciones, para poder mantener el sistema operando de forma estable, este debería ser capaz de limitar la cantidad de eventos recibidos de modo que el tiempo de procesamiento se mantenga dentro de las restricciones de la capacidad de cómputo del sistema.

En el caso de la encuesta, la tasa máxima a la cual pueden llegar los eventos está determinada única y exclusivamente por la tasa a la cual el software realiza la encuesta de los dispositivos. Sin embargo, en el caso del tratamiento por interrupción, la tasa a la cual se generan las interrupciones puede estar fuera del control del software del sistema y pudiera no ser acotada. Como consecuencia, en el caso de la encuesta el sistema puede controlar las tasas de llegada de los eventos; mientras que en el caso del tratamiento por interrupciones las posibilidades de control son muy bajas.

Dado que en un sistema basado en interrupciones se dificulta el control de la tasa de entrada y debido a que las interrupciones poseen mayor prioridad que cualquier tarea ejecutándose en el procesador, casi invariablemente, bajo cargas pesadas, estos sistemas se salen de control porque la mayoría del tiempo de CPU se gasta en manejar interrupciones con muy pocos o ningún ciclo de CPU que quede para el procesamiento de las tareas de usuario. Adicionalmente, si los eventos recibidos no se procesan hasta terminar (como ocurre en el caso del procesamiento en múltiples niveles), es probable que el sistema incluso no pueda llevar a cabo ningún trabajo útil ya que no encontrará tiempo para completar el procesamiento que postergó fuera de los manejadores de interrupción.

Esta característica es fatal para el caso de sistemas de tiempo real que tienen que realizar sus funciones dentro de determinados plazos de tiempo estrictos. En estos sistemas una

situación de sobrecarga de interrupciones puede poner en peligro la puntualidad de la aplicación. En el escenario de peor caso, pueden incumplirse plazos críticos.

4.4.5 Contraste entre el Modelo Integrado y los enfoques precedentes

La Figura 23 muestra el contraste entre el modelo integrado aquí propuesto y el modelo tradicional de manejo de ISRs y tareas en sus distintas alternativas. La misma permite apreciar claramente la diferencia cualitativa existente entre el modelo integrado y el modelo tradicional y que está dada porque bajo el modelo integrado el subsistema de administración de interrupciones está libre de entidades no planificables. Ello se ha logrado gracias a que a diferencia de los esquemas anteriores, en los cuales las IRQ (y las ISRs correspondientes) no estaban bajo el control del planificador de tiempo real, en el modelo integrado se ha conseguido planificar a las mismas IRQs, transformando a su manejador (el LLIH que se denota como HIH en la figura) en una entidad planificable. En el esquema se destaca esta característica como la contribución fundamental del modelo integrado.

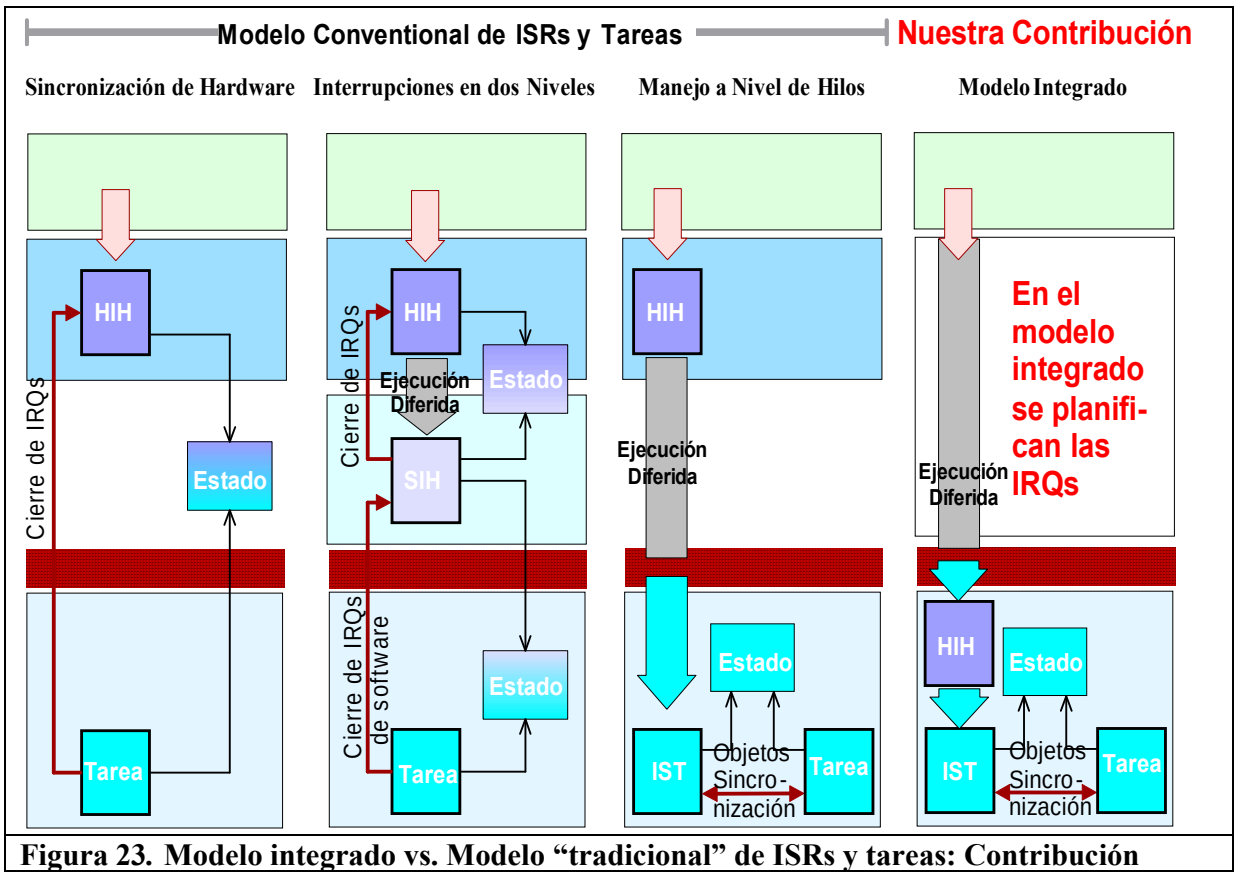


Figura 23. Modelo integrado vs. Modelo "tradicional" de ISRs y tareas: Contribución

La tabla de la Figura 24 hace el contraste entre el modelo integrado y las alternativas existentes de manejo tradicional de interrupciones y de eliminación de las interrupciones (o tratamiento de los eventos externos por encuesta). En la misma se pone de manifiesto como el esquema integrado logra conjuntar las ventajas de ambos enfoques. En otras palabras, con el modelo integrado se ha logrado combinar las características de baja latencia y baja sobrecarga operativa (propias del tratamiento por interrupción) con las características de

predecibilidad temporal y posibilidad de control de la sobrecarga de trabajo (propia del esquema de tratamiento por encuesta).

De las características deseables del modelo integrado mostradas en la Figura 24, las tres primeras no merecen mayor explicación; sin embargo, vale la pena comentar sobre la posibilidad de control de sobrecarga. En esencia esto es una consecuencia derivada de la posibilidad única del modelo integrado de planificar las mismas peticiones de interrupción (IRQs). Estro ofrece la posibilidad única de hacer pasar toda la carga del sistema por cualquier algoritmo de planificación de tiempo real con capacidad de control de sobrecarga.

Por ejemplo, en un sistema basado en un planificador con expropiación por prioridades fijas (tal como sucede en el caso de una asignación monótona en tasa) pudiera utilizarse el algoritmo de servidor esporádico [87]. Bajo este esquema las HAT del sistema harían las veces de servidor esporádico a los que se les asignaría una prioridad primaria o normal, una prioridad secundaria o de fondo (inferior a la prioridad normal), un presupuesto de tiempo de CPU y un intervalo de reabastecimiento. Juntos, el intervalo de reabastecimiento y el presupuesto de ejecución definen la máxima utilización del procesador que puede ser empleada en la atención a las peticiones de servicio del dispositivo externo que genera interrupciones por la IRQ asociada a dicha HAT a su prioridad normal.

Característica	Modelo "Tradicional" de ISRs y Tareas	Modelo Integrado de IRQs y Tareas	Eliminación de las Interrupciones
Predecibilidad temporal del sistema	Deteriorada debido a la presencia de ISRs (entidades no planificables)	**Reforzada debido a la Eliminación de las ISRs (entidades no planificables)**	**Reforzada debido a la eliminación de las ISRs (entidades no planificables)**
Latencia de respuesta a los eventos	**Baja Latencia (alta sensibilidad). Dada por la latencia de interrupción**	**Baja Latencia (alta sensibilidad). Dada por la latencia de interrupción**	No es óptima (baja sensibilidad). Determinada por el período de muestreo
Sobrecarga operativa	**Baja: Dada por la conmutación de contexto. Proporcional a la carga de trabajo**	**Baja: Dada por la conmutación de contexto. Proporcional a la carga de trabajo**	Apreciable, dada por la tarea de encuesta: WCET[15] del Muestreo / Período de Muestreo
Control de la Sobrecarga de trabajo	Malo: Está fuera del control del sistema	**Bueno: bajo el control del planificador**	**Bueno: El sistema ignora cualquier sobrecarga**

Figura 24. Comparación entre las características del Modelo integrado y las alternativas existentes

Si por alguna razón de sobrecarga de trabajo, alguna HAT sobrepasase el tiempo de procesamiento que le ha sido asignado en su presupuesto, la política de planificación esporádica le disminuye dinámicamente su prioridad a su prioridad de fondo. Cuando, a ese

[15] WCET son las siglas en ingles del término Tiempo de Ejecución en el Peor Caso ("*Worst Case Execution Time*")

nivel de prioridad de fondo, la HAT sea la tarea de más prioridad del sistema podrá seguir consumiendo tiempo de procesamiento, pero ello lo hará sin interferir con el resto de las tareas de mayor prioridad del sistema. De este modo, se consigue todavía utilizar el tiempo no utilizado por las tareas de mayor prioridad para dar servicio a la sobrecarga de trabajo. Transcurrido el período de reabastecimiento, se vuelve a renovar el presupuesto de tiempo y se restituye la HAT a su prioridad normal.

Como puede apreciarse, la utilización de este mecanismo de interrupciones integrado elimina la necesidad de utilizar mecanismos de encuesta (o incluso peor de espera ocupada) para las operaciones de E/S, ya que permite el tratamiento de los eventos externos por interrupción sin sacrificar en lo absoluto el determinismo temporal del sistema.

Si a lo anterior se adiciona el requerimiento de que los sistemas de tiempo real tienen que ser confiables. Entonces, la posibilidad de realizar sistemas confiables, por lo general va a ser inversamente proporcional al número de abstracciones que suministre el núcleo y la complejidad de la solución de las interacciones entre estas. En consecuencia, en núcleos destinados a la realización de sistemas en los que la respuesta en tiempo y la confiabilidad sean factores determinantes, existe poca justificación para mantener ambas actividades (ISRs y tareas) como abstracciones separadas.

Por tanto, del análisis anterior se puede concluir que el esquema integrado aquí propuesto es más adecuado que las alternativas existentes para el caso de sistemas de tiempo real.

4.5 Resumen

En este capítulo hemos presentado el modelo integrado de tratamiento de interrupciones y tareas. Con la ayuda de la teoría de planificación de tiempo real y el empleo de criterios de ingeniería hemos justificado las ventajas de este modelo para el caso de aplicaciones de tiempo real confiables. Adicionalmente, se presentó el diseño de un subsistema de administración de interrupciones de bajo nivel que puede ser utilizado en cualquier sistema operativo que quiera soportar este modelo. El diseño de este subsistema puede ser implementado de diversas formas. Una de ellas pudiera ser mediante el diseño de un controlador de interrupciones a la medida utilizando FPGAs, otra pudiera ser mediante emulación en software sobre arquitecturas de interrupciones convencionales. A manera de concluir se analizaron las características del modelo integrado en contraste con las alternativas existentes para el tratamiento de eventos externos en sistemas de tiempo de real. Este análisis puso de manifiesto las bondades del modelo propuesto para el caso de sistemas de tiempo real.

5 Realización sobre un Hardware PC compatible

El diseño descrito en el capítulo anterior (sección 4.3) permite que el núcleo del sistema sea completamente independiente del hardware de interrupciones. Pueden existir varios módulos alternativos que implementen la interfaz **iINTHAL** cada uno para un hardware de interrupción concreto. En el caso ideal la funcionalidad de este subsistema debería ser soportada directamente por el hardware de interrupciones del sistema. Sin embargo, la generalidad del hardware de interrupciones actuales (incluyendo el de las PC convencionales) no lo soportan de forma adecuada.

Una alternativa sería la implementación de un controlador de interrupciones a la medida utilizando FPGAs; sin embargo, esta opción podría añadir costos al sistema o incluso pudiera no ser posible en todas las arquitecturas incluyendo las PC convencionales.

En este capítulo se describen los detalles de la implementación del diseño antes propuesto y del modelo completamente integrado sobre una arquitectura de interrupciones estándar de una PC. Esta implementación nos ha permitido validar la factibilidad del modelo propuesto incluso en situaciones en las cuales no exista soporte directo del hardware. Sin embargo, con el propósito de comprender esta implementación, se hace necesario primero una comprensión del hardware de interrupciones de estos sistemas.

5.1 Hardware de interrupciones en sistemas PC compatibles

El **Controlador de Interrupciones Programable**, PIC ("*Programmable Interrupt Controle*r") 8259A es el encargado de recibir las solicitudes de interrupciones provenientes del hardware y transmitirlas a la CPU. La CPU entonces activa el manejador de interrupción correspondiente.

La responsabilidad del PIC es poner en cola las solicitudes de interrupción provenientes de los diferentes dispositivos periféricos para enviarlas de uno en uno a la CPU en un orden de prioridad predeterminado. El PIC puede gestionar hasta ocho líneas de petición de interrupciones que están numeradas del 0 al 7. Estos números también representan al mismo tiempo la prioridad de la interrupción: IRQ0 tiene la máxima prioridad, IRQ1 la siguiente, y así hasta IRQ7 que tiene la mínima prioridad. El diseño de la tarjeta del sistema es el que decide qué dispositivos que originan interrupción están unidos a cada una de las líneas de petición de interrupción del PIC.

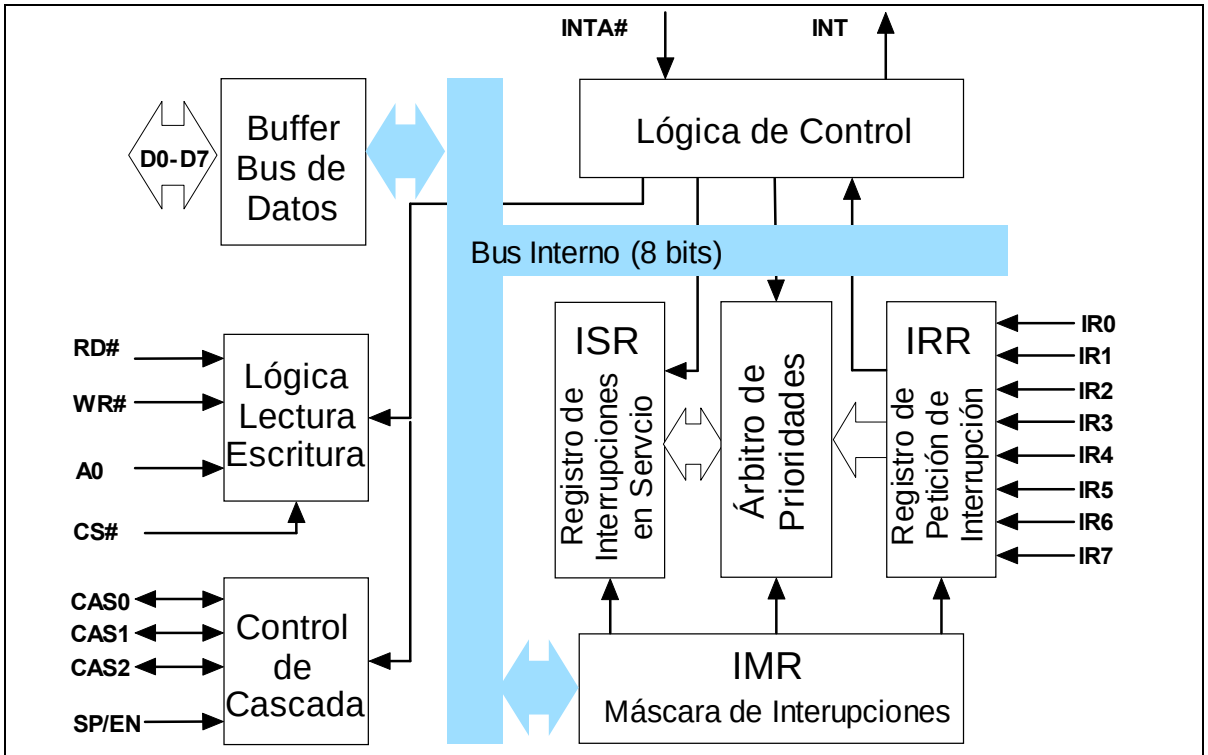


Figura 25. Diagrama de Bloques del 8259

La Figura 25 muestra un diagrama con la estructura interna del controlador de interrupciones 8259A. Este posee varios registros internos que afectan su operación. Los tres registros más importantes para esta aplicación son el Registro de Petición de Interrupción IRR ("*Interrupt Request Register*"), el Registro de Interrupciones en Servicio, ISR ("*In Service Register*") y el Registro de Máscara de Interrupción, IMR ("*Interrupt Mask Register*"). Los bits en el IRR indican cuando una de las ocho líneas de petición de interrupción está activada (cuando un dispositivo de hardware está solicitando servicio"), independientemente del estado de los demás registros.

Situando un bit en el IMR, se puede inhibir el servicio de una IRQ. Los bits en el IRR todavía reflejarán la presencia de una petición de interrupción pendiente, pero si la interrupción está enmascarada, el 8259A no responderá hasta que se limpie la máscara.

Los bits en el ISR se sitúan luego de que el 8259 ha activado la línea de petición de interrupción del procesador (INT) y el procesador reconoce la interrupción (INTA#). Este protocolo de reconocimiento de la interrupción por parte del procesador ocurre completamente bajo el control del hardware. El único control que puede ejerce el software sobre este es la suspensión del reconocimiento de todas las interrupciones limpiando la bandera de interrupciones (IF) del procesador (mediante la instrucción CLI). El IRR le permite al 8259 llevar la cuenta de cuales interrupciones están en servicio de forma que cuando ocurra una IRQ, esta no provoque una interrupción en el procesador mientras estén en servicio en la CPU interrupciones de mayor prioridad.

La forma en que se limpian los bits en el ISR (y en consecuencia se habilitan las interrupciones de mayor o igual prioridad) depende del modo de fin de interrupción con que se haya programado al 8259:

- **Modo de EOI normal (o explícito) o simplemente EOI**: En este modo los bit se limpian cuando el software (de la ISR) le envía un comando de Fin de Interrupción o EOI ("*End Of Interrupt*") al 8259A su vez, el PIC soporta dos tipos de comandos EOI: el EOI específico y el no específico. El EOI no específico da acuse de recibo a la última IRQ que ocurrió (la cual, por definición, es la IRQ de más alta prioridad que todavía no ha sido finalizada). El EOI específico reconoce, una IRQ particular.

- **Modo EOI automático o AEOI ("*Automatic EOI*")**: En este modo el bit del ISR correspondiente a la IRQ que se solicita se limpia automáticamente cuando la CPU reconoce la petición de interrupción. Inmediatamente después, el 8259A puede enviar otra petición de igual o menor nivel al procesador interrumpiendo a la anterior.

Todos los sistemas operativos de que tenemos conocimiento utilizan el modo EOI normal (mediante el comando de EOI no específico). Ello se debe a que es el que permite que el 8259 gestione las prioridades de las IRQ de forma adecuada.

5.1.1 Interrupciones disparadas por nivel vs. Interrupciones disparadas por Flanco

El PIC 8259A soporta dos modos de solicitudes de interrupción (ver Figura 26): IRQ disparada por flanco ("*edge-triggered*") e IRQ sensible al nivel ("*level triggered*").

En el modo sensible al flanco la activación de la interrupción se consigue con la transición de nivel de voltaje en la línea (por ejemplo el cambio de nivel bajo a nivel alto). El 8259A soporta entradas de interrupción disparadas por flanco positivo. Esto es, cuando se produce una transición del 0 lógico al 1 lógico en una línea IRx, esta se asegura ("*latch*") en el bit correspondiente del IRR (bit x). Si la entrada IR permanece en nivel lógico 1 incluso después de que finalice la rutina de servicio, la interrupción no es vuelta a emitir, sino que queda inhabilitada. Para poder ser reconocida de nuevo, la entrada tiene que primero retornar al nivel lógico 0 y entonces ser puesta nuevamente en 1. El problema con las interrupciones sensibles al flanco es que no pueden ser compartidas de forma confiable entre múltiples dispositivos periféricos.

En el modo sensible al nivel la activación de la interrupción se consigue estableciendo cierto voltaje estático (por ejemplo nivel alto) en la línea IRQ correspondiente. En este modo la solicitud de interrupción tiene que ser retirada antes de que la rutina de servicio de interrupción termine. De lo contrario, la interrupción volverá a ser solicitada por segunda vez y la rutina de servicio ejecutada nuevamente. Esto implica que la ISR no debe enviar el EOI al PIC antes de que o el manejador haya provocado que el dispositivo retire la petición de la línea IRQ, o la IRQ como tal haya sido enmascarada en el IMR, porque de lo contrario la IRQ ocurrirá nuevamente de forma instantánea. Si la entrada IR regresa a nivel lógico 0 antes de que sea reconocida por la CPU, la solicitud de servicio será ignorada.

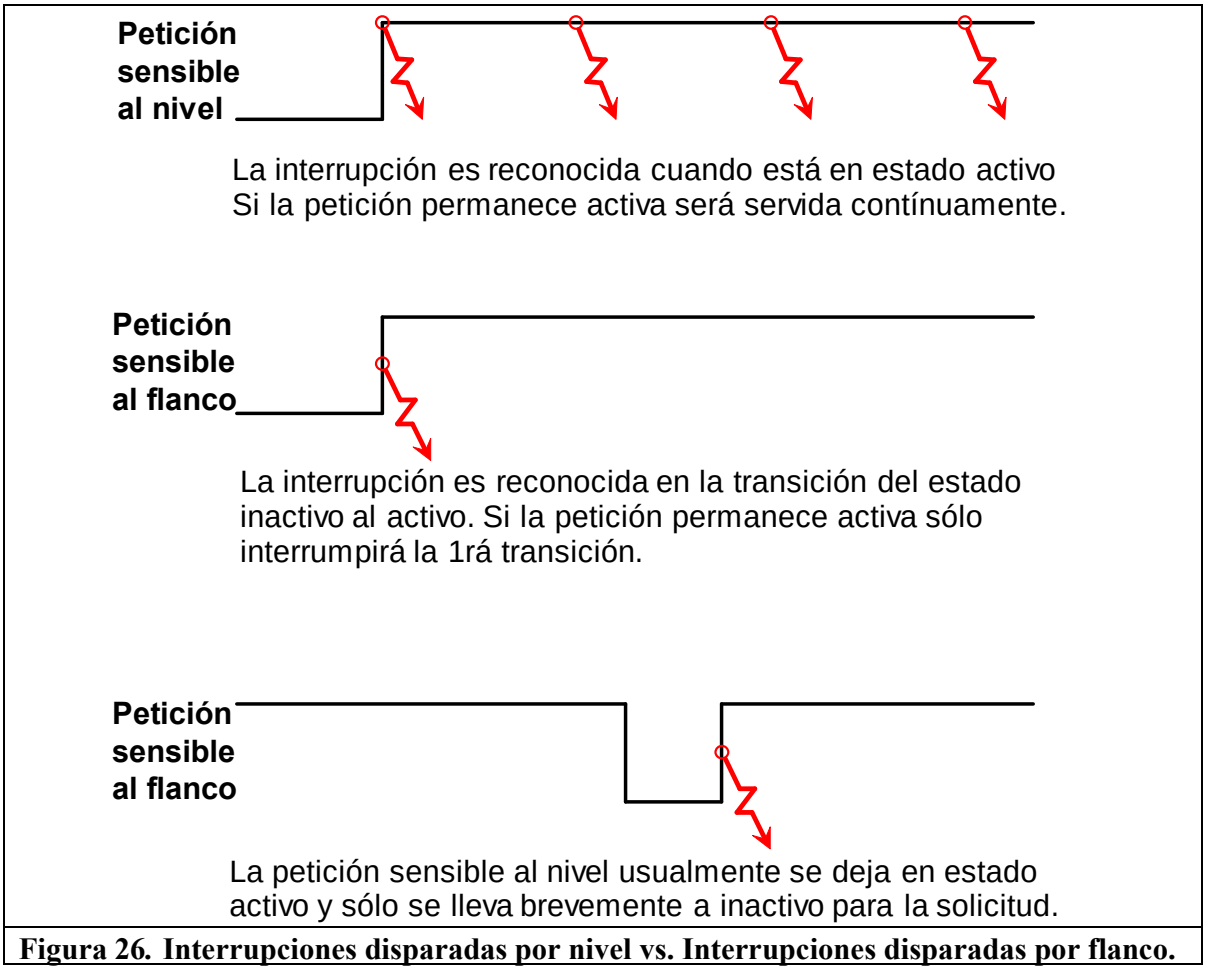


Figura 26. Interrupciones disparadas por nivel vs. Interrupciones disparadas por flanco.

Debido a que el bus ISA no soporta interrupciones disparadas por nivel, el modo disparado por nivel no debe ser utilizado para interrupciones conectadas a dispositivos ISA. Esto quiere decir que en los sistemas PC/XT, PC/AT, el 8259A tiene que ser programado en modo disparado por flanco. En los sistemas EISA, PCI y posteriores existe un Registro de Control de Flanco/Nivel o ELCR ("*Edge/Level Control Registers*") que controla el modo para cada una de las líneas IRQ. En lo que resta del capítulo supondremos que se utiliza el modo de disparo por flanco.

5.1.2 Secuencia de petición/reconocimiento de interrupciones del hardware.

La Figura 27 muestra la secuencia de eventos que se suceden cuando se lleva a cabo una petición de interrupción por un periférico. Esta consiste de los siguientes acciones:

1. El dispositivo realiza la petición de interrupción con el flanco de subida de la señal de solicitud IRQ0-IRQ15 correspondiente las cuales llegan a las líneas de petición de interrupción IR0-IR7. Esto provoca que se active el bit correspondiente en el IRR del 8259A. La operación del IRR no es afectada por el IMR

   Nota: Esta señal debe mantenerse activa (nivel alto) hasta tanto el PIC 8259A reciba la señal INTA proveniente de la CPU. Si la IRQ no permanece activa

hasta tanto llegue el INTA al 8259A, este generará un nivel de interrupción IRQ7 independientemente del nivel de prioridad de la interrupción solicitada.

2. El 8259A detecta esta señal evalúa las prioridades y si no hay una IRQ de mayor prioridad en servicio y dicha petición no está enmascarada en el IMR responde con una señal INT al procesador.

3. La CPU recibe la señal INT por su pin INTR y si la bandera IF está activa (interrupciones habilitadas) inicia un ciclo de reconocimiento de interrupción consistente de dos pulsos –INTA enviados al 8259A:

   - **1er pulso -INTA**: congela todos los requerimientos y prioridades internamente para resolver, entre las solicitudes (bits del registro IRR), cuál es la de mayor prioridad cuyo bit será activado en el registro ISR indicando que se está sirviendo y su correspondiente bit en el IRR se desactivará para, cuando se termine de atender, evitar que vuelva a causar interrupción (si fuese por flanco).

   - **2do pulso –INTA**: solicita un byte de código de tipo de interrupción el formato del vector de interrupción es causando que el 8259A ponga un apuntador de 8 bit en el bus de datos. La CPU lee este apuntador como el número del manejador de interrupción a invocar.

4. En el modo AEOI el bit ISR se limpia automáticamente al final del segundo pulso –INTA. De otro modo la CPU debe emitir un comando EOI al 8259A cuando se esté ejecutando el manejador de interrupción para limpiar manualmente el bit del ISR.

Obsérvese que si se inhabilitan las interrupciones en el procesador (IF = 0), el controlador de interrupciones continuará evaluando sus entradas, llevando la pista de la solicitud de interrupción con mayor prioridad pendiente. Cuando el procesador sea capaz de aceptar la interrupción (IF = 1), el controlador de interrupciones primero emitirá la interrupción con mayor prioridad.

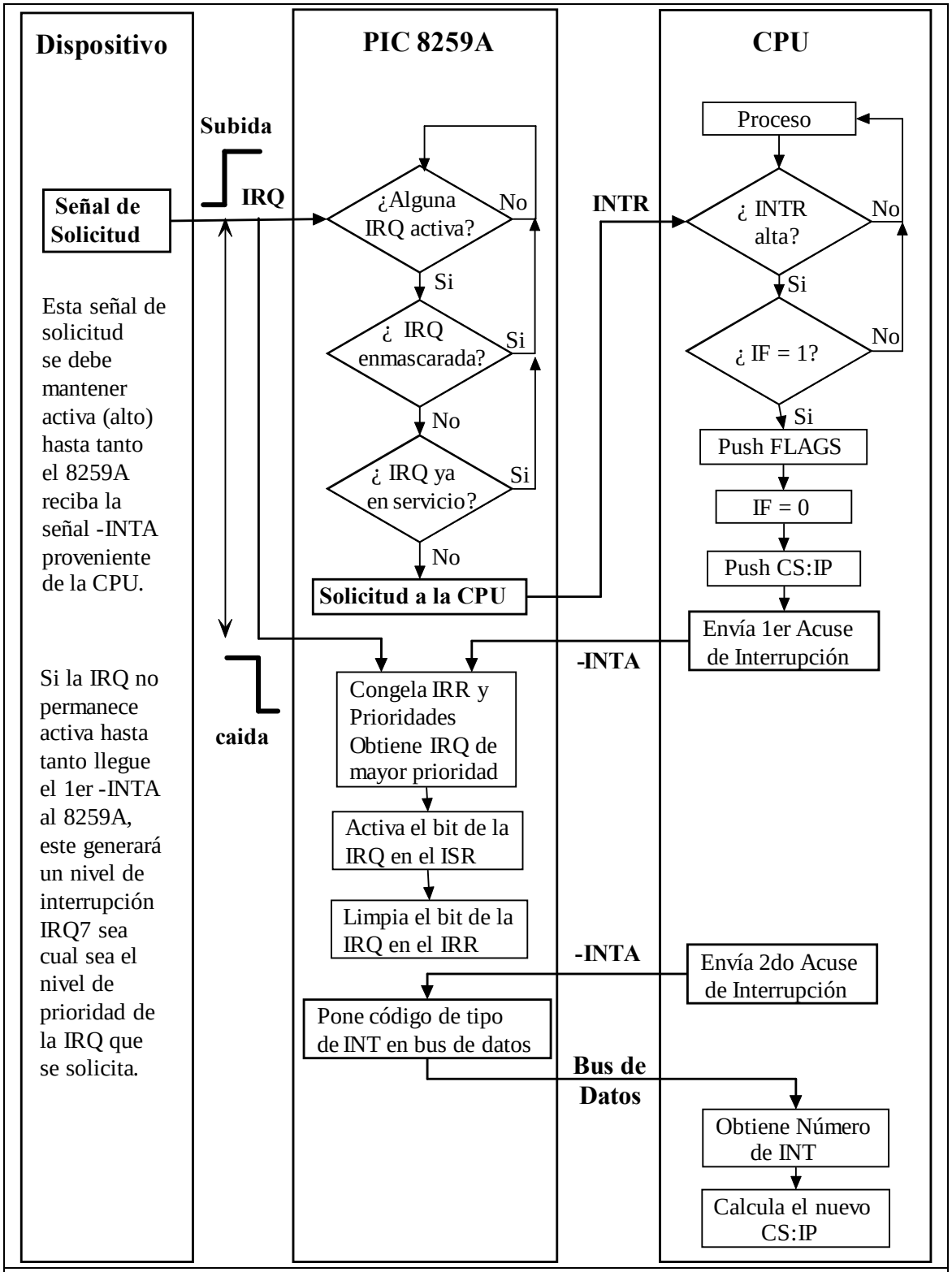


Figura 27. Secuencia de Petición y Reconocimiento de Interrupciones del Hardware de interrupciones de los PC.

5.1.3 Configuración del hardware de Interrupciones en los sistemas PC compatibles

El PIC ofrece la posibilidad de configurarse en cascada para atender más de ocho interrupciones. Esta posibilidad se utilizó desde los tiempos de los PC-AT cuando se instalaron dos PIC y con ello podían causar interrupciones de hardware 16 dispositivos diferentes. La tabla que aparece en la Figura 28 muestra la configuración de los PIC en los sistemas PC AT y compatibles. Al temporizador, por ejemplo, se le concede una prioridad mayor que a la impresora e incluso que al teclado

	Línea IRQ		Nro de INT (Hex)	Descripción
	8259A-1	8259A-2		
	IRQ0		08H	Temporizador del sistema 818.2 veces/segundo)
	IRQ1		09H	Solicitud d Servicio del teclado
	IRQ2		0AH	Línea INT del 8259A- esclavo (8259A-2)
		IRQ8	70H	Reloj de tiempo real
		IRQ9	71H	Redirigido por software a la INT 0AH (IRQ2)
		IRQ10	72H	Reservado
		IRQ11	73H	Reservado
		IRQ12	74H	AT: Reservado; PS/2: Ratón PS/2
		IRQ13	75H	Coprocesador numérico (80287/387)
		IRQ14	76H	Controlador de Disco Duro AT
		IRQ15	77H	Reservado
	IRQ3		0BH	Puerto serie #2 (COM2)
	IRQ4		0CH	Puerto serie #1 (COM1)
	IRQ5		0DH	Puerto paralelo de impresora #2 (LPT2)
	IRQ6		0EH	Disco Flexible
	IRQ7		0FH	Puerto paralelo de impresora #1 (LPT1)

Figura 28. Tabla resumen de Niveles de Interrupción en la IBM PC AT

El efecto de cascada se logra conectando la línea INT del segundo 8259A, el cual funciona como esclavo, a la línea IRQ2 del primer 8259A el cual hace las funciones de 8259A maestro. Cuando un dispositivo conectado a una de las líneas de solicitud de interrupción IRQx del esclavo solicita interrupción la línea INT del esclavo va a nivel alto y causa una activación de la IRQ2 del 8259A maestro, la cual a su vez causa que se active la línea INTR de solicitud de interrupción al microprocesador 80286 en este caso. Ver Figura 29.

El microprocesador ignora la presencia del segundo 8259A y simplemente genera una señal de reconocimiento de interrupción al recibir la solicitud de interrupción del 8259A maestro. Esta señal inicializa ambos 8259A así como causa que el 8259A maestro, el cual sí conoce que en IRQ2 está conectado otro 8259A, le ceda el control al esclavo el cual con el segundo pulso INTA situará el byte de código de tipo de interrupción en el bus, con lo que completará la solicitud de interrupción.

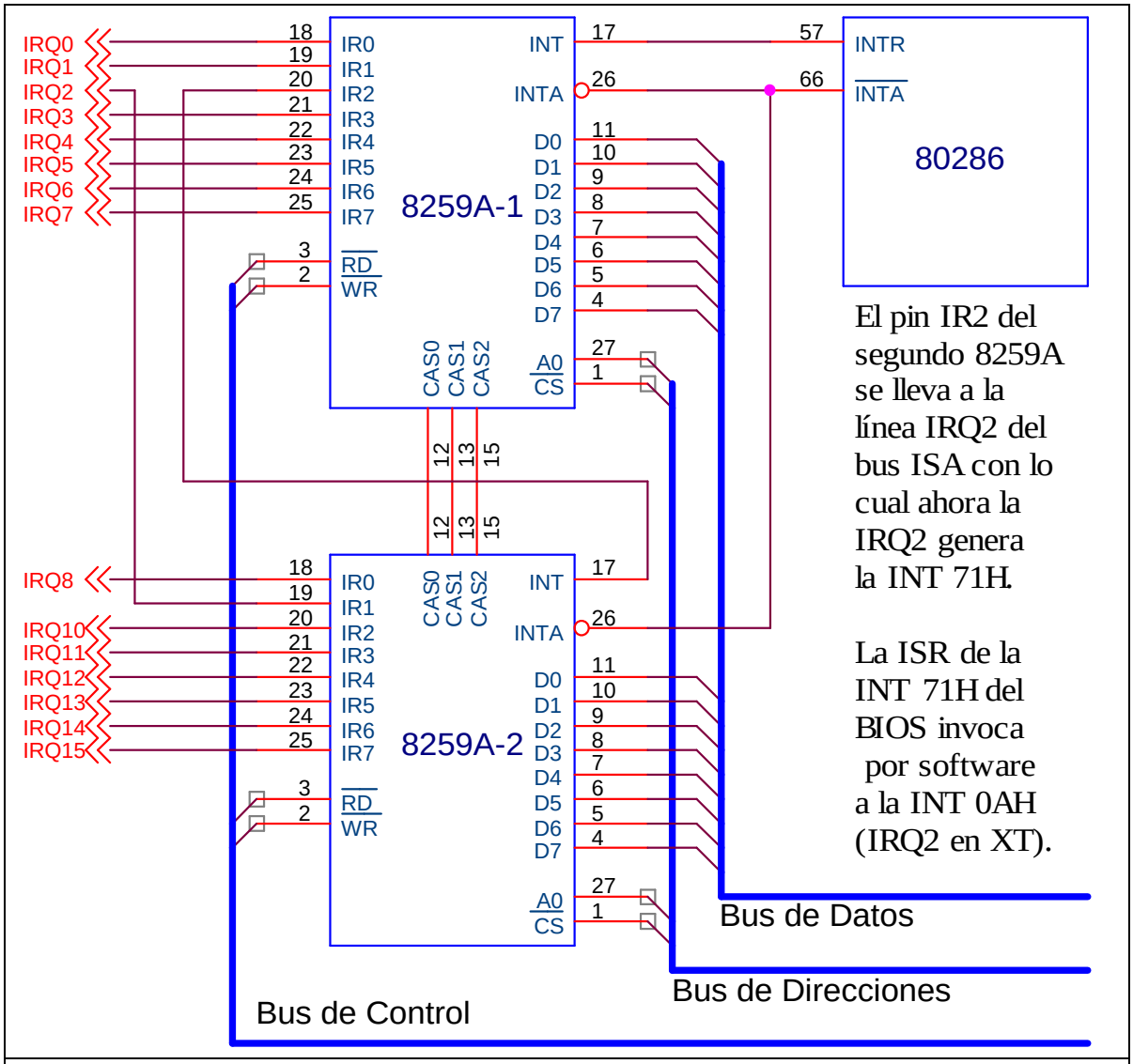


Figura 29. Esquema de conexión en cascada de a PC AT

El segundo 8259A de la IBM PC/AT se programa de forma que las ocho líneas adicionales se mapean en las interrupciones 70H a la 77H (ver Figura 28). Debido a que las 8 líneas adicionales se conectan a la línea IRQ2 del 8259A amo, las mismas son más prioritarias que las líneas IRQ3 a IRQ7 de este último.

El problema aquí es cómo lograr en el nuevo diseño compatibilidad con el diseño anterior. Para ello la 2da línea (IRQ1) del segundo 8259A se conectó a la línea IRQ2 del bus de expansión con lo cual entonces esta generará una INT 71H en lugar de una INT 0AH como en el diseño del IBM PC/XT. Para completar el diseño la ROM BIOS realiza una redirección por software de la INT 71H a la INT 0AH con lo cual se logra que los dispositivos conectados a IRQ2 continúen siendo atendidos por la INT 0AH cuando solicitan interrupción.

Obsérvese que durante la secuencia INTA los bits correspondientes de los registros ISR de ambos 8259A se activan por lo que para completar el servicio de interrupción deben enviarse dos comandos EOI: uno para el 8259A esclavo y otro para el 8259A maestro.

Este diseño de 16 niveles priorizados de interrupciones se ha convertido en el estándar de los sistemas basados en microprocesadores 80286 y superiores.

5.2 Lógica subyacente en la realización del HAL para sistemas compatibles IBM

En el caso de sistemas compatibles IBM el HAL de interrupciones suministra peticiones de interrupción que van desde la IRQ0 hasta la IRQ15. En este caso el hardware de interrupciones del PC impone un esquema de prioridades estáticas en donde la IRQ0 es la petición de más alta prioridad seguida por IRQ1, IRQ8, IRQ9,...,IRQ14, IRQ15, IRQ3, IRQ4, IRQ5, IRQ6, IRQ7 en ese orden. Aunque este constituye el orden de prioridades por defecto del INTHAL, el esquema de prioridades soportado es de prioridades dinámicas y difiere completamente del anterior. En lo adelante es necesario entonces distinguir entre:

- **Prioridad Lógica**: valor de prioridad (entre 0 y 255) soportado por el INTHAL y asignado dinámicamente por el núcleo.

- **Prioridad Física**: prioridad relativa impuesta por el hardware de interrupción.

Como consecuencia de lo anterior, el INTHAL tiene la responsabilidad de establecer un esquema de prioridad lógica incompatible con el esquema de prioridad física, impuesto automáticamente por los PICs del sistema. Esto es posible mediante la manipulación apropiada de los registros ISR ("*Interrupt Service Register*") e IMR ("*Interrupt Mask Register*") de los PICs 8259A lo cual, en esencia, se logra en dos etapas:

- **Anulación del manejo de prioridades automático de los PICs**: En esencia, el HAL tiene que asegurar que los registros ISR de ambos 8259 siempre se encuentren en 00000000B de forma que permitan cualquier IRQ que haya sido habilitada de forma explícita por los IMRs. Esto es posible mediante la ejecución oportuna del comando de fin de interrupción EOI en los 8259A.

- **Gestión de las prioridades por software**: Para implantar la gestión de prioridades lógicas, una vez ocurrida cada IRQ el INTHAL debe establecer de forma explícita los registros IMR de cada 8259A con una máscara que inhabilite todas las IRQ de menor o igual prioridad (incluida la IRQ actual) y habilite las IRQs de mayor prioridad.

5.3 El Controlado Programable de Interrupciones Personalizado Virtual (VCPIC)

Con el propósito de establecer el esquema de prioridad adecuado, el INTHAL está a cargo de mantener el estado de un Controlador de Interrupciones Programable a la Medida Virtual o VCPIC ("*Virtual Custom Programmable Interrupt Controler*") el cual es capaz de soportar el esquema integrado de prioridades (Ver Figura 30). El VCPIC mantiene una tabla con la prioridad actual para cada IRQ y el nivel de prioridad actual del sistema.

Siempre que haya un cambio de estado del VCPIC el INTHAL calcula y establece la mascara apropiada en los IMRs de ambos PICs 8259A.

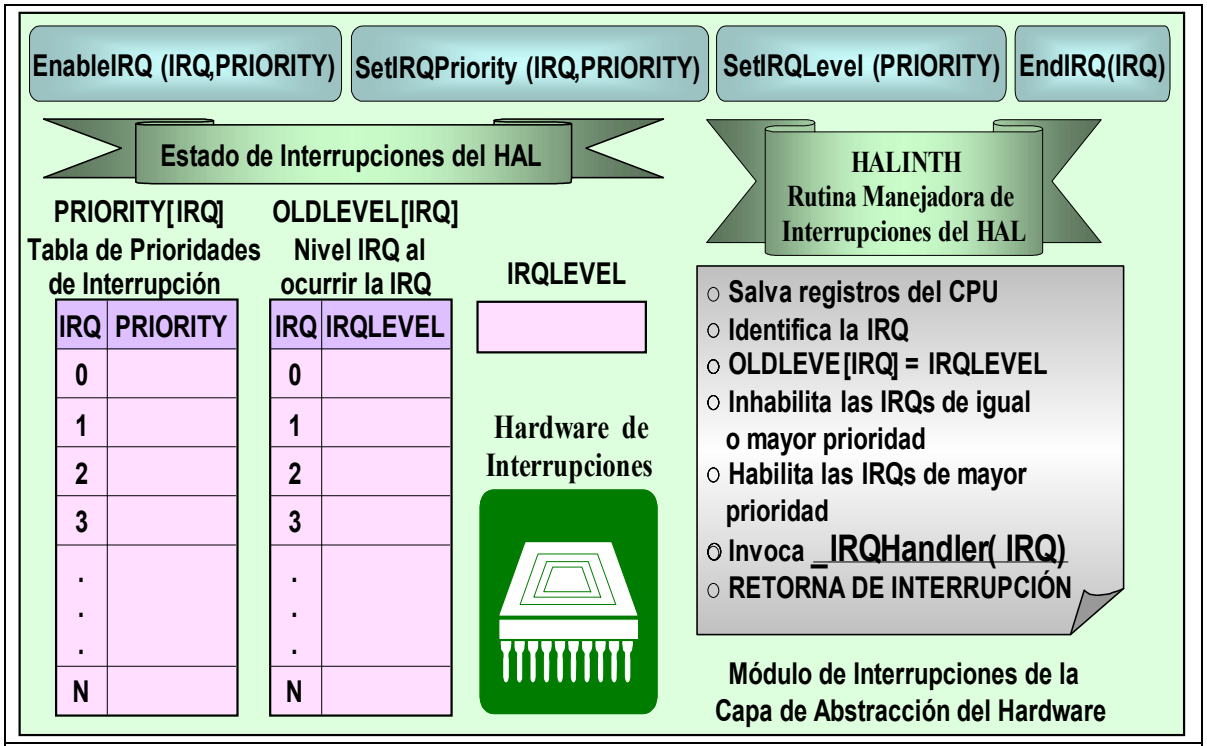


Figura 30. Diagrama del módulo de Interrupciones de la capa de abstracción del hardware (HAL)

La Figura 31 muestra un diagrama de secuencia UML para ilustrar los elementos que intervienen en el tratamiento de la interrupción y la secuencia de eventos (o mensajes) entre ellos cuando ocurre una interrupción. En el extremo izquierdo se representan dos elementos de hardware: el dispositivo que emite la interrupción y el PIC 8259. El otro elemento de hardware que interviene es la CPU; sin embargo, en lugar de representar a esta directamente se han representado los elementos de software involucrados en la petición de interrupción: los componentes VCPIC (INTHAL) y KRNLINT del núcleo; así como, el objeto de sincronización asociado a la interrupción (IRQx Sync Object) y la Tarea Activada por Hardware (HAT) que maneja la petición proveniente del dispositivo (IRQxHAT).

Como puede observarse en el extremo derecho del diagrama de secuencia, IRQxHAT indica su disposición de manejar la interrupción invocando la operación wait() sobre el objeto asociado a esta IRQxSync Object (mensaje 1) lo cual provoca que quede bloqueada hasta tanto se produzca la interrupción (mensaje 2).

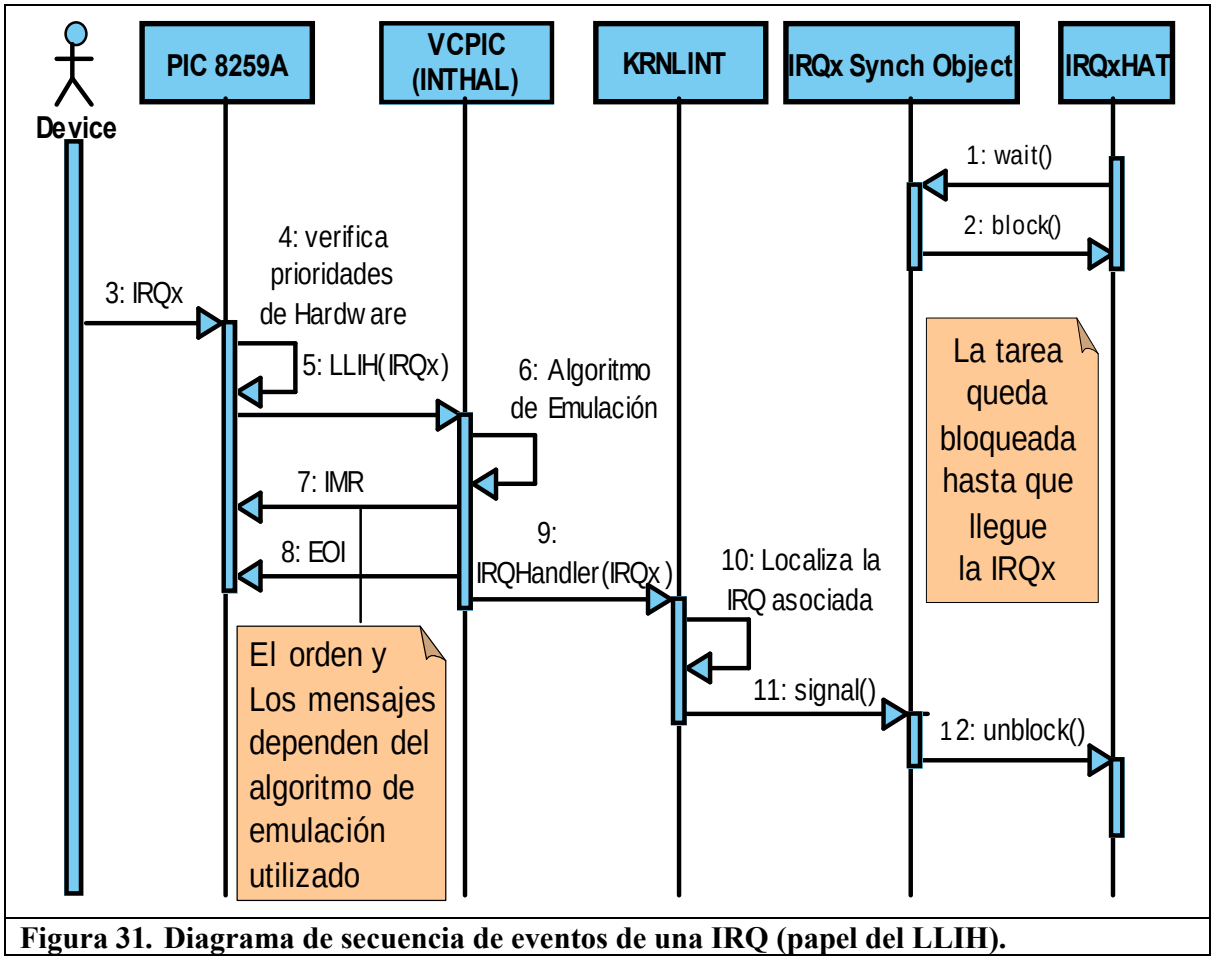


Figura 31. Diagrama de secuencia de eventos de una IRQ (papel del LLIH).

Cualquiera sea la petición de interrupción que emita un dispositivo, con ayuda del PIC 8259A (mensaje 3) y el mecanismo de la tabla de vectores de interrupción de la CPU, se produce la invocación del Manejador de interrupción de bajo nivel o LLIH del INTHAL recibiendo como argumento un identificador de la IRQ (mensaje 5). El centro de este manejador es el mecanismo de emulación del espacio integrado del prioridades (mensaje 6) lo cual se logra mediante la manipulación adecuada del registro de máscara de interrupción del 8259 (IMR) y la emisión del comando de fin de interrupción (EOI) – mensajes 7 y 8. Los detalles de este mecanismo están regidos por el algoritmo de emulación que se utilice (son posibles más de uno, cuyos detalles se analizarán en el resto de este capítulo). Cualquiera que sea el algoritmo de emulación, el resultado final será el establecimiento de la prioridad adecuada y la invocación del servicio **IRQHandler**() del KRNLINT (mensaje 9).

El trabajo de KRNLINT consiste en localizar el objeto de sincronización asociado a la petición de interrupción (mensaje 10) para entonces invocar sobre este la operación signal() (mensaje 11). El efecto de esta operación es desbloquear a la tarea que esperaba por dicha interrupción (mensaje 12) quién entra en la lista de tareas elegibles por el planificador y (en virtud de su prioridad) puede tomar el control de la CPU para darle el servicio apropiado a la IRQ del dispositivo.

5.4 Algoritmos de Emulación con enmascaramiento físico explícito

En esencia, el esquema de emulación con enmascaramiento físico consigue inhabilitar la administración automática de las prioridades de las interrupciones en el PIC para entonces dejar la administración de las prioridades de las IRQ en manos del planificador del sistema operativo. Esto se consigue sincronizando el enmascarado/desenmascarado de las IRQ con el establecimiento del nivel de prioridad actual del sistema. De esta forma se consigue que el sistema operativo en cada instante habilite/inhabilite de forma automática las diferentes IRQs en dependencia de la relación que exista entre la prioridad de cada una de ellas y la prioridad de la tarea actualmente en ejecución.

En nuestro diseño de la lógica de anulación del esquema de administración de prioridades del PIC implementamos dos algoritmos de emulación: uso de **EOI explícito** o uso de **EOI automático**. A continuación se analizan de forma detallada cada uno de ellos.

5.4.1 Algoritmo de Emulación #1: Uso de EOI Explícito

El fundamento de este algoritmo de emulación consiste en que cada ISR tenga un prólogo que enmascare explícitamente en el IMR del 8259A todas aquellas peticiones de interrupción (no deseadas) que tienen una prioridad lógica menor o igual a la prioridad lógica de la IRQ a la que se le está dando servicio. Una vez enmascaradas las IRQ no deseadas se envía de forma explícita al 8259A el EOI correspondiente a la IRQ actual. La Figura 32 muestra el seudo-código de la ISR del INTHAL para este algoritmo de emulación. Obsérvese que esta secuencia de pasos impide el empleo de la característica de EOI automático del PIC.

El enmascaramiento previo al envío del EOI impide que luego de que se limpie el ISR – paso (3) – el 8259A le notifique a la CPU interrupciones pendientes de menos prioridad lógica porque estas están ya enmascaradas.

Primero Enmascara y luego envía el EOI		
	Comando de la ISR común (LLIH)	**Propósito**
1	Salva registros de la CPU	
2	Sitúa IMR en el 8259A	Inhabilitar todas las IRQs de menor o igual prioridad lógica.
3	Envía EOI no específico	Permite interrupciones con menor nivel de prioridad física.
4	Invoca punto de entrada en el núcleo	
5	Restaura registros de la CPU	

Figura 32. Algoritmo de emulación #1: Uso de EOI explícito.

5.4.2 Algoritmo de Emulación #2: Uso de EOI automático

El fundamento de este es algoritmo es el mismo que el del caso anterior sólo que ahora se utiliza el modo de EOI automático del 8259A de modo que ya no es necesario enviar el EOI como parte de la atención a la interrupción.

5.4.3 Ejemplo de comportamiento de la emulación con enmascaramiento físico

Para ilustrar la operación de la emulación del modelo integrado utilizando el esquema de emulación mediante enmascaramiento físico de las IRQs antes descrito, supóngase un sistema compuesto por cuatro tareas: dos de ellas activadas por software que se denotan por SAT_1 y SAT_2 y dos activadas por hardware que se denotan como HAT_1 y HAT_2 y que son activadas por la IRQ1 y la IRQ2 respectivamente. Este conjunto de tareas posee una configuración de prioridades de acuerdo al siguiente orden (de mayor prioridad a menor prioridad): SAT_1, HAT_1, HAT_2 y SAT_2.

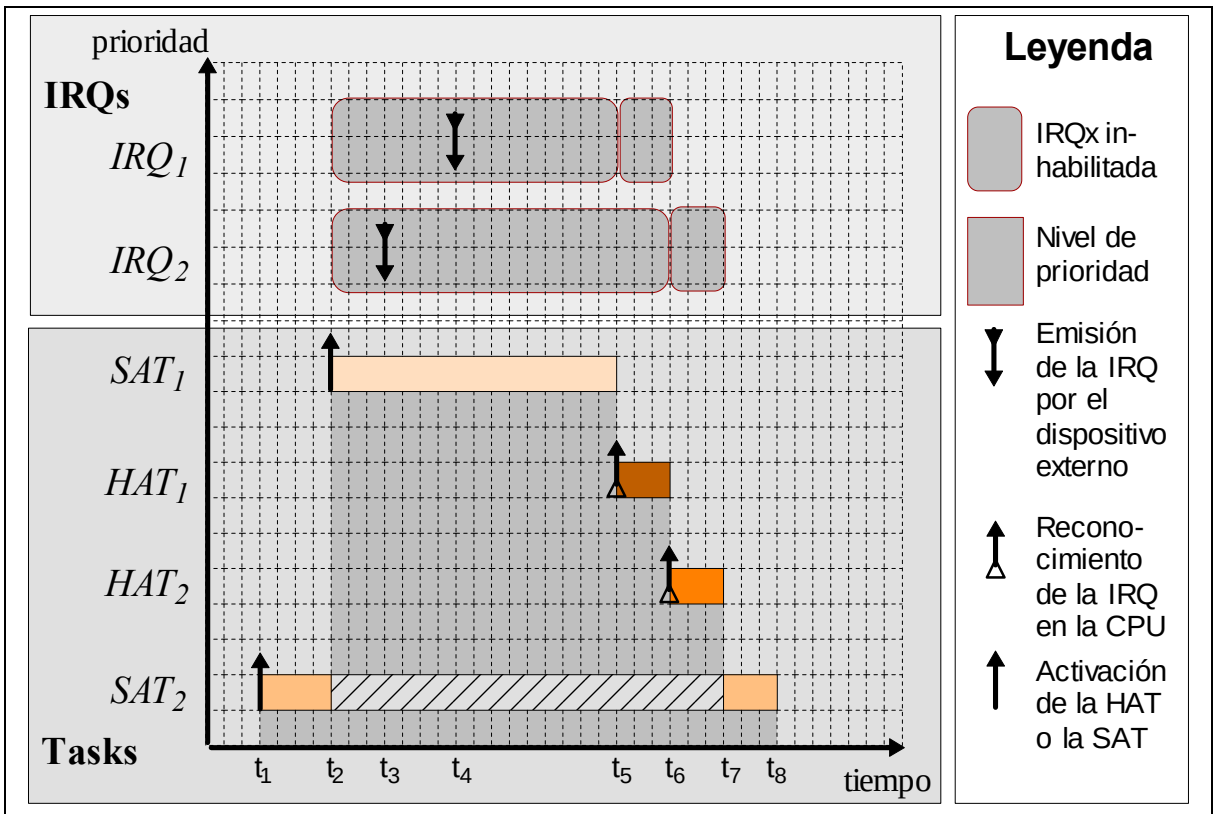


Figura 33. Emulación del Modelo Integrado con Enmascaramiento Físico.

La Figura 33 muestra un diagrama de tiempo con una secuencia de ejecución de este conjunto de tareas la cual se describe a continuación. Inicialmente no hay ninguna tarea en ejecución, por lo que el nivel de prioridad (de interrupción) del sistema es cero que se corresponde con todas las interrupciones (IRQ1 e IRQ2) habilitadas. A partir de esta situación se producen los siguientes eventos:

- En el instante de tiempo t_1 se activa la tarea SAT_2 y como no hay ninguna tarea de mayor prioridad lista para ser ejecutada, entonces se inicia su ejecución de forma inmediata. La conmutación de contexto hacia la tarea SAT_2 eleva el nivel de prioridad (de las interrupciones) hasta el nivel correspondiente a dicha tarea (representado por la región sombreada en la figura); sin embargo, como ambas HATs poseen mayor prioridad que la prioridad de SAT_2 ambas permanecen habilitadas (no hay escritura de la máscara del PIC).

- SAT_2 continúa ejecutándose hasta el instante de tiempo t_2, momento en el cual se produce la activación de SAT_1. Debido a que esta posee una mayor prioridad, expropia de forma inmediata a SAT_2 y se eleva el nivel de prioridad de las interrupciones hasta el nivel correspondiente a SAT_1. Debido a que SAT_1 posee mayor prioridad que HAT_1 y HAT_2, como parte de esta conmutación de contexto, el sistema operativo enmascara automáticamente la IRQ1 y la IRQ2 en el registro IMR del PIC de forma que estas no interfieran con la ejecución de SAT_1 (primera escritura de la máscara en la secuencia).

- En el instante de tiempo t_3 el dispositivo externo asociado a la IRQ2 emite una petición de interrupción; sin embargo, como dicha IRQ está enmascarada en el PIC, la misma no interrumpe (ni consume tiempo de ejecución) de la CPU. El PIC se encarga de recordar dicha petición (en su registro IRR) hasta el momento en que sea desenmascarada.

- En el instante de tiempo t_4 el dispositivo externo asociado a la IRQ1 emite una petición de interrupción; sin embargo, como dicha IRQ está enmascarada en el PIC, la misma no interrumpe (ni consume tiempo de ejecución) de la CPU. El PIC se encarga de recordar dicha petición hasta el momento en que sea desenmascarada.

- En el instante de tiempo t_5 finaliza la ejecución de la SAT_1 y el sistema intenta hacer una conmutación de contexto de regreso a la SAT_2. Al restituir el nivel de prioridad a la prioridad de SAT_2 el sistema desenmascara la IRQ2 y la IRQ1 en el PIC (segunda escritura de la máscara) instante en el cual el PIC le envía la IRQ1 (que estaba pendiente) a la CPU provocando que la conmutación de contexto se haga realmente hacia la tarea HAT_1. Como parte de esta conmutación de contexto se establece el nivel de prioridad (de interrupción) correspondiente a HAT_1 lo cual vuelve a enmascarar la IRQ1 y la IRQ2 (tercera escritura de la máscara) para pasar a la ejecución de HAT_1.

- En el instante de tiempo t_6 finaliza la ejecución de HAT_1 y el sistema intenta nuevamente hacer una conmutación de regreso a la SAT_2. Al restituir el nivel de prioridad a la prioridad de SAT_2 el sistema desenmascara nuevamente la IRQ2 y la IRQ1 en el PIC (cuarta escritura de la máscara) instante en el cual el PIC le envía la IRQ2 (que quedaba pendiente) a la CPU provocando que la conmutación de contexto se haga realmente hacia la tarea HAT_2. Como parte de esta conmutación de contexto se establece el nivel de prioridad (de interrupción) correspondiente a HAT_2 lo cual vuelve a enmascarar la IRQ2 (quinta escritura de la máscara). Obsérvese que ahora la IRQ1 permanece desenmascarada debido a que la HAT correspondiente posee mayor prioridad. Producto de esta conmutación de contexto se pasa a la ejecución de HAT_2.

- En el instante de tiempo t_7 finaliza la ejecución de HAT_2 y como no quedan peticiones de interrupción pendientes en el PIC, el sistema consigue finalmente hacer una conmutación de regreso a la SAT_2. Como parte de esta conmutación se disminuye el nivel de prioridad lo que provoca que se vuelva a desenmascarar la IRQ2 (sexta escritura de la máscara).

- Por último en el instante t_8 la SAT_2 finaliza su ejecución quedando la CPU ociosa.

Como se puede apreciar claramente a partir del ejemplo, el empleo de la emulación del modelo integrado con enmascaramiento físico explícito consigue de forma efectiva planificar la llegada de las IRQs a la CPU. De hecho, este consigue el efecto de convertir los eventos asíncronos e impredecibles de solicitud de interrupción (flechas en la parte superior de la figura) en interrupciones a la CPU sincronizadas con el planificador del sistema. Debido al empleo de un planificador de tiempo real, los eventos de petición de interrupción que llegan a la CPU lo hacen de forma predecible.

Como puede apreciarse, este esquema de emulación consigue eliminar por completo las perturbaciones de las ISRs sobre las tareas a costa de un incremento en la sobrecarga de la conmutación de contexto debido a la necesidad de establecer el registro IMR del PIC. En la secuencia del ejemplo anterior se registraron un total de seis escrituras de máscara.

5.4.4 Análisis de factibilidad de la implementación mediante emulación

Como se puso de manifiesto en el ejemplo de la sección anterior, la implementación de la emulación con enmascaramiento físico explícito (utilizando cualquiera de las variantes de anulación del esquema de prioridades del PIC antes descritas – EOI explícito o EOI automático), introduce una sobrecarga operativa adicional en la conmutación de contexto debido a la necesidad de calcular y establecer el nivel de interrupción actual en el hardware de interrupciones del sistema. Sea δ^M este tiempo de sobrecarga. Entonces la Ecuación (9) incluyendo δ^M se puede reescribir como:

$$U_i^P = \frac{C_i + 2\delta^M}{T_i} + \sum_{j \in (P(i) - H(i))} \frac{C_j + 2\delta^M}{T_j} + \sum_{j \in H(i)} \frac{c_j^H + 2\delta^p + 2\delta^M}{T_J^H}$$

Ahora, la disminución en la utilización $U_{loss} = U^{PI*}$ debido a la sobrecarga del modelo integrado utilizando el esquema de enmascaramiento físico será:

$$U_i^{PI*} = \frac{2\delta^M}{T_i} + \sum_{j \in (P(i) - H(i))} \frac{2\delta^M}{T_j} + \sum_{j \in H(i)} \frac{2\delta^p + 2\delta^M - \delta^I}{T_J^H} \qquad (13)$$

Como se puede observar en la ecuación 13, el diseño e implementación del sistema utilizando el VCPIC, impone una penalización en el desempeño del sistema. Sin embargo, incluso cuando U_i^{PI*} no sea inferior a U_{iS}, este esquema posee las ventajas discutidas en las Secciones 4.1.1 y 4.1.2 y entre las cuales se destaca la capacidad de predecir el comportamiento temporal.

En aquellos sistemas donde el determinismo temporal sea importante, esta implementación puede ser una alternativa atractiva a la eliminación de las interrupciones (ver sección 3.4.4) Muchas de las aplicaciones de este tipo preferirán la adición de una sobrecarga adicional al sistema con el propósito de lograr el determinismo temporal sin sacrificar los beneficios del tratamiento por interrupción.

5.5 Uso de Enmascaramiento Virtual

A medida que los procesadores se han hecho más rápidos y complejos, el enmascaramiento de las interrupciones se ha ido haciendo cada vez más costoso. En los procesadores canalizados ("*pipelined*"), la escritura de la máscara de interrupción del procesador requiere típicamente un vaciado de la canalización. En los sistemas superescalares, la manipulación del nivel de interrupción requiere expedir instrucciones escalares, limitando aún más el desempeño. Adicionalmente, la mayoría de los procesadores incorporan sólo parte de la lógica de enmascaramiento de interrupción, el resto del enmascaramiento de interrupción se implementa en un controlador de hardware fuera del procesador. En este caso, el enmascaramiento se lleva a cabo en tres pasos: 1) inhabilitar las interrupciones del procesador, 2) escribir los registros de máscara fuera del chip, y 3) por último rehabilitar las interrupciones. El primer paso requiere un vaciado de la canalización, y el segundo requiere un acceso fuera del procesador potencialmente costoso. Esto representa un incremento significativo en la latencia relativa de las manipulaciones de la máscara de interrupción

5.5.1 Protección de interrupciones optimista

Desafortunadamente, con frecuencia el núcleo del sistema operativo tiene que inhabilitar las interrupciones para impedir la expropiación mientras se están ejecutando secciones de código que manipulan estructuras de datos críticas para el sistema operativo y volverlas a habilitar al terminar dicha sección crítica de código. Con el propósito de acelerar el protocolo de entrada y salida a estas secciones críticas del núcleo en [93] Stodolsky, Chen, y Bershad introdujeron lo que se denominó **protección de interrupciones optimista**.

La idea original de la protección optimista consiste el procedimento siguiente: (1) el protocolo de entrada a una sección crítica dentro del núcleo establece **una máscara de interrupción de software**, que indica qué interrupciones necesitan enmascararse. La máscara de interrupción de hardware no se cambia. (2) A la entrada de todos los manejadores de interrupción se sitúa una sección de prólogo que verifica la máscara de interrupción en software y si la interrupción en cuestión está lógicamente enmascarada se pospone la ejecución del resto de la ISR hasta un instante posterior. (3) El protocolo de salida de la sección crítica, verifica si existe alguna interrupción pendiente y si es el caso se transfiere el control al manejador de interrupción correspondiente antes de reanudar el cómputo "normal".

Con el propósito de simplificar el código, el enmascaramiento optimista recomienda que en el caso de que ocurra una interrupción lógicamente enmascarada además de recordar la petición de interrupción, el prólogo de la ISR actualice la máscara de hardware según especificó la máscara de interrupciones de software antes de devolver el control. En este

caso, luego de que se manejen las interrupciones diferidas como parte del protocolo de salida de la sección crítica, adicionalmente se restaura la máscara de interrupción de hardware a su nivel original.

5.5.2 Adaptación para Sistemas de Tiempo Real

La penalización en el desempeño analizada en la sección 5.4.4 puede disminuirse grandemente si se adapta la idea original del enmascaramiento optimista. Mediante esta técnica, cuando el nivel de interrupción del sistema se eleva del nivel A al nivel B, las IRQs con niveles de prioridad entre A y B no son inhabilitadas físicamente, de modo que estas IRQs **no deseables** pueden ocurrir en realidad. Si se produce cualquiera de estas interrupciones, entonces la IRQ se enmascara realmente para evitar que ocurran de nuevo.

A diferencia de la idea original de la protección optimista, en este caso el enmascaramiento de las IRQs no deseadas luego de su ocurrencia no es un elemento opcional (con el propósito de simplificar la lógica de la implementación), sino que se convierte en un aspecto obligatorio con el propósito de garantizar la predecibilidad temporal. En este caso, la petición de interrupción se registra de alguna forma para hacerse efectiva luego que el nivel de prioridad baje lo suficiente. Con esto se evita la sobrecarga de enmascarar interrupciones, ya que en la gran mayoría de las veces no ocurren peticiones de interrupción mientras el sistema ejecuta códigos de alta prioridad (generalmente muy rápidos). Por otro lado, al bajar el nivel de prioridad del sistema, hay que verificar que previamente no se haya enmascarado una IRQ que puede ocurrir en el nivel nuevo, en cuyo caso hay que modificar la máscara de IRQs para habilitar las IRQs que deben estarlo.

5.5.3 Adaptación al Modelo Completamente Integrado

La existencia del modelo integrado implica otras modificaciones a la idea original del enmascaramiento optimista. Con el enmascaramiento virtual ahora es posible la ocurrencia de interrupciones no deseadas. Sea Pc el nivel de prioridad actual, entonces cualquier interrupción I con prioridad Pi que se produzca de forma no deseada cumple con la condición $Pi \leq Pc$. Si ello sucede, esta interrupción tiene que ser físicamente enmascarada en el registro IMR del 8259 correspondiente, de forma que se evite la ocurrencia de una segunda interrupción. Además, la ocurrencia de esta IRQ tiene que ser registrada. En este caso no se modifica el nivel de prioridad actual (Pc).

5.5.3.1 Mecanismo de Enmascaramiento de la IRQ indebida

Como se planteo en 5.5.2, a diferencia de la protección optimista, en el enmascaramiento virtual es ineludible el enmascarado de la IRQ indebida dentro del LLIH con vistas a garantizar el determinismo temporal. Sin embargo, todavía existen tres formas posibles de realizar el enmascaramiento de una IRQ indebida que garantizan una cota máxima en la inversión de prioridad debido a sus perturbaciones:

1. **Enmascarar exclusivamente la IRQ indebida que se produjo**: esto implica calcular una máscara que inhabilite la IRQ específica (sin tocar las demás) y fijarla. No se abundará más en esta porque no presenta ninguna ventaja con respecto a las otras dos.

2. **Enmascarar todas las IRQs con prioridades menores o iguales que Pi**: esta variante tiene dos ventajas: primero, es muy fácil de implementar ya que solamente hay que fijar la máscara (previamente calculada por **setIrqPriority**()) correspondiente a la interrupción indebida **i**; segundo, cuando se invoque a **setIrqLevel**() para elevar el nivel de prioridad no es necesario que este calcule la máscara correspondiente al nuevo nivel de prioridad (si debe hacerlo en el caso de que se invoque para disminuir el nivel de prioridad). Debido a la necesidad de invocar a **setIrqLeve**l() en cada conmutación de contexto, el hecho de que no siempre se tenga que calcular la máscara implica una menor sobrecarga de conmutación de contexto. Sin embargo, este método tiene como inconveniente el permitir la ocurrencia de otras IRQs indebidas (todas aquellas ***x*** que cumplen con Pc > Px > Pi). Esto no sólo implicaría un nuevo enmascaramiento de IRQs; sino que también, arroja un peor caso más grande que el necesario en la perturbación causada por interrupciones indebidas.

3. **Enmascarar todas las IRQs con prioridades menores o iguales que P_c**: esta opción equivale a igualar el nivel de IRQ físico (máscara física) con el nivel de prioridad del sistema (nivel de IRQ virtual). Con esto se garantiza que una vez ocurrida la interrupción indebida, no ocurrirá ninguna otra interrupción indebida mientras el nivel de prioridad del sistema sea mayor o igual al nivel actual. En esta variante, el servicio **serIrqLevel**() tiene que calcular la máscara de interrupción correspondiente a cada nivel que se establezca, independientemente de si se invoca para elevar o para disminuir el nivel de prioridad actual del sistema. Esto tiene como inconveniente una mayor sobrecarga en la conmutación de contexto pero ofrece la ventaja de que para cada nivel de prioridad puede ocurrir sólo una interrupción indebida (su ocurrencia enmascara todas las demás) lo que arroja el mejor valor posible de la la perturbación en el peor caso debido a interrupciones indebidas.

5.5.3.2 Mecanismo para "recordar" la IRQ indebida

Una vez ocurrida y enmascarada una interrupción indebida, esta debe recordarse para cuando la prioridad del sistema baje lo suficiente como para permitir su ocurrencia. En este aspecto, la propia existencia del modelo integrado de administración de interrupciones (planificación y sincronización) posibilita dos opciones fundamentales en dependencia de qué componente tiene la responsabilidad de memorizarlas:

1) **La responsabilidad de "memorizar" se sitúa en el propio VCPIC (INTHAL)**. Esta es la opción equivalente a la idea original del prologo de interrupción en el enmascaramiento optimista. Para lograr esto habría que mantener un indicador de ocurrencia para cada posible IRQ ("continuación" en la terminología de Stodolsky [93]), donde se recordarían las interrupciones indebidas ocurridas. Luego, al bajar el nivel de prioridad, se simularía su ocurrencia invocando a **IRQHandler**(**IRQ**) dentro del servicio del HAL **setIrqLevel**(). Esta opción tiene como ventaja la total transparencia del núcleo con respecto al modo de enmascaramiento que utilice el INTHAL.

Vale la pena señalar que esta opción es la única disponible cuando se utiliza un esquema convencional de manejo de interrupciones. Obsérvese además que ello implicar un incremento en el tiempo de cómputo del servicio **setIrqLevel**(). En el caso del esquema convencional de administración de interrupciones y tareas, y para los sistemas operativos de propósito general para los cuales el enmascaramiento optimista fue diseñado, esto no constituye una dificultad, ya que la invocación a **setIrqLevel**() no se realizar como parte de cada conmutación de contexto, sino como parte del protocolo de entrada/salida a las secciones críticas del núcleo y siempre este servicio será más económico que la alternativa de manipular directamente el nivel de interrupción (que es precisamente la razón por la cual se introdujo el enmascaramiento optimista).

Para el caso del esquema integrado de administración de interrupciones y tareas, y para los sistemas operativos de tiempo real, lo anterior no es válido. Ello se debe primero a un rediseño completo del paradigma de implementación y por otro a los objetivos de diseño:

- Ahora **setIrqLevel**() no se invoca como parte del protocolo de entrada/salida a las secciones críticas del núcleo sino que se invoca como parte de cada conmutación de contexto. De hecho, la propia conmutación de contexto constituye una sección crítica del kernel por tanto no es posible simular una interrupción desde **setIrqLevel**(). Obsérvese que con el empleo del modelo integrado aquí propuesto se elimina completamente la necesidad de inhabilitar las interrupciones dentro del micro-núcleo. Las secciones críticas más internas del mismo se protegen simplemente inhabilitando la expropiación (protocolo de techo de prioridad inmediata), mientras que las secciones críticas más externas se protegen utilizando mutexes (que pueden implementar cualquiera de los protocolos de herencia de prioridades disponibles).

- La eficiencia del esquema integrado, y el grado de factibilidad de planificación que puede alcanzar el sistema como tal, es muy sensible al tiempo de ejecución en el peor caso del servicio **setIrqLevel**(). Por consiguiente cualquier mínimo incremento en el mismo constituye un inconveniente significativo.

2) **La responsabilidad de "memorizar" se sitúa en el núcleo** (**KRNLINT**). Afortunadamente la solución a estas aparentes dificultades a la hora de utilizar este esquema de enmascaramiento optimista para el caso del modelo integrado se encuentran en su mismo diseño, el cuál no sólo integra el esquema de prioridades sino que también ofrece un mecanismo integrado de comunicación y sincronización entre manejadores de interrupción y tareas. Bajo este modelo no se hace necesario que el VCPIC le oculte al núcleo la ocurrencia de una interrupción indebida. Ahora se abre la posibilidad de notificarle este evento al núcleo para que este sea el que se encargue de recordarla. Existen dos alternativas en que el VCPIC le puede notificar al núcleo la ocurrencia de una interrupción indebida: (1) invocar a un punto de entrada (**IRQHandler** (**IRQ**)) siempre que ocurriese una interrupción debida y a otro punto de entrada distinto en los casos en que la interrupción sea indebida; (2) invocar siempre a un mismo punto de entrada pero con un parámetro adicional que indica si la interrupción es debida o indebida (**IRQHandler**(**IRQ**, **flag**)). Sin embargo, obsérvese

que realmente no se hace necesario que el VCPIC le indique al núcleo si la interrupción que se produjo es debida o indebida ya que este tiene suficiente información para diferenciar la una de otra (con el nivel de prioridad del sistema) e incluso pudiera hacerlo de forma totalmente transparente.

El hecho está en que cuando ocurre una interrupción (se invoca a **IRQHandler(irq**)) la tarea del núcleo es señalar a los objetos de sincronización asociados a dicha IRQ e invocar al planificador. Si se establece la restricción (ya conveniente de todas formas) de que sólo pueden asociarse a una IRQ objetos que tengan capacidad de memorizar las señales (semáforos, buzones) entonces se garantiza de forma totalmente transparente la capacidad de recordar una interrupción indebida. En este caso, una interrupción debida provocaría la expropiación de la tarea actualmente en ejecución en favor de la IST que espera por ella (esta tiene mayor prioridad) mientras que la ocurrencia de una interrupción indebida pondría en listo a la IST correspondiente pero no provocaría expropiación de la tarea actualmente en ejecución (su IST posee menor prioridad). Esta sería planificada automáticamente cuando el nivel de prioridad del sistema baje lo suficiente.

En el caso de que se esté en modo de enmascaramiento virtual y ocurre una interrupción i cuya prioridad **Pi > Pc**, entonces el LLIH del INTHAL debe realizar las mismas operaciones que en el modo de enmascaramiento físico con la excepción del enmascaramiento de la IRQ.

La tabla de la Figura 34 resume las diferencias entre la idea inicial del enmascaramiento de interrupciones optimistas (propuesta en [93]) y la adaptación que hemos realizado para adecuarla a sistemas de tiempo real con el modelo integrado de administración de interrupciones y tareas. A esta adaptación le denominamos enmascaramiento virtual.

Técnica	Enmascaramiento Optimista	Enmascaramiento Virtual
Sistemas a los que se destina	Sistemas operativos de **Propósito general**	Sistemas operativos de **tiempo real**
Enmascarami-ento Físico	**Opcional**, con el propósito de simplificar la implementación	**Obligatorio**, con el propósito de garantizar la predecibilidad temporal
Omisión de la ejecución del manejador de IRQ indebida	**Explícita** por parte del prólogo de cada ISR	**Automático** por parte del planificador del núcleo (la tarea no se ejecuta porque no tiene suficiente prioridad)
Registro de Interrupcio-nes Indebidas	**Explícito** como parte del prólogo de cada ISR	**Automático** como parte del objeto de sincronización saciado a la IRQ
Ejecución del manejador pospuesto	**Explícito** como parte del protocolo de salida de las secciones críticas.	**Automático** por parte del planificador del núcleo (cuando sea la tarea de más prioridad)

Figura 34. Diferencias entre enmascaramiento optimista y virtual

5.5.4 Soporte de Interrupciones sensibles al Nivel

Cualquiera de los esquemas de enmascaramiento virtual descritos en la sección 5.5.3 funciona sin ningún tipo de dificultad para el caso de que las líneas de petición de interrupción sean sensibles al flanco (ver sección 5.1.1) ya que en este caso aunque LLIH no elimine la causas que originó la petición de interrupción (ni enmascare la interrupción en el IMR del PIC) no se generarán más peticiones de interrupción. Sin embargo, como se planteo anteriormente, el problema con las líneas de petición de interrupción sensibles al flanco es que no pueden ser compartidas de forma confiable entre múltiples dispositivos periféricos. En los sistemas con bus EISA o PCI es posible programar la sensibilidad (al nivel o al flanco) de cada una de las líneas de petición de interrupción de forma individual. Solamente los canales de interrupción que se conectan al bus EISA/ISA pueden ser programados en el modo sensible al nivel con el propósito de ser compartidos entre varios dispositivos. Esto significa que IRQ0, IRQ1, IRQ2 e IRQ13 deben programarse sensibles al flanco. IRQ8 debe programarse sensible al nivel.

Si alguna línea se ha configurado para ser sensible al nivel entonces la solicitud de interrupción debe ser eliminada antes de que se envíe el comando EOI o se produciría otra interrupción. Este requerimiento constituye una dificultad para los esquemas de enmascaramiento virtual de la sección 5.5.3. El problema es que en los algoritmos de enmascaramiento virtual descritos, el LLIH enmascara la petición de interrupción en el registro IMR del PIC sólo en los casos en que la IRQ sea indebida (comportamiento optimista). Si no es el caso, el LLIH no enmascara la IRQ pero tampoco elimina la condición que provocó la petición en el dispositivo solicitante. Como consecuencia, una vez que el LLIH envía el EOI (en el caso de EOI explicito) o si se usa EOI automático, la interrupción se volverá a generar haciendo que el sistema caiga en un lazo infinito de invocación al LLIH.

Afortunadamente la situación anterior se puede solucionar fácilmente haciendo que en el caso de que la línea de petición de interrupción sea sensible al flanco el LLIH sitúe explícitamente la máscara de IRQ. Observe que esta modificación no afecta el tiempo de conmutación de contexto ya que la máscara sólo se sitúa en caso de ocurrencia de las interrupciones y no en cada conmutación de contexto. Realizada esta aclaración, en el resto de este trabajo se supondrá líneas de interrupción activadas por flanco.

5.5.5 Ejemplo de Comportamiento de la emulación con enmascaramiento Virtual

En esta sección se ilustra la operación de la emulación del modelo integrado utilizando el esquema de emulación mediante enmascaramiento virtual antes descrito. Para ello se utilizará el mismo ejemplo (conjunto de tareas y secuencia de eventos) que se introdujo en la sección 5.4.3 para ilustrar la emulación mediante enmascaramiento físico explícito. En realidad el comportamiento exacto estará en dependencia del mecanismo de enmascaramiento de la IRQ indebida que se utilice (ver sección 5.5.3.1) por lo que a continuación se analizarán los tres casos.

5.5.5.1 Enmascarar exclusivamente la IRQ indebida que se produjo

La Figura 35 muestra el comportamiento para el caso en que sólo se enmascare la IRQ indebida que se produjo (caso 1 de la sección 5.5.3.1). La secuencia de eventos del ejemplo ilustra el peor caso posible de perturbación a la tarea SAT_1 que se produce cuando ocurren todas las IRQs de forma indebida.

Como puede observarse, en este caso cada una de las peticiones de interrupción de los dispositivos externos (IRQ2 en el instante t_3 e IRQ1 en el instante t_4) interrumpe a la CPU y provoca la ejecución del LLIH que inmediatamente se percata que es una interrupción indebida y enmascara la IRQ correspondiente, señala al objeto de sincronización de interrupción asociado a la IRQ y retorna inmediatamente a la ejecución de la tarea SAT_1 interrumpida (esto ocurre en los instantes t_3' y t_4' para los casos de la IRQ2 y la IRQ1 respectivamente).

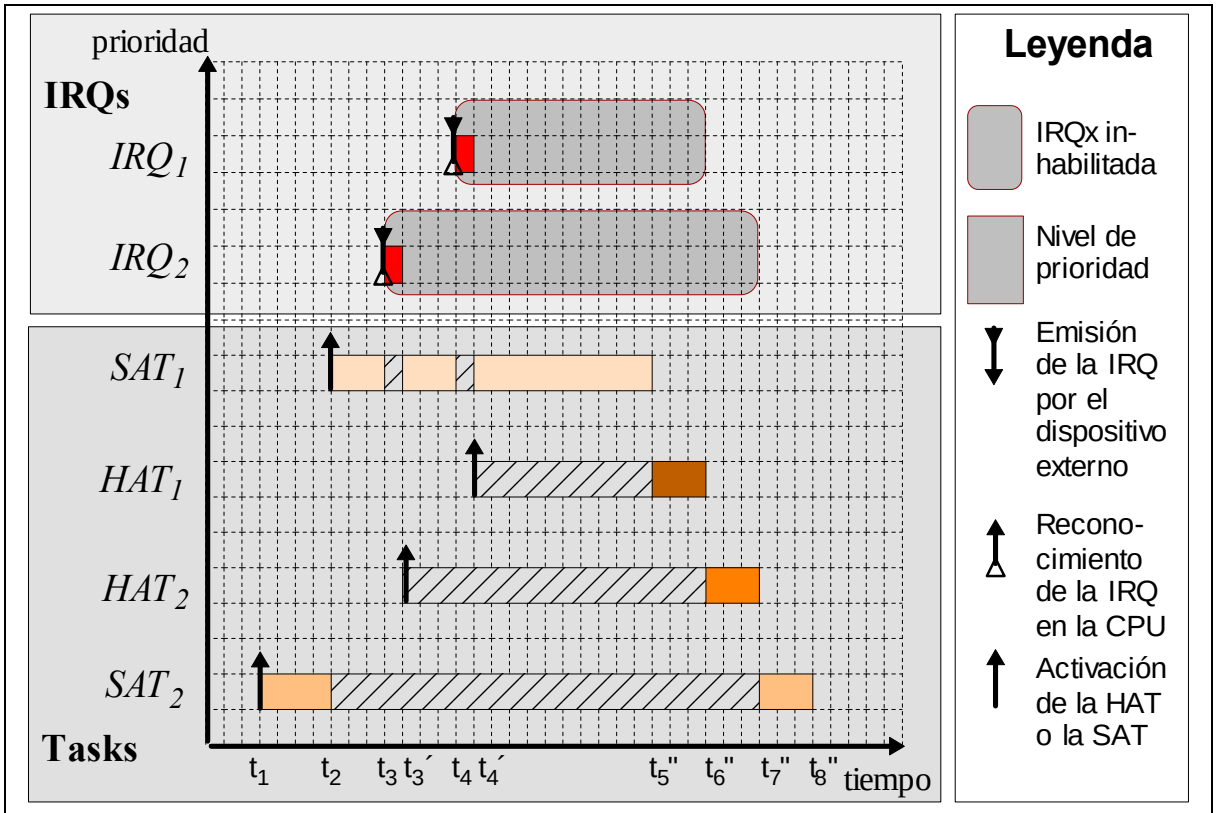


Figura 35. Enmascaramiento Virtual: Enmascara sólo la IRQ indebida que se produjo.

Con esta variante de emulación las conmutaciones de contextos hacia tareas de mayor prioridad o expropiaciones (como ocurre en el instante t_2) no modifican el nivel de prioridad de las interrupciones (y por tanto tampoco establecen la máscara de interrupción). Sólo se producen escrituras de la máscara cada vez que ocurre una interrupción indebida para enmascararla (instantes t_3 y t_4) y al finalizar la ejecución de las HATs correspondientes para desenmascararla (instantes t_6" y t_7"). Como puede observarse, en el caso de la secuencia de ejemplo, sólo ocurren cuatro operaciones de escritura de la máscara (en lugar de las 6 que ocurrieron para la misma secuencia cuando se utilizó enmascaramiento físico explícito – ver sección 5.4.3) para una disminución del 33% de estas operaciones (en el peor caso). Como puede apreciarse en la figura, el costo de esta reducción (mejora en el

comportamiento promedio) es la introducción de perturbaciones pequeñas y acotadas (intervalos t_3-t_3' y t_4-t_4') a la ejecución de la tarea SAT_1 (empeoramiento del comportamiento en el peor caso). Sin embargo, debido a que el número (dos en este caso) y el tamaño de estas perturbaciones están acotados, el sistema continúa exhibiendo un comportamiento predecible.

5.5.5.2 Enmascarar IRQs con prioridades menores o iguales que la IRQ indebida

La Figura 36 muestra el comportamiento para el caso en que se enmascaren todas las IRQs con prioridades menores o iguales que la IRQ indebida que se produjo (caso 2 de la sección 5.5.3.1). La secuencia de eventos del ejemplo ilustra el peor caso posible de perturbación a la tarea SAT_1 que se produce cuando ocurren todas las posibles IRQs indebidas en orden inverso de prioridad (primero la de menor prioridad y luego la de mayor prioridad).

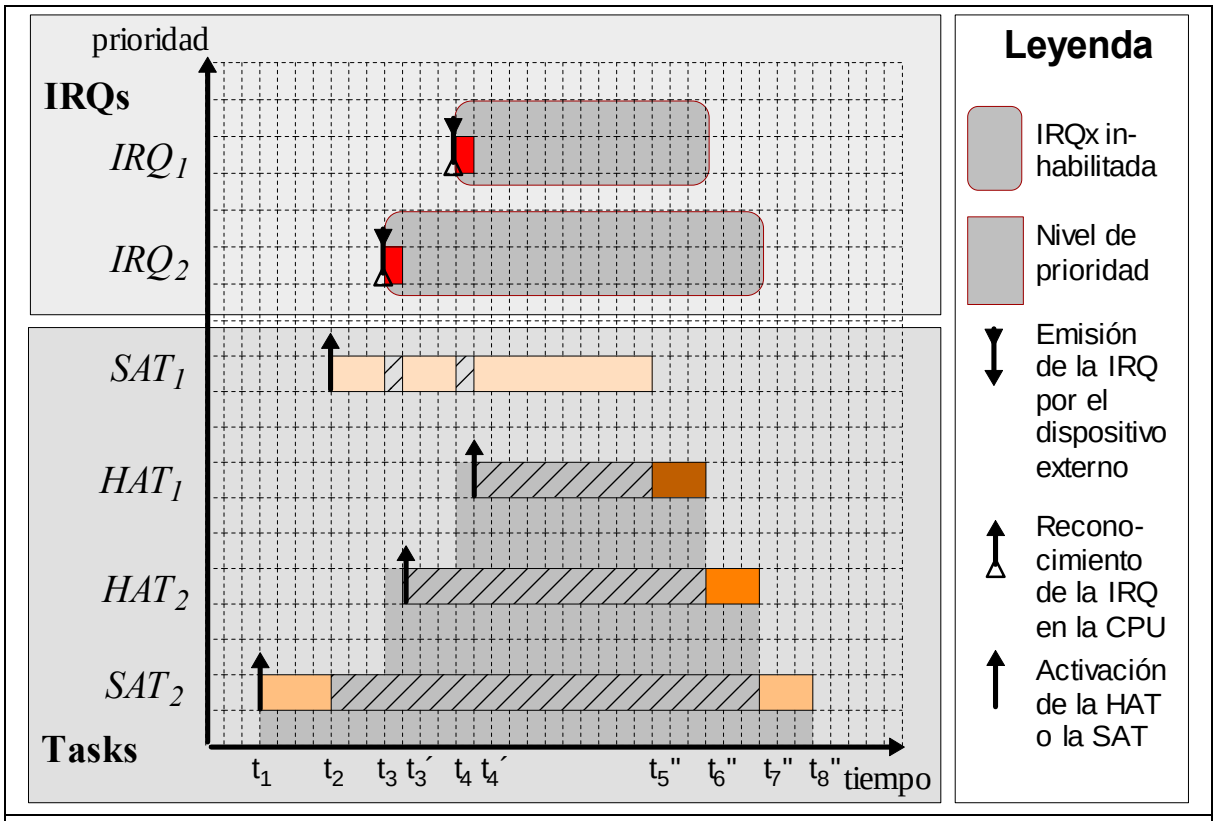


Figura 36. Enmascaramiento Virtual: Enmascara IRQs con prioridades menores o iguales que la IRQ indebida que se produjo.

Para este caso específico, cada una de las peticiones de interrupción de los dispositivos externos (IRQ2 en el instante t_3 e IRQ1 en el instante t_4) interrumpe a la CPU y provoca la ejecución del LLIH que inmediatamente se percata que es una interrupción indebida y establece físicamente el nivel de prioridad de interrupción correspondiente a la IRQ indebida. Esto efectivamente enmascara todas las IRQs con prioridades menores o iguales que la IRQ correspondiente. Posteriormente el LLIH señala al objeto de sincronización de interrupción asociado a la IRQ y retorna inmediatamente a la ejecución de la tarea SAT_1

interrumpida (esto ocurre en los instantes $t_3´$ y $t_4´$ para los casos de la IRQ2 y la IRQ1 respectivamente).

Con esta variante de emulación las conmutaciones de contextos hacia tareas de mayor prioridad o expropiaciones (como ocurre en el instante t_2) no actualizan la máscara de interrupción. Al igual que en el caso anterior, sólo se producen escrituras de la máscara cada vez que ocurre una interrupción indebida para enmascararla (instantes t_3 y t_4) y al finalizar la ejecución de las HATs correspondientes para desenmascararla (instantes t_6" y t_7"). Nuevamente, para el caso de la secuencia de ejemplo, sólo ocurren cuatro operaciones de escritura de la máscara (en lugar de las 6 que ocurrieron para la misma secuencia cuando se utilizó enmascaramiento físico explícito – ver sección 5.4.3) para una disminución del 33% de estas operaciones (en el peor caso). Sin embargo, la mayoría de las veces ocurrirán menos aún ya que esto sólo sucede cuando las interrupciones indebidas ocurren en orden inverso de prioridades (lo cual es improbable). Como puede apreciarse en la figura, nuevamente el costo de esta reducción (mejora en el comportamiento promedio) es la introducción de perturbaciones pequeñas y acotadas (intervalos t_3-$t_3´$ y t_4-$t_4´$) a la ejecución de la tarea SAT_1 (empeoramiento del comportamiento en el peor caso). Sin embargo, debido a que el número (dos en este caso) y el tamaño de estas perturbaciones están acotados, el sistema continúa exhibiendo un comportamiento predecible.

5.5.5.3 Enmascarar las IRQs con prioridades menores o iguales que el nivel actual

La Figura 37 muestra el comportamiento para el caso en que se enmascaren todas las IRQs con prioridades menores o iguales que el nivel de prioridad actual (caso 3 de la sección 5.5.3.1). Nuevamente, la secuencia de eventos del ejemplo ilustra el peor caso posible de perturbación a la tarea SAT_1. En este caso, no importa cual sea la petición de interrupción que ocurra de forma indebida (IRQ1 o IRQ2), su ocurrencia impedirá que cualquier otra pueda producirse de forma indebida (hasta tanto finalice la tarea en ejecución en ese instante).

En la secuencia específica, la IRQ2 sucede inicialmente de forma indebida en el instante t_3 interrumpiendo a la CPU y provocando la ejecución del LLIH que inmediatamente se percata que es una interrupción indebida y establece físicamente el nivel de prioridad (de interrupción) actual. Esto efectivamente enmascara todas las IRQs que pudieran ocurrir de forma indebida (IRQ1 e IRQ en el ejemplo). Posteriormente el LLIH señala al objeto de sincronización de interrupción asociado a la IRQ y (en el instante $t_3´$) retorna inmediatamente a la ejecución de la tarea interrumpida SAT_1. Obsérvese como en este caso, cuando el dispositivo externo emite la IRQ1 en el instante t_4, ésta ya no interrumpe a la CPU (ya fue enmascarada por el LLIH cuando la IRQ2 ocurrió de forma indebida en t_3). Esta IRQ1 es recordada en el registro IRR del PIC y su emisión a la CPU será demorada (planificada) hasta tanto sea desenmascarada producto de la disminución del nivel de prioridad actual al finalizar la ejecución de SAT_1 (lo cual ocurre en el instante $t_5´$).

Igual que en las dos variantes anteriores de emulación con enmascarado virtual, las conmutaciones de contextos hacia tareas de mayor prioridad o expropiaciones (por ejemplo en el instante t_2) no actualizan la máscara de interrupción. Sólo se producen escrituras de la máscara para enmascarar IRQs, cada vez que ocurre una interrupción de forma indebida.

En la secuencia del ejemplo esto sucederá en el instante t_3 (primera escritura de la máscara). Otra escritura de la máscara (la segunda) se producirá en el instante t_5', al finalizar la ejecución de la SAT_1 cuando el sistema intenta conmutar a la HAT_2 (que se había activado – puesto lista – en t_3'), al disminuir el nivel de prioridad hasta la prioridad correspondiente a la HAT_2 se desenmascara la IRQ1 (no la IRQ2) lo que provoca que el PIC le envíe a la CPU (planifique) la IRQ1 que estaba pendiente provocando la interrupción de la CPU para ejecutar la HAT_1. Obsérvese que esta conmutación hacia la HAT_1 es en efecto una expropiación a la HAT_2 y por tanto no provoca una nueva escritura de la máscara (como ocurriría en el caso de que se usase enmascaramiento físico – ver sección 5.4.3). Es por esto que ahora la HAT_1 se ejecuta con la IRQ1 habilitada.

La última escritura de la máscara (tercera) que se produce en la secuencia de ejemplo ocurre (en el instante t_7') al finalizar la ejecución de la HAT_2 como parte de la disminución del nivel de prioridad (de interrupción) hasta el nivel correspondiente a la tarea SAT_2 que se reanuda. En este caso, la escritura es tal que vuelve a desenmascarar ambas IRQs.

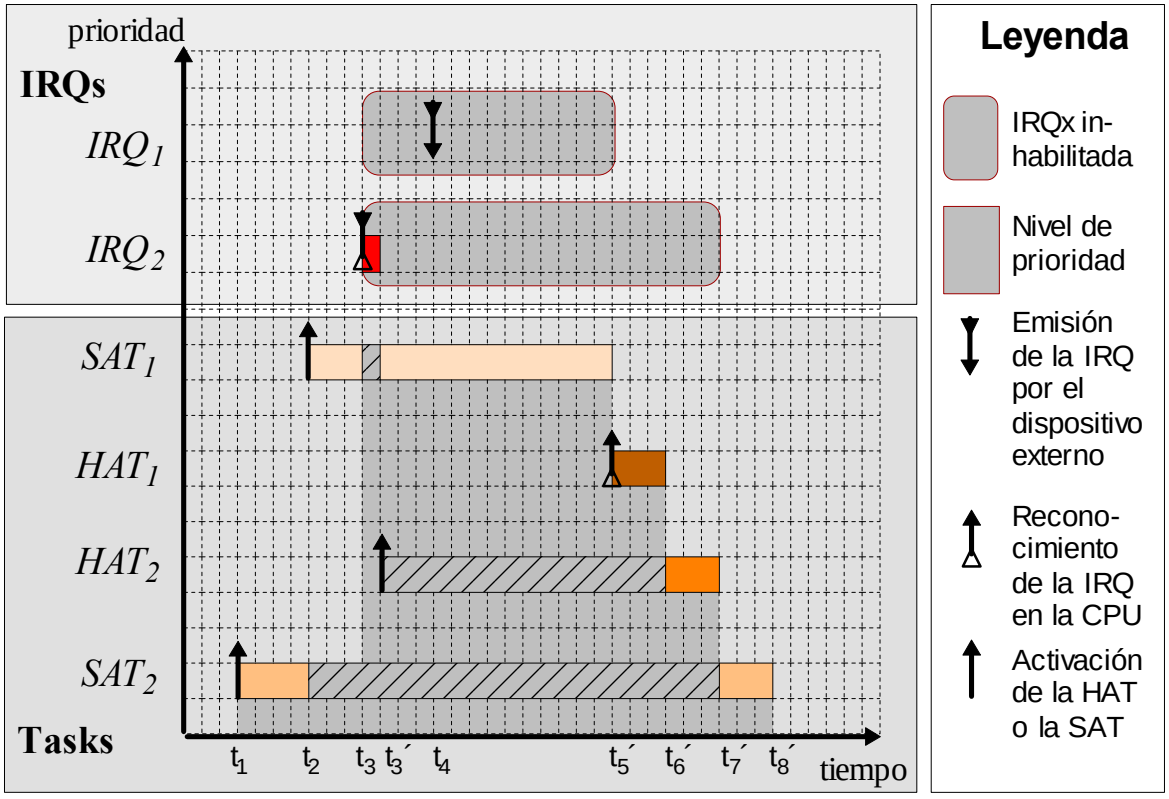


Figura 37. Enmascaramiento Virtual: Enmascara IRQs con prioridades menores o iguales al Nivel de Prioridad Actual

En total, en esta secuencia del ejemplo sólo ocurren tres operaciones de escritura de la máscara (en lugar de las 6 que ocurrieron para la misma secuencia cuando se utilizó enmascaramiento físico explícito – ver sección 5.4.3 – o las cuatro que ocurrieron cuando se utilizaron las dos variantes de emulación del enmascarado virtual anteriores – ver secciones 5.5.5.1 y 5.5.5.2) para una disminución del 50% con respecto al modo de emulación con enmascaramiento físico. Para este caso, el costo de esta reducción (mejora

en el comportamiento promedio) es la introducción de una sola perturbación pequeña y acotada (intervalo t_3-t_3') a la ejecución de las tareas (empeoramiento mínimo del comportamiento en el peor caso). Sin embargo, debido a que ahora sólo puede ocurrir una sola perturbación (para cada tarea) y el tamaño de esta perturbación está acotado, el sistema continúa exhibiendo un comportamiento predecible. Adicionalmente, la afectación al comportamiento promedio es la mínima posible.

5.5.6 Análisis de factibilidad con enmascaramiento virtual

Cuando se utiliza enmascarado virtual, el enmascarado de una IRQs sólo puede ocurrir si la misma se produjo realmente (de forma indebida), mientras que el desenmascarado se produce sólo, si como parte de una conmutación de contexto (de salida de una actividad), es necesario habilitar nuevamente a dicha IRQ.

Para las actividades en el conjunto *P*(i) el peor caso se dará cuando todas las IRQs del subconjunto *H*(i) ocurran de forma indebida mientras se estén ejecutando actividades en *P*(i) de mayor prioridad que la IRQ correspondiente. En este caso, cada una implicaría una primera escritura de la máscara como parte de su manejador y una segunda escritura cuando finalice la actividad en $P(i)$ que fue indebidamente interrumpida por ésta. Sin embargo, como esta escritura sólo se producirá en el caso de que se haya producido realmente la IRQ, la misma no se debe asociar a cada conmutación de contexto en $P(i)$; sino que, basta con asociar dos escrituras de máscara ($2\delta^{M}$) a cada posible activación de las actividades en $H(i)$. En consecuencia, la fracción de utilización U_i^M debido a la interferencia que ejercen las HAT del conjunto $H(i)$ sobre la tarea t_i está dada por:

$$U_i^M = \sum_{j \in H(i)} \frac{c_j^H + 2\delta^p + \gamma + 2\delta^M}{T_J^H}$$

Donde γ es el tiempo de ejecución del prólogo asociado a las IRQs no deseadas, que es necesario para registrar su ocurrencia. Además, ahora es necesario tener en cuenta la perturbación asociada a la posible ejecución de los pequeños prólogos de atención a las IRQs asociadas a cualquiera de las actividades en los conjuntos $S(i)$ y $L(i)$ (no la actividad de atención como tal) cuando ellas se producen de forma indebida. Este prólogo se encarga de enmascarar dicha IRQ. En este caso, sólo hay que contabilizar un enmascaramiento por cada ocurrencia de IRQ no deseada, ya que ninguna conmutación de contexto de actividades en $P(i)$ implicarán un desenmascarado de cualquiera de las IRQs asociadas a actividades en $S(i)$ o $L(i)$. Sin embargo, el número de veces que puede ejecutarse este prólogo depende del modo en que se haga el enmascaramiento (según se describió en la sección 5.5.3.1). Sea γ es el tiempo de ejecución del prólogo asociado a las IRQs no deseadas, necesario para recordar su ocurrencia, entonces:

1) Para los casos en que se enmascare sólo la IRQ actual o todas las IRQ con prioridades menores o iguales que la IRQ actual (o sea, casos 1 y 2 de la sección 5.5.3.1), entonces en el peor caso, cada una de las IRQ del conjunto $S(i) \cup L(i)$ puede ocurrir a lo sumo una vez. En consecuencia, la ecuación (9) puede escribirse como:

$$U_i^P = \frac{C_i + |S(i) \cup L(i)| \cdot (\gamma + \delta^M)}{T_i} + \sum_{j \in (P(i) - H(i))} \frac{C_j}{T_j} + \sum_{j \in H(i)} \frac{c_j^H + 2\delta^p + \gamma + 2\delta^M}{T_J^H} \quad (14)$$

A partir de la ecuación (14) podemos establecer que la disminución en la utilización $U_{loss} = U_i^{PI*}$ debido a la sobrecarga del modelo integrado con enmascaramiento virtual (empleando el esquema de enmascarado 1 ó 2 de la sección 5.5.3.1), será:

$$U_i^{PI*} = \frac{|S(i) \cup L(i)| \cdot (\gamma + \delta^M)}{T_i} + \sum_{j \in H(i)} \frac{\gamma + 2\delta^p + 2\delta^M}{T_J^H} \quad (15)$$

Observe que aunque los esquemas de enmascaro 1 y 2 de la sección 5.5.3.1 tiene el mismo peor caso, en el esquema 2 este peor caso es muy poco probable y en general tendrá un comportamiento promedio mejor que el esquema 1.

2) Para el caso en que se establezca la máscara que corresponde con la prioridad actual del sistema P_C (o sea, el caso 3 de la sección 5.5.3.1), entonces sólo puede ocurrir una sola IRQ no deseada en el conjunto $S(i) \cup L(i)$. En consecuencia, la ecuación (9) puede escribirse como:

$$U_i^P = \frac{C_i + \gamma + \delta^M}{T_i} + \sum_{j \in (P(i) - H(i))} \frac{C_j}{T_j} + \sum_{j \in H(i)} \frac{c_j^H + 2\delta^p + \gamma + 2\delta^M}{T_J^H} \quad (16)$$

A partir de la ecuación (15) podemos establecer que la disminución en la utilización $U_{loss} = U_i^{PI*}$ debido a la sobrecarga del modelo integrado con enmascaramiento virtual (empleando el esquema de enmascarado 3 de la sección 5.5.3.1), será:

$$U_i^{PI*} = \frac{\gamma + \delta^M}{T_i} + \sum_{j \in H(i)} \frac{\gamma + 2\delta^p + 2\delta^M}{T_J^H} \quad (17)$$

Obsérvese que, a diferencia del mecanismo tradicional, consistente en utilizar una ISR mínima postergando el servicio al nivel de tarea (subsecciones 4.2.1.1 y 4.2.2), este esquema optimista es temporalmente determinista. Ahora se introduce nuevamente una inversión de prioridad debido a una pequeña perturbación debido a la ejecución del prólogo de una interrupción no deseada (dada por $\gamma + \delta^M$), pero como demuestran las ecuaciones (15 ó 17), esta inversión de prioridad es acotada. Este esquema garantiza una administración de interrupciones predecible con una perdida en la utilización muy pequeña. También observe que ahora U_i^{PI*} (Ecuación 14) sólo depende de las HATs en el conjunto H(i) y no de las tareas activadas por software (como sucede en la Ecuación 12), haciendo al sistema más escalable

Como un comentario general, vale la pena observar que el esquema de protección optimista es un caso específico de una técnica de optimización más general utilizada con frecuencia en los sistemas operativos de propósito general conocida como técnicas perezosas ("*lazy*

techniques"), y que consiste en evitar la ejecución de operaciones costosas hasta que sean absolutamente necesarias. Aunque estas técnicas se han utilizado para optimizar el caso más frecuente, en este caso su aplicación a los sistemas operativos de tiempo real nos ha permitido disminuir la pérdida en la utilización en el peor caso.

5.6 Protocolo de enmascarado implícito y desenmascarado explícito

Una alternativa simple al modelo integrado aquí propuesto que también puede enfrentar el no determinismo introducido por las interrupciones sin necesidad de integrar el espacio de prioridades es el empleo de lo que denominamos **protocolo de enmascaramiento implícito y desenmascaramiento explícito**. Con esta técnica, el núcleo sitúa un LLIH común a todas las interrupciones que hace el reconocimiento inmediato de la interrupción al PIC (EOI) y acto seguido enmascara la petición de interrupción, evitando con ello ocurrencias adicionales de la misma petición. Luego, activa la ejecución de una tarea de servicio de interrupción o IHT ("*Interrupt Handler Task*") la cual se planifica igual que el resto de las demás tareas según la prioridad que le corresponda por el planificador (de tiempo real) del sistema. La IHT tiene entonces la responsabilidad de desenmascarar de forma explícita la petición de interrupción una vez que le haya dado servicio mediante un servicio de acuse de recibo que brinda el núcleo con ese propósito.

5.6.1 Análisis de Factibilidad

La operación de acuse de recibo (desenmascarado) de interrupción dentro de la IHT funciona como petición ("*request*") de una interrupción adicional qué sólo pueden ocurrir en respuesta ("*reply*") a esta petición. En consecuencia, con esta técnica, se consigue un efecto similar al empleo del esquema integrado con enmascaramiento virtual y el caso 1 de enmascaramiento. O sea, a lo sumo puede ocurrir una sola instancia de cada una de las IRQ asociadas a IHT de menor prioridad que la tarea t_i (que serían IRQ no deseadas). Con lo cual se consigue acotar la inversión de prioridad en el peor caso. Sin embargo, en este caso se añade la sobrecarga de una escritura de máscara más el envío del EOI a la latencia de interrupción de todas las IRQs y adicionalmente se añade una segunda escritura de máscara al tiempo de cómputo de cada una de ellas.

Sea δ^{EOI} el tiempo de cómputo asociado al envío del EOI al PIC, entonces la ecuación (9) puede escribirse como:

$$U_i^P = \frac{C_i + |S(i) \cup L(i)| \cdot (\gamma + 2\delta^M + \delta^{EOI})}{T_i} + \sum_{j \in (P(i) - H(i))} \frac{C_j}{T_j} + \sum_{j \in H(i)} \frac{c_j^H + 2\delta^p + \gamma + 2\delta^M + \delta^{EOI}}{T_J^H}$$

Por tanto, la pérdida en la utilización $U_{loss} = U_i^{PI*}$ debido al empleo del protocolo de enmascaramiento explícito y desenmascaramiento explícito esta dada por:

$$U_i^{PI*} = \frac{|S(i) \cup L(i)| \cdot (\gamma + 2\delta^M + \delta^{EOI})}{T_i} + \sum_{j \in H(i)} \frac{\gamma + 2\delta^p + 2\delta^M + \delta^{EOI}}{T_J^H} \qquad (18)$$

Como puede observarse (además del incremento en la latencia de interrupción debido al enmascaramiento de la IRQ como parte del LLIH), la sobrecarga introducida en términos de pérdida de utilización en este caso es superior al sobrecarga introducida por el esquema integrado con enmascaramiento virtual (ecuaciones 15 y 17).

Lo interesante de esta última técnica es que, con el propósito de hacer que las IHT fuesen completamente independientes del hardware de interrupciones del sistema, esta fue implementada en el micro-núcleo L4 como parte del proceso de portarlo a arquitecturas diferentes de la IA32 para la cual fue creado originalmente [27]. El resultado es que, como efecto colateral, este micro-núcleo (a pesar de no estar diseñado para aplicaciones de tiempo real [76]) consiguió acotar la inversión de prioridad de las interrupciones.

Sin embargo, esta conversión no es gratis ya que el acuse de recibo y enmascarado automático de la petición de interrupción como parte del FLIH trae consigo un aumento significativo en la latencia de interrupción y en la sobrecarga operativa ("*overhead*").

5.7 Diseño detallado del HAL para sistemas PC compatibles

En esta sección se describe la lógica de cada uno de los servicios de soporte de interrupciones del HAL para sistema PC compatibles, así como se muestran los seudo-códigos de cada uno de los mismos

5.7.1 Tratamiento a las interrupciones dentro del INTHAL

Como se planteo en la sección 5.2, para poder llevar a cabo su tarea el INTHAL tiene que capturar todas las IRQs. Para ello durante la inicialización el INTHAL (servicio **initIrqHardware**(…)) se encarga de situar la tabla de vectores de interrupción de la CPU de forma que todas las IRQ apunten a una entrada dentro de una la **Tabla de Entradas de IRQs del INTHAL**. Esta disposición se muestra en la Figura 35. Como se aprecia en el esquema, esta tabla, situada en la sección de código del INTHAL posee una entrada para cada una de las IRQs de hardware (un total de 15 entradas en un sistema PC-AT). En realidad, cada una de estas entradas posee un pequeño código que se encarga de invocar a un manejador de interrupción de bajo nivel LLIH ("*Low Level Interrupt Handler*") pasándole como parámetro la IRQ correspondiente.

En realidad, el tratamiento que le da el INTHAL a las IRQs está en dependencia de su estado (capturada o ignorada – ver el diagrama de estado de la Figura 21). Si la IRQ está en el estado ignorada, entonces el punto de entrada del LLIH es PANIC_ENTRY; si está capturada, el punto de entrada del LLIH es CAPTURED_ENTRY. Al cargarse el sistema, todas las entradas de la Tabla de Entradas de IRQ están configuradas para saltar al LLIH **PANIC_ENTRY(irq)**. Posteriormente la ejecución de los servicios **enableIrq**() y disableIrq() provocan las transiciones de estado que modifican el LLIH asociado a cada IRQ.

Adicionalmente, el HAL posee varios modos de emulación los cuales se resumen en la Figura 39. Cada modo de emulación posee algoritmos diferentes para los Manejadores de IRQ de Bajo Nivel. Estos se configuran durante la inicialización del INTHAL.

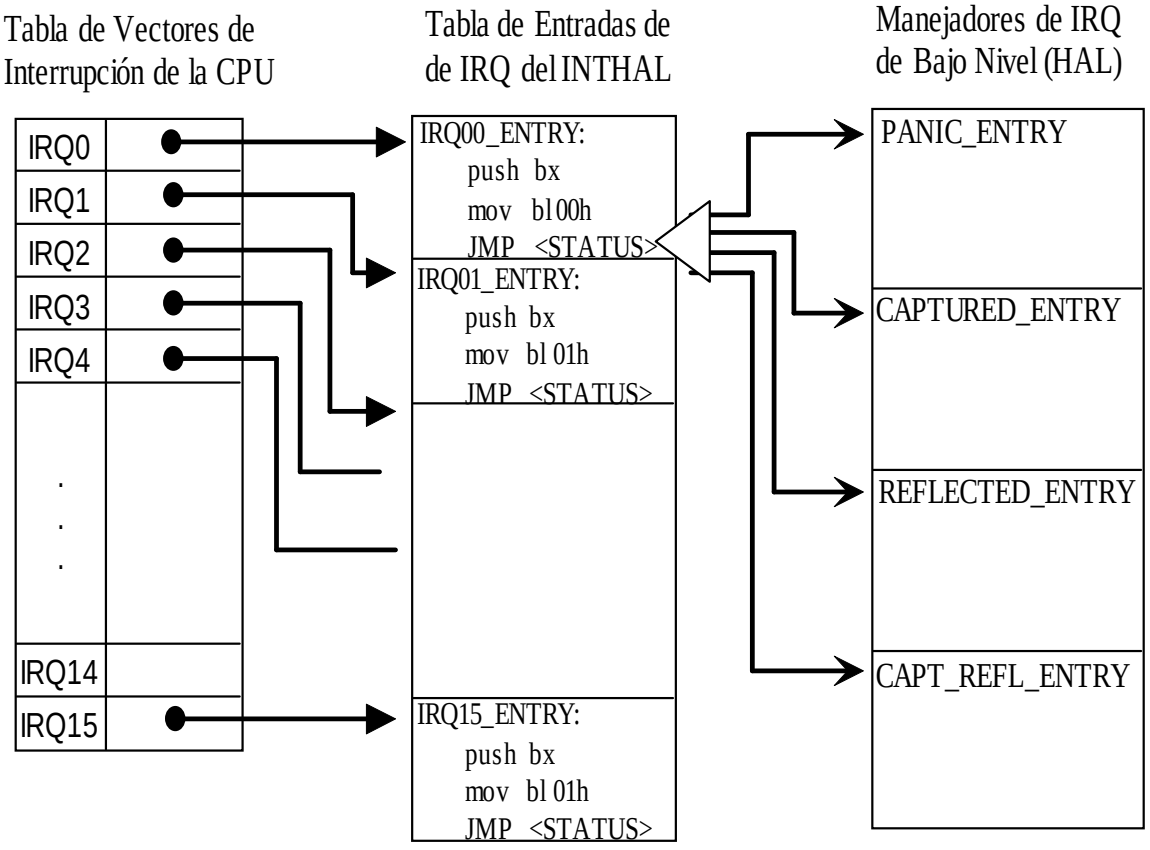


Figura 38. Flujo de control de las interrupciones a través del INTHAL

			Descripción	
Código	**EndIRQ()**	**Modo EOI**	**EOI**	**EndIRQ()**
0	0	0	Explícito	Soportado
1	0	1	Automático	Soportado
2	1	0	Explícito	No Soportado
3	1	1	Automático	No Soportado

Figura 39. Modos de Emulación del INTHAL

5.7.2 Estructuras de Datos para Mantener el Estado del HAL de Interrupciones

El INTHAL mantiene el estado del VCPIC mediante un conjunto de variables internas cada una de las cuales se especifican a continuación.

Se mantiene los siguientes arreglos de 16 elementos, uno por cada IRQ del sistema:

- **IRQ_Priority**: arreglo de bytes (8 bits) con la prioridad de cada IRQ del sistema. La IRQ2, que conecta en cascada al segundo 8259, se pone en prioridad máxima (255). Con esto se consigue que siempre esté habilitada y permita el paso de las interrupciones

desde el segundo 8259, sin tener que hacer una distinción especial para ella en los algoritmos internos.

- **IRQ_Mask**: arreglo de palabras (16 bits), cada una de las cuales mantiene la máscara a situar en los registros IMR de ambos 8259 en el momento de la ocurrencia de la IRQ correspondiente. La máscara posee un "0" en los bits correspondientes a las IRQ de mayor prioridad y "1" en los bits correspondientes a las IRQ de prioridad menor o igual (incluyéndose a la misma IRQ).
- **OLD_Level**: arreglo de bytes (8 bits) con el nivel de interrupción activo cuando ocurrió la IRQ particular.
- **OLD_Mask**: arreglo de palabras (16 bits) con la máscara de interrupción activa cuando ocurrió la IRQ particular.

Además se mantienen las siguientes variables:

- **Virtual_Mask_Mode**: Indicador que indica si se está operando en modo de enmascarado virtual (TRUE) u físico (FALSE).
- **IRQ_Level**: variable de tipo byte que conserva el nivel de prioridad actual del sistema.
- **Logical_Mask**: variable de tipo palabra que contiene la máscara de interrupción correspondiente al **IRQ_Level**.
- **Fisical_Mask**: variable de tipo palabra que contiene la máscara de interrupción que está situada realmente en los registros IMR de los 8259s. En el modo de enmascarado físico siempre va a coincidir con **Logical_Mask**. Sin embargo, en el modo de enmascarado virtual puede no coincidir.

5.7.3 Seudo-código de los servicios fundamentales del INTHAL

5.7.3.1 Inicialización y Terminación del INTHAL

Los servicios initIrqHardware(minDefault, maxDefault, eoiMode, maskMode) y restoreIrqHardware() se encargan de inicializar y restaurar el hardware de interrupciones. En el caso del servicio de inicialización su función es establecer los valores iniciales de las variables de estado interna e iniciar la tabla de vectores de interrupción según se describió en la secciones 5.7.1 y 5.7.2.

5.7.3.2 Servicios de Administración de Prioridaddes

Los servicios de administración de las prioridades de interrupción son quizás los más importantes del INTHAL para la implantación del VCPIC. Esos servicios son los que logran la ilusión de que a las IRQs se le puedan asignar prioridades de forma dinámica y dentro del mismo espacio de prioridades del planificador del núcleo. Este subconjunto está conformado por los servicios de interfaz **setIrqPriority**() y **setIrqLevel**(); así como, por los servicios auxiliares **Set8259IMR**() y **setIRQMask**().

Servicios auxiliares: Set8259IMR(mask) y setIRQMask(mask)

Es conveniente contar con un servicio auxiliar **set8259IMR**(…) que debe ser invocado cada vez que es necesario situar la máscara en el hardware de interrupción (registros IMR de ambos 8259). Mediante ello es posible mantener una variable en memoria (**Fisical_Mask**) con el valor actual de los registros de máscara de forma que se pueda evitar la costosa operación de E/S necesaria para establecer la máscara siempre que la nueva máscara a situar coincida con la que ya está presente.

Adicionalmente, para el soporte del enmascaramiento virtual, es conveniente tener un segundo servicio **setIRQMask**(…) que va a ser invocado siempre que se quiere establecer la máscara correspondiente con determinado nivel de prioridad.

En la Figura 40 se muestra el algoritmo del servicio **set8259IMR**(**mask**). El servicio primero verifica que la máscara física sea diferente de la máscara que se desea establecer. Si ello se cumple, establece realmente la máscara en los registros IMR de ambos 8259 y actualiza los valores de las variables de memoria **Logical_Mask** y **Fisical_Mask**.

```
Set8259IMR (mask) {
  Si ( mask <> Fisical_Mask )  {
     Logical_Mask = mask
     Fisical_Mask = mask
     Registros IMR de ambos 8259 ← mask
  }
```

Figura 40. Algoritmo de Set8259IMR()

```
setIRQMask (mask) {
  Si ( VirtualMaskMode =  TRUE)  {
     Si ( Fisical_Mask AND ( NOT mask ) )
       Set8259IMR(mask)
     Sino
       Logical_Mask = mask
  }
  Sino {
    Set8259IMR (mask)
  }
}
```

Figura 41. Algoritmo de setIRQMask()

En la Figura 41 se muestra el algoritmo del servicio **setIRQMask**(**mask**). El comportamiento de este servicio está en dependencia del modo de enmascaramiento (físico o virtual) el cual se conserva en la variable de estado **Virtual_Mask_Mode**:

- En el caso de enmascaramiento físico (**Virtual_Mask_Mode** = **FALSE**) este servicio simplemente invoca a **Set8259IMR**() para establecer realmente la máscara en ambos 8259.

- En el caso de enmascaramiento virtual (**Virtual_Mask_Mode** = **TRUE**), sólo sitúa la máscara si ello implica habilitar (desenmascarar) una IRQs. En caso de que la nueva máscara implique la inhabilitación (enmascarado) de una IRQ entonces sólo se actualiza el valor de la máscara lógica (**Logical_Mask**).

Puede observarse que en todo instante **Logical_Mask** contiene el valor de la máscara que se corresponde con el nivel de prioridad actual (**irqLevel**). En el caso en que se esté operando en el modo de enmascaramiento físico este valor siempre va a ser igual al valor de los registros IMR de ambos 8259 y al valor de **Fisical_Mask**. Por el contrario, si se opera en modo de enmascaramiento virtual el valor de **Logical_Mask** puede coincidir o no con el valor de **Fisical_Mask** (y de los registros IMR de ambos 8259). Sin embargo, en este modo siempre se tiene que cumplir que **Fisical_Mask** sea menos o igual de restrictiva que **Logical_Mask**. En otras palabras, siempre se tiene que cumplir la siguiente propiedad (o invariante):

$$(\neg \textbf{Logical\_Mask} \ \& \ \textbf{Fisical\_Mask}) = 0$$

Servicio setIrqPriority (IRQ, PRIORITY)

La Figura 42 muestra el algoritmo de **setIrqPriority**(). Este servicio permite establecer el nivel de prioridad de una IRQ. Es su responsabilidad hacer la correspondencia entre las prioridades asignadas a cada una de las IRQs (dentro del espacio de prioridades del sistema) y el valor correspondiente de la máscara a situar en el hardware de interrupciones (registros IMR de ambos 8259).

Como parte de esta tarea, **setIrqPriority**() mantiene los arreglos **IRQ_Priority** e **IRQ_Mask** ambos con una entrada por cada IRQ posible en el sistema. El primer arreglo conserva la prioridad actual de cada una de las IRQs (en el espacio de prioridades del núcleo). En cada invocación **setIrqPriority**() actualiza la entrada correspondiente a la IRQ a la que se le está estableciendo la prioridad. Luego, a partir de la nueva configuración de prioridades reflejada en el arreglo obtiene la máscara asociada a cada una de las IRQ. Estas máscaras se conservan en el segundo arreglo. Por último, este servicio invoca **setIRQMask**() para actualizar la máscara de interrupción (física y lógica) en los casos en que la IRQ a la que se le está estableciendo (modificando) la prioridad pase de habilitada a inhabilitada o viceversa.

```
setIrqPriority (IRQ, Priority) {
      IRQ_Priority[IRQ] ← Priority    /*  actualiza  la  prioridad
      */
      t_IRQMask  ← 0
      Para_( t_IRQ ← 0..IRQ_NUMBERs-1 )  {
            Si ( IRQ_Priority[t_IRQ] < Priority)  {
                  Pone 0 en bit IRQ de IRQ_Mask[t_IRQ]
                  Pone 1 en bit t_IRQ de t_IRQMask
            }
            Si ( IRQ_Priority[t_IRQ] = Priority )  {
                  Pone 1 en bit IRQ de IRQ_Mask[t_IRQ]
                  Pone 1 en bit t_IRQ de t_IRQMask
            }
            Si ( IRQ_Priority[t_IRQ] > Priority)  {
                  Pone 1 en bit IRQ de IRQ_Mask[t_IRQ]
                  Pone 0 en bit t_IRQ de t_IRQMask
            }
      }
      IRQ_Mask[IRQ] ← t_IRQMask;
      /* actualizar máscaras de 8259s */
      t_IRQMaskBit ← (1 << IRQ )/*  1  en  la  posición  de  la  IRQ
      */
      Si( Priority <= IRQ_Level )  {
            /*enmascara la IRQ (pone a 1 el bit)*/
            SetIRQMask( Fisical_Mask  OR  t_IRQMaskBit )
      }
      Sino {
            /*activa la IRQ (pone a 0 el bit)*/
            SetIRQMask( Fisical_Mask  AND (NOT t_IRQMaskBit) )
      }
}
```

Figura 42. Algoritmo de SetIrqPriority()

Servicio setIrqLevel (priority)

Este servicio es el encargado de situar el nivel de interrupción actual del sistema. Las IRQs con prioridades menores o iguales que este nivel quedan inhabilitadas. El mismo se encarga de mantener la variable **IRQ_Level**. Utilizando los arreglos **IRQ_Priority** e **IRQ_Mask**, el procedimiento determina la máscara a situar para establecer el nuevo nivel de prioridad Esta máscara es tal que inhibe todas las IRQ con un nivel de prioridad menor o igual que el **IRQ_Level**.

```
setIrqLevel (Priority) {
      /* Obtiene y establece la máscara de interrupción */
      IRQI ← 255    /* última IRQ encontrada (valor no válido) */
      IRQI_Priority ← 0  /* prioridad de última IRQ encontrada */
      Para (t_IRQ ← 0 .. IRQNUMBERs – 1) {
            Si ( IRQ_Priority[ t_IRQ ] <= Priority )
               y ( IRQ_Priority[ t_IRQ ] >= IRQI_Priority ) {
                  IRQI ← t_IRQ
                  IRQI_Priority ← IRQ_Priority[ t_IRQ ]
            }
      }
      IRQ_Level ← Priority
      Si (IRQI = 255)              /* No se encontró IRQ         */
            SetIRQMask( 0 );       /* habilitar todas las IRQs   */
      Sino
            SetIRQMask( IRQ_Mask[ IRQI ] );
}
```

Figura 43. Algoritmo de SetIrqLevel()

Luego de haber calculado la máscara que se corresponde con el nuevo nivel de prioridad, si la misma implica enmascarar o desenmascarar alguna IRQ se establece dicha máscara según el modo de enmascaramiento (físico o virtual) con ayuda del servicio **serIRQMask**().

La Figura 43 muestra el seudocódigo de **setIrqLevel**, téngase en cuenta que si una interrupción no está siendo usada (es decir, no ha sido capturada mediante **enableIrq()**, ni es interrupción heredada) su prioridad en el arreglo **IRQ_Priority** es cero. De esta forma el siguiente algoritmo la mantendrá siempre enmascarada.

En el seudocódigo puede observarse que **setIrqLevel** realiza una búsqueda entre las IRQ que tienen prioridad menor o igual que el nivel de prioridad a fijar (parámetro **Priority**), para seleccionar de ellas a la de mayor prioridad. Luego fija la máscara correspondiente a la IRQ encontrada dentro del arreglo **IRQ_Mask,**

5.7.3.3 Manejadores de Interrupción de Bajo Nivel (LLIH) del HAL

El INTHAL mantiene un manejador de interrupción de bajo nivel para cada posible estado en que puede estar una IRQ (Capturada o Ignorada). Estos manejadores reciben el control cada vez que se produce una IRQ con el estado asociado y reciben como parámetro la IRQ solicitada.

Manejador de Bajo nivel para las IRQ capturadas.

La rutina CAPTURED_ENTRY es la asociada a las interrupciones capturadas y tiene como responsabilidad fundamental anular el esquema de prioridades convencional de los PICs, hacer efectivo el esquema unificado de prioridades y transferir el control al manejador de interrupciones del núcleo. La forma en que consigue su propósito está en dependencia del

modo de fin de interrupción (explícito o automático) y del modo de enmascarado (físico o virtual).

En el modo de enmascaramiento físico (**Virtual_Mask_Mode** = 0), cuando ocurre una interrupción de prioridad **Pi**, necesariamente es porque se cumple que **Pi** > **irqLevel**. En este caso se procede enmascarando todas las IRQs con prioridad menor o igual que **irqLevel**, e incrementando el nivel de prioridad del sistema para hacerlo igual a **Pi**. Por último, se invoca al manejador de la capa superior (**IRQHandler()**).

Cuando se opera en modo de enmascaramiento virtual (**Virtual_Mask_Mode** = 1), es posible la ocurrencia de interrupciones indebidas. Estas cumplen con la condición **Pi <= irqLevel**. Si ello sucede, se procede a enmascarar físicamente dicha IRQ en el registro IMR del 8259 correspondiente (adicionalmente se actualiza la máscara física para reflejar este hecho) de forma que se impida la ocurrencia de una segunda interrupción indebida. Además se debe "recordar" la ocurrencia de la IRQ. Obsérvese que en este caso no se modifica el nivel de interrupción actual (**irqLevel**).

La Figura 44 muestra el seudo-código del LLIH asociado a las IRQ capturadas (CAPTURED_ENTRY):

```
CAPTURED_ENTRY(IRQ) {
  Salva los registros de la CPU que utiliza
  /*---------------- Para implementar EndIRQ() -----------------*/
  OLD_Level [IRQ] <-- IRQ_Level      /* Salva el nivel actual   */
  OLD_Mask [IRQ]  <-- Fisical_Mask   /* salva la m scara actual */
  /*------------------------------------------------------------*/
  Si ( VirtualMaskMode = TRUE ) {
    Si (IRQ_Priority[IRQ]<=IRQ_Level){/* Es IRQ indebida        */
      Set8259IMR (Logical_Mask)       /*  Si, máscara que debía */
      /*-------------------------------------------------------*/
      OLD_Mask [IRQ] = Logical_Mask   /*     Sólo para EndIRQ() */
      /*-------------------------------------------------------*/
    }
    sino {                            /*  No,                   */
      Logical_Mask <--IRQ_Mask[IRQ]   /*   Pone máscara lógica  */
      IRQ_Level <--IRQ_Priority[IRQ]  /*   Fija nivel actual    */
    }
  } Sino {
    Set8259IMR ( IRQ_Mask[IRQ] )      /* Sitúa máscara de IRQ   */
    IRQ_Level <-- IRQ_Priority[IRQ]   /* Fija el nivel actual   */
  }
  sendEOI                             /* Envía EOI segun modo   */
  IRQHAndler(IRQ)                     /* Transfiere al Núcleo   */
  Restaura los registros de CPU usados
}
```

Figura 44. Algoritmo del LLIH asociado a las IRQ capturadas

Servicio EndIRQ (IRQ)

Este servicio, cuyo algoritmo se muestra en la Figura 45 se usa para restaurar el nivel de prioridad existente antes de la ocurrencia de una IRQ particular. Este servicio es útil para señalar el fin de un manejador en sistemas donde las IRQs tienen la mayor prioridad global y su prioridad relativa no cambia. Nuestro micronúcleo (PARTEMOS) no utiliza este servicio. En su lugar fija explícitamente el nivel de IRQ deseado invocando a **setIrqLevel (Priority).**

```
EndIRQ( IRQ ) {
        Set8259IMR( OLD_Mask[IRQ] )
        IRQLevel ← OLD_Level[IRQ]
}
```

Figura 45 Algoritmo de EndIRQ()

5.8 Resumen

En este capítulo se ha puesto de manifiesto que el modelo completamente integrado de manejo de interrupciones puede implementarse incluso en sistemas con arquitecturas de interrupciones que no lo soportan de forma directa. Esto es posible siempre y cuando el hardware de interrupciones brinde la posibilidad de enmascaramiento explícito de cada una de las fuentes de petición de interrupción de forma individual y alguna vía para inhabilitar (de forma explícita o implícita) el algoritmo de planificación incorporado.

Se ha mostrado cómo esto es posible para el caso del hardware de interrupciones estándar de los sistemas PC compatibles. En específico se introdujeron cuatro modos de emulación: enmascaramiento físico explícito con y sin EOI automático y enmascaramiento virtual con y sin EOI automático. En el caso del enmascaramiento virtual se analizaron tres variantes y se eligió la implementación de una de ellas por ser la que ofrecía el mejor comportamiento promedio y en el peor caso. Todas las variantes de implementación son completamente predecibles pero con diferentes compromisos entre un mejor comportamiento en el peor caso y un mejor comportamiento promedio. Se mostraron los detalles de los algoritmos de emulación así como las ecuaciones que modelan el comportamiento temporal y la sobrecarga de cada uno de estos esquemas de emulación.

Como parte de nuestro trabajo, hemos realizado una implementación del diseño del subsistema de administración de interrupciones propuesto en el capítulo anterior utilizando estos modos de emulación. Adicionalmente, se desarrolló un micro-núcleo experimental que utiliza este componente y que nos permite mostrar la posibilidad y las virtudes de este esquema integrado.

6 Resultados Experimentales

En este capítulo presentamos los resultados experimentales obtenidos mediante la aplicación del modelo integrado. En este estudio realizamos dos tipos de experimentos: pruebas de comportamiento y pruebas de sobrecarga operativa ("*overhead*"). Las pruebas de comportamiento sirven para dos propósitos fundamentales: Primero, permiten comprobar la validez de los algoritmos de emulación descritos en el capítulo precedente para emular el modelo integrado; así como, su implementación. Segundo, nos permiten comprobar experimentalmente la característica de predecibilidad temporal del modelo integrado. Las pruebas de sobrecarga operativa nos permiten caracterizar la sobrecarga de los algoritmos de emulación para tener una idea cuantitativa de la factibilidad de este modelo para satisfacer los requerimientos temporales de las aplicaciones reales.

Los experimentos fueron realizados sobre un micro-núcleo experimental para sistemas de tiempo real que desarrollamos para este propósito.

En la primera parte de este capítulo abordamos las pruebas de comportamiento y en la segunda parte, las pruebas de desempeño.

6.1 Pruebas de Comportamiento.

6.1.1 Caso de Prueba

Con el propósito de comparar el comportamiento del esquema integrado con respecto al esquema convencional se realizó un programa de prueba conformado por dos actividades (tareas y manejadores de interrupción) asíncronas:

- t_1^S es un manejador de la interrupción del puerto serie sin requerimientos de tiempo real crítico (que recibe caracteres a una razón de 100 caracteres por segundo) con un tiempo mínimo entre llegadas T_1^S de 10 ms y un tiempo de cómputo en el peor caso C_1^S de 5 ms (para una utilización $U_1^S = 0.5$).

- t_2 es una tarea periódica de tiempo real crítico con período T_2 de 50ms, un tiempo de cómputo en el peor caso C_2 de 20ms (para una utilización $U_2 = 0.4$) y un plazo de 30ms.

Esta configuración puede aparecer por ejemplo en un sistema de control digital en la que t_2 es responsable del lazo de control mientras que t_1 tiene la responsabilidad de atender la comunicación con un sistema remoto (Ej. reportes y configuración).

6.1.2 Análisis de Factibilidad del Caso de Prueba

Como se vio en las secciones 2.3 y 4.2.2, el tiempo de respuesta de cada una de las actividades se puede calcular utilizando la siguiente ecuación de recurrencia [7]:

$$R_i^{(n+1)} = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i^{(n)}}{T_j} \right\rceil C_j$$

En el caso en que P(t_1^S) > P(t_2), entonces se tiene que:

$$R_1^S = C_1^S = 5ms$$

$$R_2^{(1)} = C_2 + \left\lceil \frac{R_2^{(0)}}{T_1^s} \right\rceil C_1^s$$

$$R_2^{(1)} = 20ms + \left\lceil \frac{20ms}{10ms} \right\rceil 5ms = 30ms$$

$$R_2^{(2)} = 20ms + \left\lceil \frac{30ms}{10ms} \right\rceil 5ms = 35ms$$

El valor de $R_2^{(2)}$ de 35 ms es mayor que el plazo de 30ms, por tanto la tarea periódica t_2 (que es de tiempo real duro) no cumple su plazo.

Si las prioridades se asignan en orden inverso, en donde P(t_2) > P(t_1^S), entonces se tiene que:

$$R_2 = C_2 = 20ms$$

$$R_1^{s(1)} = C_1^s + \left\lceil \frac{R_i^{s(0)}}{T_2^s} \right\rceil C_2$$

$$R_1^{s(1)} = 5ms + \left\lceil \frac{5ms}{50ms} \right\rceil 20ms = 25ms$$

$$R_1^{s(2)} = 5ms + \left\lceil \frac{25ms}{50ms} \right\rceil 20ms = 25ms$$

Como $R_1^{S(1)} = R_1^{S(2)}$, entonces se ha encontrado que el tiempo de respuesta de t_1^S en 25 ms es mayor que su plazo de 10ms, por tanto hay pérdida de interrupciones. Sin embargo, con esta configuración la tarea periódica t_2 (que es de tiempo real duro) si cumple su plazo. Como el manejador de la interrupción no es de tiempo real duro esto no es un problema mayor.

Puede observarse entonces que este conjunto de tareas es factible de planificar sólo cuando P(t_2) > P(t_1). O sea, esta es la única configuración de prioridades que garantiza los requerimientos temporales de la tarea periódica t_2.

6.1.3 Comportamiento utilizando el Modelo Tradicional (ISRs y Tareas)

La Figura 46 muestra la representación gráfica de una sección de las trazas generadas por la ejecución real del programa para el caso de prueba anterior, utilizando un esquema tradicional de tratamiento de interrupciones. En este caso t_1 se codificó como la ISR de la IRQ4. Bajo este esquema es imposible asignarle a una tarea mayor prioridad que a una interrupción (ISR). En consecuencia (a pesar que esta configuración no garantiza los requerimientos temporales de la tarea periódica) no hay otra opción mas que asignarle mayor prioridad a la ISR que a la tarea periódica.

Puede observarse como durante la primera activación de la tarea t_2 en el instante 0 µs, esta sufre cuatro expropiaciones. Estas expropiaciones garantizan la regularidad de la ejecución de la ISR (de forma que no se pierden caracteres); sin embargo, las ISRs ejercen una perturbación en la ejecución de la tarea periódica de tiempo real crítico, que provocan que en el instante de tiempo 30000 µs la ejecución de la misma exeda su plazo. Esta situación (las expropiaciones y el incumplimiento resultante en los plazos) se mantiene en todas las demás activaciones de la tarea periódica. Obsérvese que para este conjunto de tareas, la única forma posible en que la tarea periódica de tiempo real t_2 pudiera cumplir sus plazos es que la misma no pueda ser expropiada por la ISR. Sin embargo, para que esto suceda la ISR debería tener menor prioridad que la tarea periódica lo cual es imposible con un esquema tradicional de administración de interrupciones.

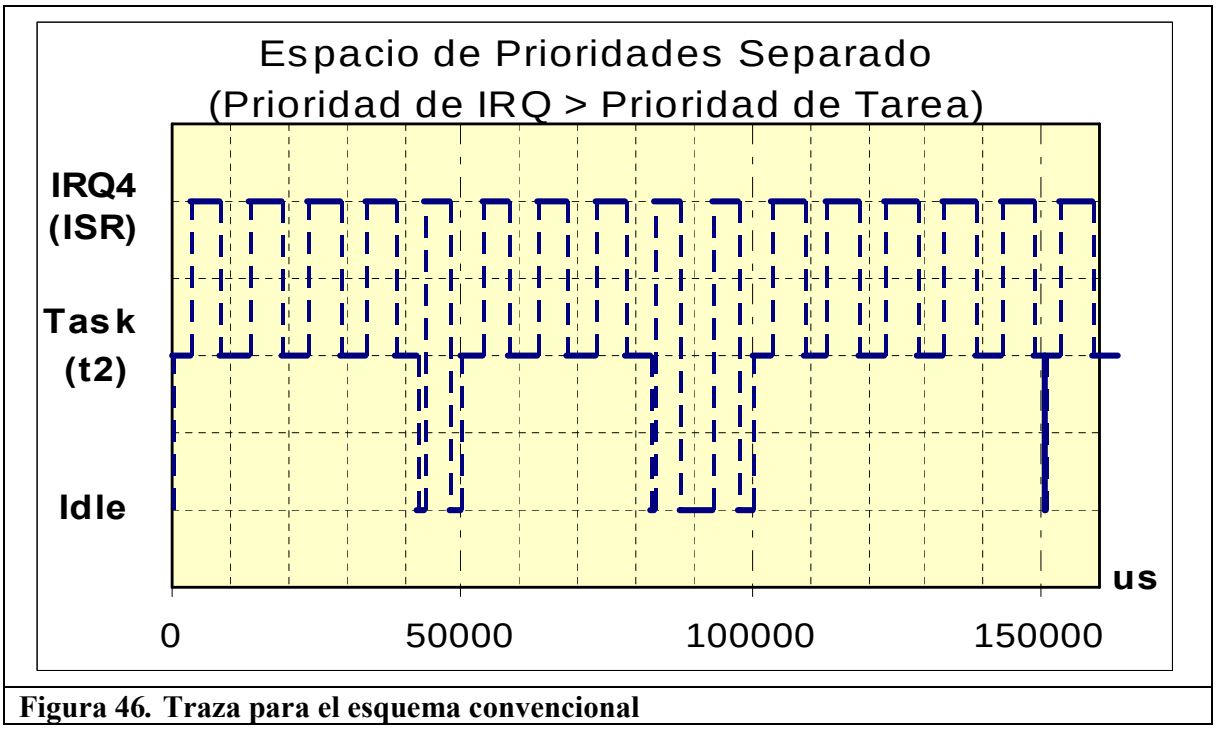


Figura 46. Traza para el esquema convencional

6.1.4 Comportamiento utilizando el Modelo Integrado

Con el modelo integrado, se tiene total flexibilidad para la asignación de las prioridades a las diferentes actividades en el sistema. Por tanto es posible utilizar la única configuración de prioridades que garantiza el cumplimiento de los plazos de la tarea periódica. En este caso t_1 se codificó como una HAT asociada a la IRQ4 y a esta se le asignó menor prioridad que a la tarea periódica.

La gráfica de la Figura 47 se corresponde con las trazas generadas por la ejecución de estas mismas dos actividades utilizando el esquema integrado aquí propuesto.

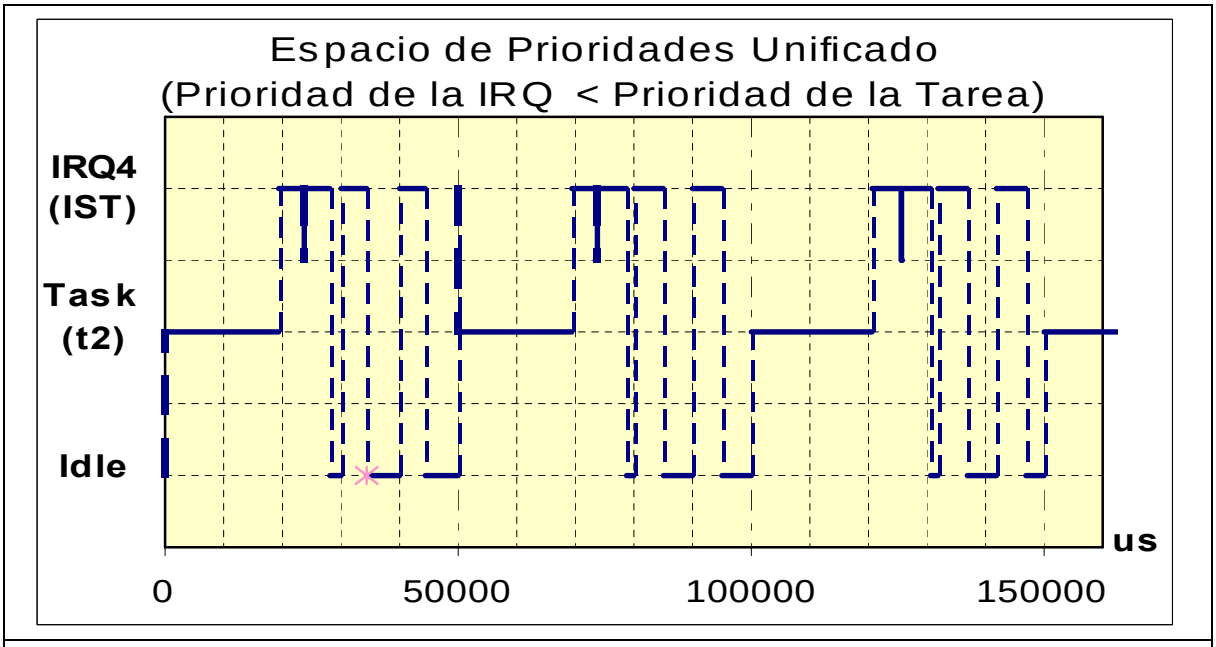


Figura 47. Traza para el esquema integrado

Al observar esta gráfica se destaca lo siguiente: (1) ahora t_2 se ejecuta sin ningún tipo de interferencia de la IRQ y en consecuencia todas sus instancias cumplen sus plazos (a los 30000 µs, 80 000 µs y 130 000 µs); (2) en el instante correspondiente a los 50000 µs la tarea periódica t2 expropia a la HAT poniendo de manifiesto que su instante de inicio no se ve postergado por la ejecución de la HAT y (3) en cada período de t_2 sólo se atienden 4 IRQ (en lugar de las 5 del caso convencional). Durante los 20ms del tiempo de cómputo de t_2 se emiten dos peticiones de interrupción (de menor prioridad que la tarea actual), que no provocan interrupción en la CPU. Sin embargo, el hardware de interrupciones "recuerda" una de ellas (la otra es ignorada). La capacidad del hardware de recordar una interrupción provoca que se agrupen la peticiones de interrupción luego de concluido el tiempo de respuesta de la tarea t_2 razón por la cual se ejecutan dos instancias consecutivas de la HAT a partir de los instantes 20 000 µs, 70 000 µs y 120 000 µs.

6.1.4.1 Comentario acerca de la pérdida de peticiones de interrupción

Vale la pena hacer algunos comentarios acerca de la pérdida de algunas señales de petición de interrupción que se observa en la Figura 47 y que es provocada por esta configuración de prioridades. El primero es que aquí tenemos un compromiso inevitable: como se vio en el análisis de factibilidad (de la sección 6.1.2) para este conjunto de tareas, el sistema no puede garantizar el procesamiento de todas las interrupciones y al mismo tiempo garantizar satisfacer los plazos de la tarea periódica de tiempo real crítico. De hecho, esta es la razón de ser de esta configuración de prioridades: garantizar los requerimientos de tiempo de aquellas tareas activadas por software que poseen requerimientos de tiempo real independientemente de la sobrecarga que podrían provocar aquellas tareas activadas por hardware y sin requerimientos de tiempo real crítico.

El segundo es que mediante el uso del análisis de tiempo real a través de todas las etapas del proceso de desarrollo de software, se puede asegurar que nunca se pierdan interrupciones cuando todas las fuentes de interrupción se comportan según se espera y que, al mismo tiempo, no afecten los requerimientos temporales de las tareas de tiempo real crítico cuando alguna de ellas introduce una sobrecarga no esperada.

El tercero es que la pérdida de interrupciones asociadas a tareas sin requerimientos de tiempo real crítico no es un problema. Por ejemplo, un puerto serie puede ser configurado para situar los datos en un buffer en hardware, el manejador de interrupción puede vaciar todo el buffer en vez de procesar un único carácter a la vez y el protocolo de comunicación de capa de enlace puede recuperar el dato perdido.

6.1.5 Comportamiento utilizando los Modos de Máscara Física y Máscara Virtual

En esta sección se contrasta el comportamiento del esquema integrado implementado con el modo de mascara física y el modo de máscara virtual. Se utiliza el mismo conjunto de tareas de los experimentos anteriores. Con esta configuración se realizaron dos corridas, ambas utilizando el modelo integrado de administración de interrupciones y tareas y asignándole a la tarea t_2 mayor prioridad que a la HAT t_1^S (única asignación que garantiza el cumplimiento de los plazos de la tarea periódica).

La diferencia entre ambos experimentos está dada solo porque en la primera corrida (mostrada en la Figura 48) se utilizó el modo de emulación con enmascaramiento físico, mientras que en la segunda (mostrada en la Figura 49) se utilizó el modo de emulación con enmascaramiento virtual. Independientemente del modo de emulación que se utilice es de esperar que el comportamiento de ambos experimentos sea muy similar, la única diferencia que podría esperarse está dada en la ejecución del LLIH del INTHAL. Por ello, para distinguir la diferencia de comportamiento entre ambos modos, ahora (además de las trazas de inicio y fin de ambas actividades) es necesario registrar también la ejecución del LLIH del INTHAL cada vez que se le pase como parámetros la IRQ asociada al puerto serie (y al HAT t_1^2).

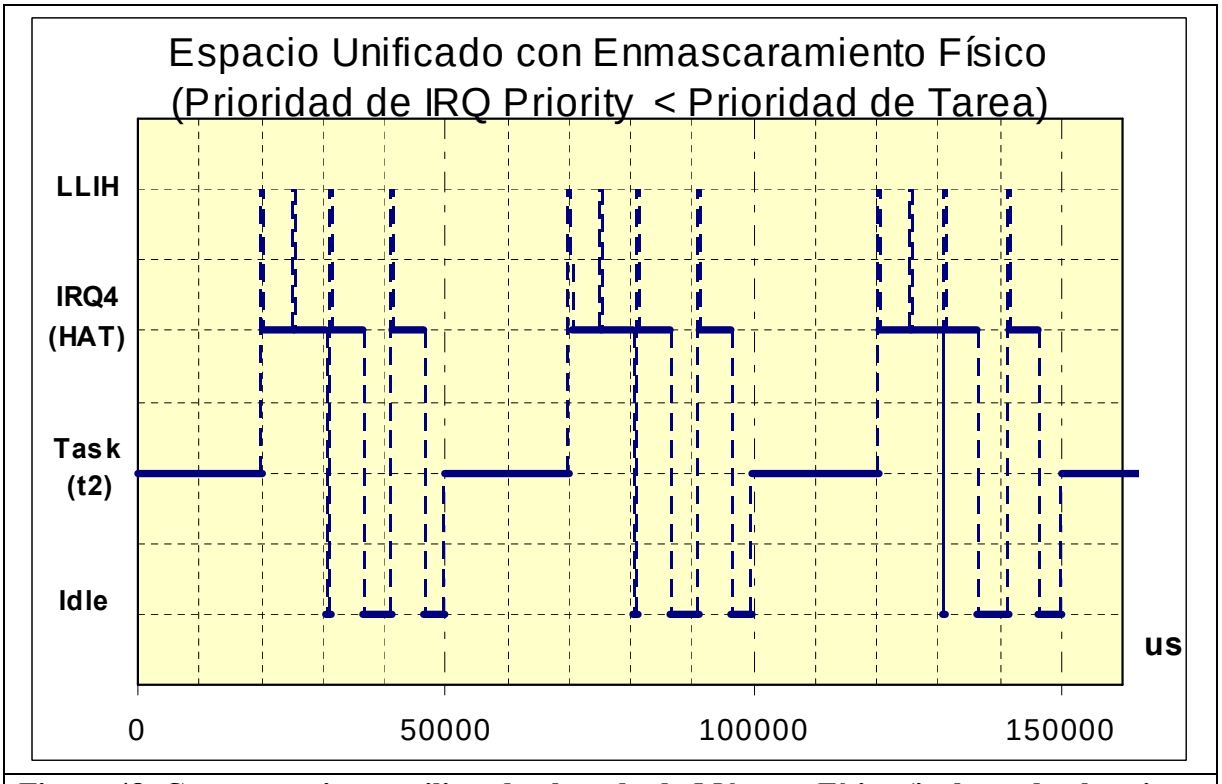


Figura 48. Comportamiento utilizando el modo de Máscara Física (incluyendo el registro de la ejecución del LLIH)

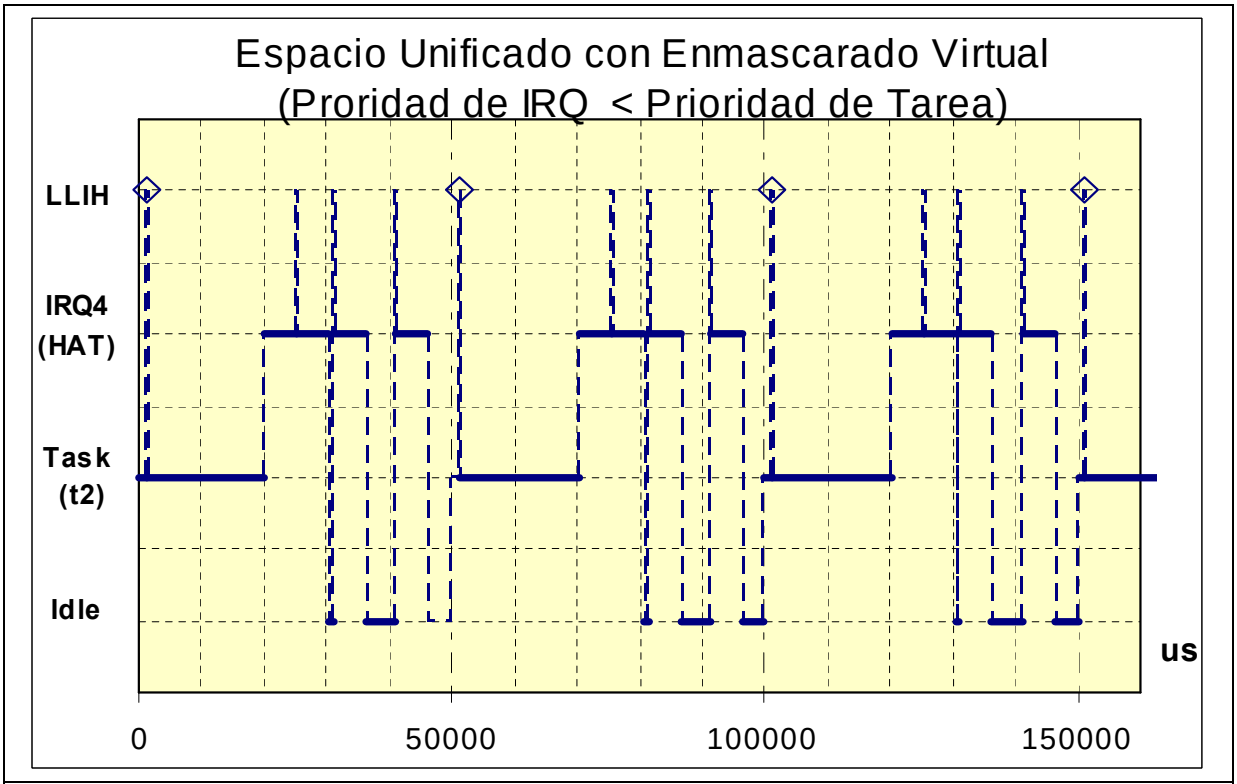


Figura 49. Comportamiento utilizando el modo de Máscara Virtual (registrando la ejecución del LLIH)

La Figura 49 muestra la traza de ejecución para el caso del modo de enmascaramiento virtual. En este caso vale la pena mencionar que la HAT asociada a la IRQ no puede expropiar a la tarea periódica t_2, por tanto, es similar al experimento previo, en donde la tarea t_2 puede finalizar antes de su plazo en todas las instancias (como se muestra en la Figura 49 a los 30000 µs, 80000 µs y 130000 µs). Sin embargo, en este caso hay una diferencia. Ahora la IRQ realmente expropia la ejecución de t_2. Esto se muestra en la figura por las trazas LLIH (destacadas con rombos) que ocurren un poco después de 0 µs, 50000µs, 100000µs y 150000µs. Observe que, en éstas la correspondiente HAT no se ejecuta y que, estas interrupciones no deseadas no son servidas en esos instantes de tiempo; sino que, en vez de ello son registradas (por el objeto de sincronización) hasta el final de la tarea periódica (en 20000µs, 70000µs y 120000µs). Esto se ilustra por las activaciones de la HAT correspondiente (donde no se ejecutan las trazas LLIH correspondientes).

En este caso es importante observar que sólo es posible una interrupción no deseada por activación de t_2. De nuevo, por cada período de t_2 sólo son aceptadas y manejadas 4 IRQs (en lugar de las 5 IRQs que fueron emitidas por el puerto serie). En cada ejecución de t_2 una IRQ es ignorada. Esta restricción garantiza una pequeña cota en la perturbación debido a interrupciones no deseadas como se expresó en la Ecuación 17 (ver sección 5.4.4).

Para una mejor comprensión, la tabla de la Figura 50 muestra una descripción detallada (punto por punto) de las similitudes y diferencias entre ambas ejecuciones.

Instante de tiempo/evento	**Figura 48 Enmascarado Físico (“*Physical Masking*”)**	**Figura 49 Enmascarado Virtual (“*Virtual Masking*”)**
Similitud Instante t = 0- (no mostrado)	Se está ejecutando la tarea ***idle***. El nivel de prioridad actual (y la máscara virtual y física) son tales que cualquier actividad (SAT o HAT) comenzará a ejecutarse inmediatamente luego de su activación.	
Similitud Instante t = 0s	Comienza la ejecución de la tarea periódica SAT t_2 (esta es la tarea con mayor prioridad y ocupa la CPU durante C_2 = 20 ms. La conmutación de contexto (de IDLE a t_2) eleva el nivel de prioridad actual	
Diferencia Instante t = 0s	Durante la conmutación de contexto se actualiza la máscara física (en el 8259) dejando enmascarada a la IRQ4 .	Como parte de la conmutación de contexto no se actualiza la máscara física (en el 8259). Sólo se actualiza la máscara lógica. Por tanto IRQ4 continúa desenmascarada (puede ocurrir de forma indeseada).
Diferencia Poco después de 0s se emite la 1ra IRQ4	La CPU no es interrumpida (no hay traza en LLIH). Ello se debe a que la IRQ4 está enmascarada (tiene menos prioridad que la tarea t_2)	La CPU es interrumpida (es por eso que está la traza en LLIH). Ello se debe a que, a pesar de que t_2 tiene más prioridad que IRQ4, esta última está enmascarada en la máscara lógica pero no en la máscara física. Hay que destacar que aquí sólo se ejecuta el LLIH pero (al igual que con el enmascaramiento físico) no se ejecuta el manejador de la IRQ4. No hay traza en IRQ (HAT). Esto último se

		debe a que la IRQ es no deseada. Por tanto el LLIH hizo el signal en el semáforo pero como la HAT tiene menos prioridad que t2 no expropia.
Similitud Poco después de 10 ms, se emite la 2da IRQ4	La CPU no se interrumpe. Se debe a lo mismo que en el caso anterior. Aquí se acumularon dos IRQ4 ignoradas por lo tanto una se pierde definitivamente. El hardware (8259) recuerda una sola	A diferencia de lo que ocurrió con la IRQ anterior. Ahora el comportamiento es igual que con enmascaramiento físico. O sea, la CPU no es interrumpida por la IRQ (LLIH). Ello se debe a que en este caso cuando ocurrió la 1ra IRQ el LLIH estableció realmente la máscara física que se corresponde con la prioridad actual (máscara lógica) por tanto no se producen más interrupciones no deseadas mientras no disminuya el nivel de prioridad actual.
Diferencia Poco después de 20ms, se emite una 3ra IRQ	Ya terminó la ejecución de la SAT t2 (C2 = 20ms) y la conmutación de contexto baja el nivel de prioridad (desenmascara IRQ4) e inmediatamente esta llega a la CPU (traza LLIH) y se ejecuta su manejador durante 5ms (traza IRQ4 HAT)	Termina la ejecución de la SAT t2 pero ahora lo que se produce es una conmutación de contexto al HAT de la IRQ4 que estaba listo (traza IRQ4 HAT). A diferencia del caso anterior, la 3ra IRQ4 emitida aquí no llega a la CPU (No hay traza LLIH). Ello se debe a que la HAT recién activada tiene igual prioridad que dicha IRQ4, por tanto no se desenmascara la IRQ4. Aquí ahora se acumulan dos IRQ ignoradas por tanto una se pierde definitivamente.
Igualdad Poco después de 25ms.	Aunque en este justo instante el puerto serie no emite ninguna IRQ, la terminación de la HAT hace que se baje el nivel de prioridad nuevamente y se desenmascare la IRQ con lo que le llega a la CPU la IRQ previa que el 8259 estaba "recordando" (traza en LLIH y luego en IRQ4-HAT)	
Igualdad Poco después de 30 ms, se emite la 4ta IRQ	Ya había terminado la ejecución de la HAT de IRQ4 y la CPU estaba en IDLE (IRQ4 desenmascarada) por tanto inmediatamente la CPU es interrumpida (traza LLIH) y se ejecuta el manejador de la IRQ4 (traza IRQ4-HAT)	
. . .		
Instante 50 ms	**Se repite el ciclo**	

Figura 50. Descripción detallada de las similitudes y diferencias del comportamiento con enmascaramiento físico y virtual.

6.2 Caracterización del Desempeño del Subsistema de Interrupciones

En esta sección se describen las pruebas realizadas para caracterizar la sobrecarga impuesta por la emulación del mecanismo integrado de manejo de interrupciones. Esta caracterización implica la medición de un conjunto de parámetros relacionados con el tiempo de manejo de interrupciones. En especifico, nuestras pruebas midieron la latencia de interrupción y el incremento en el tiempo de conmutación de contexto debido al uso del modelo integrado.

6.2.1 Latencia de Interrupción

Típicamente el procesamiento de una interrupción se divide en varias fases. Sin embargo, los términos utilizados para referirse a la duración de estas fases varían según los diversos autores. A continuación se definen varios parámetros de la respuesta a interrupción, todos referidos a la interrupción de más prioridad en un sistema, ya que las interrupciones de menos prioridad van a estar afectadas por la ocurrencia de las más prioritarias. En el caso particular del modelo integrado, donde las interrupciones comparten un único espacio de prioridades con las tareas y pueden tener un orden de prioridad distinto al impuesto por el hardware, los términos definidos a continuación se refieren a la interrupción asociada a la HAT de más prioridad en el sistema, asumiendo además que ninguna otra tarea tiene más prioridad que ella.

Al tiempo transcurrido desde que ocurre una petición de interrupción hasta que se le da servicio le denominaremos *tiempo de respuesta*. El tiempo de respuesta varía según el tiempo que tarda el código del usuario en procesar la interrupción.

Se denomina *latencia de interrupción* al tiempo que tarda un sistema en comenzar a dar servicio a una interrupción. Esta se define como el tiempo transcurrido desde que se activa la IRQ hasta que se ejecuta la primera instrucción del código manejador de interrupción establecido por el usuario; que puede ser una ISR de usuario en el caso de los sistemas en los cuales la atención a interrupción se hace a este nivel o una HAT en el caso del modelo integrado.

6.2.2 Prueba para medir la latencia de interrupción

El objetivo de estas pruebas es determinar cuanto tarda el núcleo en conmutar a una tarea activada por hardware una vez ocurrida su interrupción asociada; en otras palabras, determinar la latencia de despacho de interrupción del núcleo. Existen varios métodos para medir esta latencia. El método utilizado por nosotros se describe a continuación

6.2.2.1 Método de Manejador de Interrupción (HAT) y Tarea de Fondo.

Uno de los métodos que puede ser utilizado para medir la latencia de interrupción consiste en usar una tarea de fondo baja prioridad que envía trazas en ciclo infinito a un analizador lógico y un manejador de interrupción de alta prioridad que lo único que realiza es enviar otra traza con código distinto.

Este método puede ser adaptado para utilizar un registro de bitácora ("*logger*") de las trazas por software, y el canal 0 del 8253 como fuente de interrupciones. Esta prueba no produce resultados muy exactos en el caso de implementarse en una máquina 80486 debido a que tiene que utilizarse el contador 2 del chip 8253 para medir el tiempo; pero sirve para comparar los resultados con otras pruebas más exactas, permitiendo detectar alguna anomalía en los tiempos medidos (indicando algún error en la implementación de la prueba). Para el caso de sistemas Pentium y posteriores puede utilizarse la instrucción RDTSC ("*Read from Time Stamp Counter*") la cual copia el contenido del registro contador de tiempo (TSC) del Pentium (contador de 64 bits que se incrementa por cada ciclo de reloj) en EDX:EAX.

6.2.3 Resultados de las Mediciones de Desempeño

6.2.3.1 Incremento en el Tiempo de Conmutación de Contexto

El incremento en el tiempo de conmutación de contexto δ^M está dado por el tiempo de ejecución en el peor caso del servicio **setIRQLevel(priority)** responsable de obtener y establecer el registro IMR. Es importante destacar que este servicio sólo tiene que escribir realmente la máscara en los casos en que un cambio en el nivel de prioridad actual implique la inhabilitación o habilitación de una determinada IRQ.

La Figura 51 muestra los tiempos obtenidos para ambas plataformas en las que se destacan los valores de δ^M de 1.04294µs y 2.67849µs para el Pentium4-3GHz y el Pentium MMX-200MHz respectivamente.

Tiempos en micro-segundos	P4 - 3 GHz		P-MMX 200MHz	
	Sin escritura de IMR	**Con escritura de IMR**	**Sin escritura de IMR**	**Con escritura de IMR**
Mínimo	0.05781	0.87218	1.06140	1.91750
Average	0.06249	0.90800	1.06140	2.04367
Máximo	0.09557	**1.04294**	1.14650	**2.67849**

Figura 51. Valores de δ^M o tiempos de ejecución de la llamada a setIRQLevel()

La Figura 53 presenta estos resultados de forma gráfica para el procesador Pentium4-3GHz, mientras que la Figura 54 lo hace para el procesador PentiumMMX-200. Puede apreciarse cómo el tiempo de ejecución en el peor caso está dominado por el establecimiento de la máscara. Este contraste es más apreciable en el caso de las arquitecturas más modernas (ver la gráfica para el caso del Pentium4 en la Figura 53). En este caso, el componente de determinación del valor apropiado de la máscara (necesario para hacer coincidir los espacios de las prioridades de software del algoritmo de planificación con las prioridades de hardware del controlador de interrupciones) se lleva a cabo prácticamente sin costo alguno.

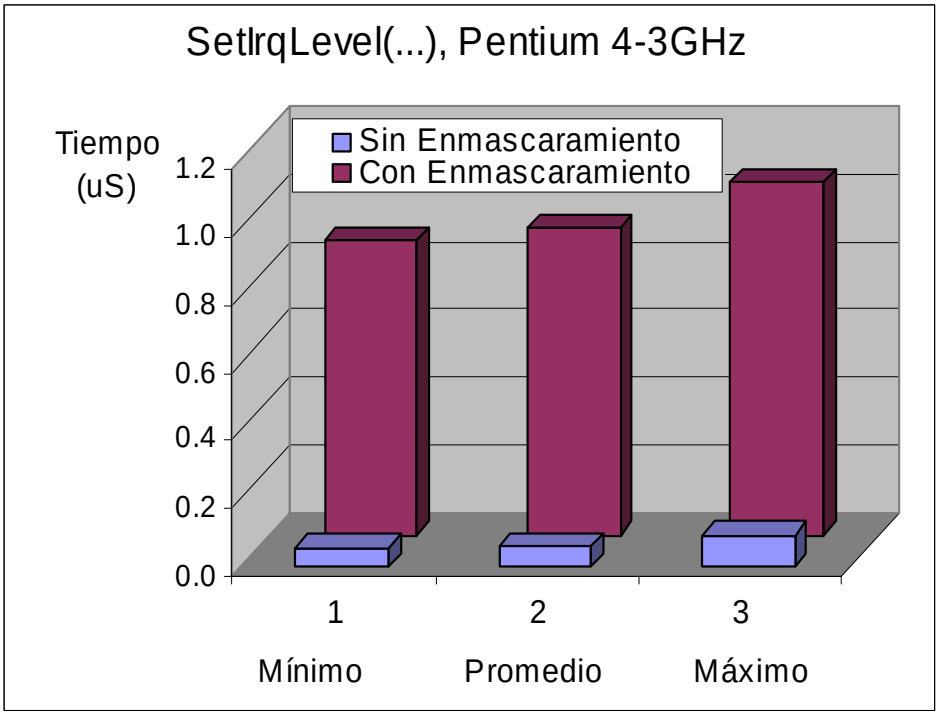


Figura 52. Gráfica de los valores de δ^M según modo de operación para un procesador Pentium 4 – 3GHz

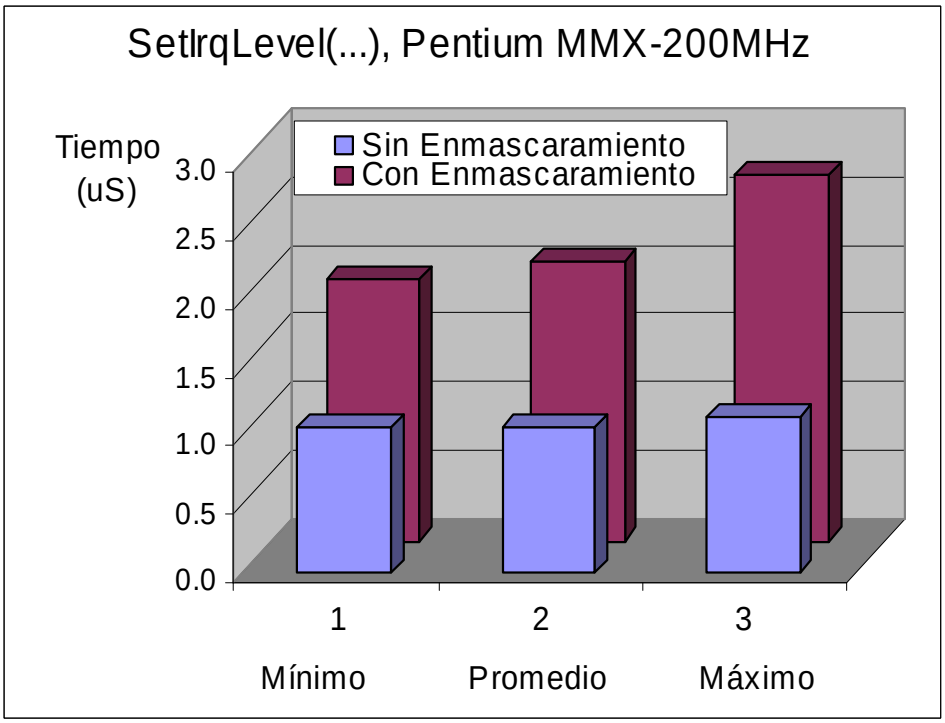


Figura 53. Gráfica de los valores de δ^M según modo de operación para un procesador Pentium MMX – 200 MHz

	Tiempo de conmutación de contexto (µs)			Costo relativo
Sistema Operativo	**Mínimo**	**Promedio**	**Máximo**	**(%)**
Hyperkernel 4.3	2.32	5.64	266.2	1.0%
RTX 4.1	4.29	4.59	10.68	25.1%
INTime 1.20	4.8	5.47	23.13	11.6%
Windows CE 2.0	2.4	-	34.4	7.8%
QNX 6.1	2.1	3.3	11.5	23.3 %
Figura 54. Costo relativo de implementar el modelo integrado				

La Figura 54 muestra valores de los tiempos de conmutación de contexto de varios sistemas operativos o extensiones de tiempo real comerciales reportados en la literatura para un sistema Pentium MMX a 200MHz [106] y que nos permiten evaluar cuál sería el costo relativo de la implementación del esquema integrado en los mismos para dicha plataforma.

6.2.3.2 Latencia de Interrupción del Kernel

La Figura 55 muestra los resultados de la medición de la latencia de Interrupción en un sistema Pentium4 a 2.8GHz mediante el método de tarea de fondo y HAT (descrito en 6.2.2.1) y utilizando el TSC. Se realizaron 1000 mediciones (que como se verá en el siguiente párrafo para el caso de la arquitectura integrada constituye una cantidad más que suficiente).

Como puede observarse en la gráfica, en todos los modos el comportamiento de la latencia de interrupción es prácticamente estable alrededor de la latencia promedio: 4.869µs (para EOI con enmascaramiento físico), 4.158 µs (para AEOI con enmascaramiento físico), 3.469µs (para EOI con enmascaramiento virtual) y 2.716µs (para AEOI con enmascaramiento virtual). Este elemento es un factor muy importante para sistemas de tiempo real ya que dota de predecibilidad al sistema. Obsérvese además que estos resultados experimentales, que nos muestran una latencia de interrupción prácticamente constante, son completamente esperados cuando se utiliza el modelo integrado debido a que: (1) el núcleo se ejecuta con interrupciones habilitadas durante todo el tiempo (sólo se inhabilitan en unas pocas secciones críticas muy bien identificadas dentro de la implementación del mecanismo de sincronización); (2) el servicio a todas las interrupciones siempre atraviesa por una trayectoria de ejecución idéntica (el LLIH del INTHAL, el manejador **IRQHandler**() del núcleo que invoca al planificador); (3) el empleo de un algoritmo de activación y planificación que presenta un tiempo de ejecución constante. La figura muestra una variabilidad muy pequeña y unos picos que sólo pueden ser provocados por la latencia de interrupción del mismo hardware (y por tanto son componentes inevitables de la arquitectura de la máquina e independientes del sistema operativo y esquema de manejo de interrupciones – por ejemplo el caché o la canalización del procesador).

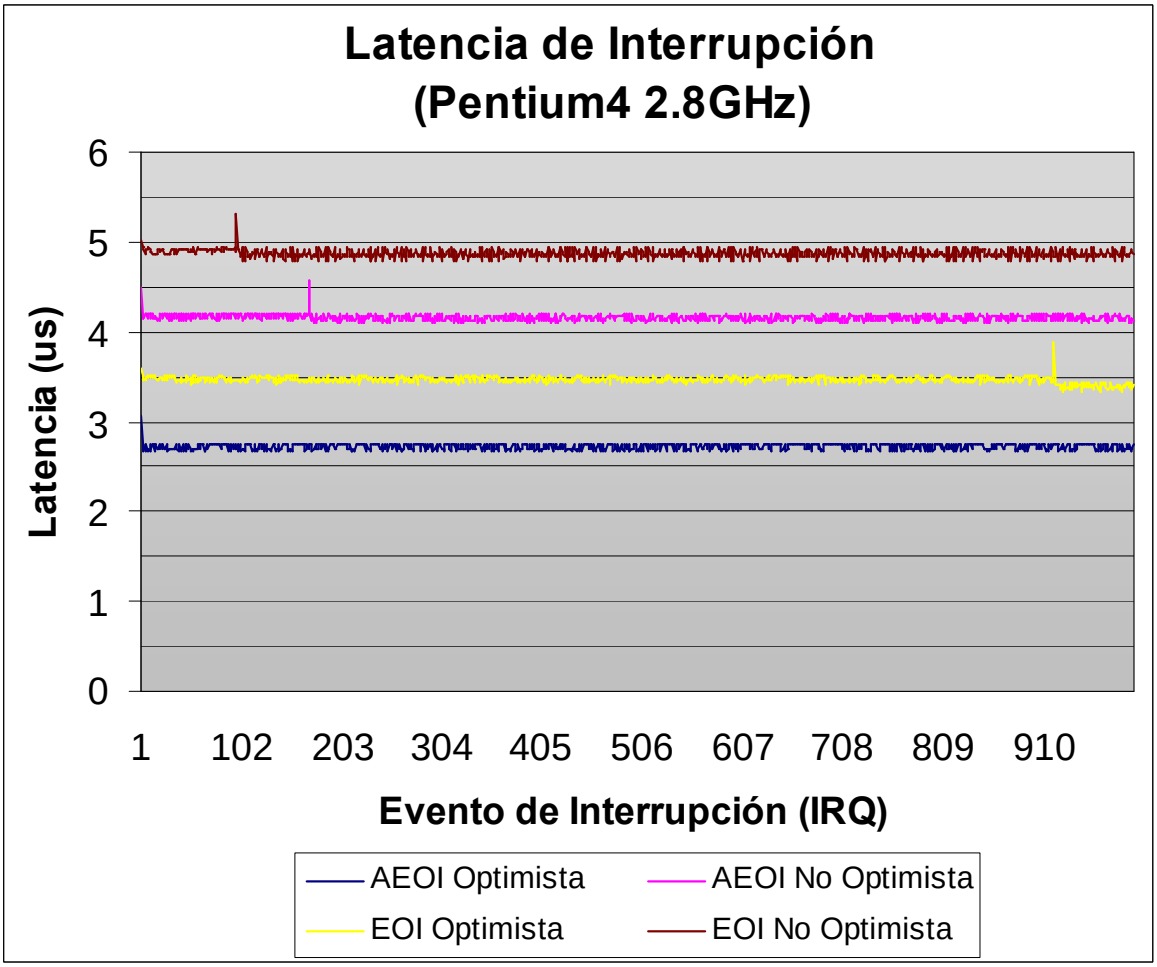


Figura 55. Latencia de Interrupción para un Pentium4 a 2.8GHz según el modo de emulación.

El otro elemento a destacar de la gráfica es que los valores para los modos de enmascaramiento virtual son un 72.9% y un 67% de los valores correspondientes para los modos pares con enmascaramiento físico. Esto representa una reducción significativa en la sobrecarga operativa del modelo integrado (con respecto al la sobrecarga que introduce el enmascaramiento físico). Esta reducción, debida al uso del enmascaramiento virtual, hace al modelo integrado factible incluso para aquellos sistemas con requerimientos de baja sobrecarga. Puede observarse además que los dos modos con enmascaramiento virtual exhiben mejores resultados que los dos modos de enmascaramiento físico. Como podría esperarse, el modo con EOI automático y enmascaramiento virtual muestra el mejor desempeño con un valor muy bajo de latencia de interrupción en el peor caso de 3.05 µs.

Es interesante analizar cómo se relacionan los resultados de la latencia de interrupción para cada modo de emulación con el número de operaciones de escritura/lectura a puerto para dicho modo. La tabla de la Figura 56 muestra esta relación. Como se puede observar, el factor dominante en el valor de la latencia de interrupción es el número de accesos a puertos (para el envío del EOI y la escritura de la Máscara).

Modo de Emulación		Escrituras a Puertos			Latencia de Interrupción (µs)		
Modo EOI	**Modo de Máscara**	**EOI**	**IMR**	**# total de OUT**	**Mínimo**	**Promedio**	**Máximo**
Explícito Normal	**No optimista**	1	2[1]	**3**	4.781 µs	4.869 µs	5.318 µs
Explícito (Normal)	**Optimista**	1	0[2]	**1**	3.340 µs	3.469 µs	3.877 µs
Auto-mático	**No optimista**	0	2[1]	**2**	4.097 µs	4.158 µs	4.564 µs
Auto-mático	**Optimista**	0	0[2]	**0**	2.672 µs	2.716 µs	3.060 µs

1. Se supone que ha sido necesario establecer los IMR de ambos 8259. En realidad muchas veces sólo es necesario establecer un solo IMR. Sin embargo, en la implementación con la que se hicieron estas mediciones siempre se establecen los registros IMR de ambos 8259 (pero caso).
2. Este valor supone que la IRQ no es una interrupción indebida. Para el caso de que sea una interrupción indebida el LLIH escribe una o ambas máscaras. Sin embargo, en este caso ello no se contabiliza en la latencia debido a que tampoco se invocará a la IST.

Figura 56. Relación entre la latencia de interrupción y los accesos a puertos.

6.3 Evaluación de los Resultados Experimentales

Como un comentario final de estos experimentos, al comparar la sobrecarga de acceso a puertos del esquema integrado con respecto a la del esquema tradicional vale la pena observar lo siguiente:

1. En el caso del esquema tradicional la latencia de interrupción no incurre en la sobrecarga de ningún acceso a puerto; sin embargo, la latencia de salida de la ISR si incurre en esta sobrecarga debido a la necesidad de enviar el comando de fin de interrupción. En consecuencia si aparece reflejada la sobrecarga de acceso a puerto en la interferencia (o perturbación).

2. En el caso del modelo integrado utilizando el modo de EOI automático se incluye el acceso a puerto en la latencia de interrupción (debido a la necesidad de situar el IMR); sin embargo, se elimina por completo el envío del EOI. En consecuencia, como las escrituras a puerto son el elemento dominante en el tiempo de ejecución, es de esperar que la sobrecarga que se introduce, sea del mismo orden.

3. Si se utiliza el modo de EOI automático combinado con el enmascaramiento virtual entonces (para las interrupciones debidas) no se realizará ninguna escritura a puerto ni como parte de la latencia de interrupción, ni como parte de la latencia de salida de la HAT. En consecuencia es de esperar que la sobrecarga introducida por el modelo integrado sea incluso menor que la sobrecarga del modelo convencional debido a la necesidad de gestionar de forma explícita el fin de interrupción.

En conclusión, como se demuestra con la ecuación 17, las corridas de validación y las mediciones de desempeño, la implementación del modelo integrado utilizando el modo de EOI automático y el enmascaramiento virtual permite un esquema de manejo de interrupciones completamente predecible o temporalmente determinista, sin incurrir en sobrecarga o incluso con sobrecargas inferiores al esquema de administración convencional.

7 Conclusiones

7.1 Conclusiones

Los detalles de implementación del manejo de interrupciones poseen un impacto dramático en el diseño de los mecanismos de sincronización y cómo se utilizan. Como consecuencia de la separación entre ISRs y tareas, aparecen severas restricciones en cuanto a los servicios del sistema que se pueden invocar dentro de los primeros. Lo que trae como consecuencia un aumento de la complejidad de diseño e implementación, que afecta la confiabilidad del software resultante. Adicionalmente, la generalidad de los desarrollos teóricos para los análisis de factibilidad de planificación de los sistemas de tiempo real, consideran sólo un único espacio de prioridades para todas las actividades en el sistema. Esta suposición contrasta con el modelo real soportado por el sistema operativo, en el cual las ISRs y las Tareas poseen espacios de prioridades y algoritmos de planificación independientes. Como consecuencia, el empleo de dos espacios de prioridades independientes afecta severamente la capacidad de predecir el comportamiento temporal del sistema. En los casos en que las ecuaciones de factibilidad tienen el cuenta el efecto de estos dos espacios, se deteriora significativamente la cota de utilización para que sea factible la planificación.

Muchos sistemas operativos de tiempo real han intentado mantener este modelo introduciendo soluciones que mejoran el determinismo con la instrumentación de un tratamiento de eventos externos en dos o incluso más niveles, cada uno con determinados grados de velocidad de respuesta, restricciones de sincronización y afectación del determinismo temporal del sistema (secciones 3.3.3 y 3.4.3). Estos esquemas pueden combinarse con varios métodos de análisis de factibilidad que tienen en cuenta las interrupciones como las actividades de mayor prioridad en el sistema (sección 3.4.5). Sin embargo, todas estas soluciones comprometen la eficiencia y aumentan aún más la complejidad del mecanismo de sincronización entre los distintos tipos de actividades de interrupción y las tareas; todo ello, sólo para disminuir la interferencia porque nunca se consigue un comportamiento verdaderamente determinista.

En un enfoque totalmente opuesto, varios sistemas operativos de tiempos de tiempo real incluso han adoptado soluciones radicales para eliminar la interferencia producto de las interrupciones consistente en el tratamiento de los dispositivos de E/S mediante encuesta (sección 3.4.4). Aunque esta solución evita completamente el no-determinismo asociado a las interrupciones, tiene como desventaja fundamental una baja eficiencia en el uso de la CPU en operaciones de E/S, debido a la espera ocupada de las tareas mientras acceden a los registros del dispositivo. Sin embargo, aunque las interrupciones constituyen un impedimento al determinismo temporal del sistema, si se quiere lograr una mejor utilización de la CPU no debiera prescindirse completamente de ellas.

En este trabajo hemos presentado el modelo integrado o unificado de tratamiento de interrupciones y tareas. Aunque previamente se han propuesto varias estrategias que logran algún grado de integración entre los diferentes tipos de actividades asíncronas (Sección 3.3); ninguna de ellas incluye la planificación (de las peticiones de interrupción) del hardware. La ventaja de este modelo integrado, es que permite lograr el determinismo temporal necesario para las aplicaciones de tiempo real, sin afectar significativamente la utilización de la CPU.

Hemos presentado el diseño de un subsistema de administración de interrupciones de bajo nivel que puede ser utilizado en cualquier sistema operativo que quiera soportar este modelo integrado. El diseño considera un alto grado de independencia tanto del hardware como del modelo de concurrencia y sincronización propio del núcleo del sistema. Este subsistema puede ser implementado de diversas formas incluso en sistemas con arquitecturas de interrupciones que no soportan de forma directa el modelo integrado. En nuestro trabajo hemos realizado una implementación sobre el hardware de interrupciones estándar de los sistemas PC compatibles. En específico se presentaron varios modos de emulación todos ellos predecibles pero con diferentes compromisos entre un mejor comportamiento en el peor caso y un mejor comportamiento promedio. Para cada uno de los modos, se mostraron los algoritmos de emulación y las ecuaciones que modelan el comportamiento de temporal y la sobrecarga. Todos los modos de emulación se implementaron dentro de un micro-núcleo experimental diseñado para este propósito

Utilizando nuestro micro-núcleo, se obtuvieron evidencias experimentales que demuestran el comportamiento predecible y, en el caso del modo de emulación con enmascaramiento virtual, sin sobrecarga.

Hemos presentado el diseño de un subsistema de administración de interrupciones de bajo nivel que soporta este modelo integrado. La implementación de este subsistema para la arquitectura de las PCs, nos ha permitido experimentar en la práctica, la factibilidad del modelo integrado propuesto.

Consideramos que el modelo integrado propuesto en este trabajo constituye un aporte significativo a la teoría y práctica de la implementación de sistemas de tiempo real al permitir adecuar los mecanismos de implementación de los sistemas de tiempo real de manera que se correspondan con los desarrollos de la teoría de planificación de tiempo real. Como consecuencia, esta propuesta es un paso importante hacia el logro de una transición sin costuras desde el modelo de análisis (basado en la teoría de planificación de tiempo real) del sistema, a su implementación real.

7.2 Trabajo Futuro

La motivación inicial de este trabajo fue hacer un estudio de los mecanismos causantes de impredecibilidad existentes en los sistemas de cómputo actuales y proponer mecanismos que permitan eliminar esta impredecibilidad. Esto se puede lograr en con dos estrategias complementarias: introducir modelos analíticos que permitan caracterizar el comportamiento temporal de los mecanismos existentes y modificar los mecanismos actualmente existentes de forma tal que puedan ser susceptibles al modelado. Entre estos

mecanismos se encuentran aquellos asociados a la arquitectura de la computadora, tales como las interrupciones, la memoria caché y el acceso directo a memoria, y mecanismos asociados con la propia arquitectura de software del sistema operativo como el mecanismo de administración de temporizadores.

Esta tesis estuvo enfocada al mecanismo de interrupciones y propuso un nuevo esquema de manejo que permite eliminar la impredecibilidad debido a estas. Sin embargo, quedan por abordar otros mecanismos que introducen impredecibilidad al sistema tales como el DMA o el manejo de temporizadores. Las estrategias para abordar el problema en cada caso es diferente y constituyen una de las áreas más importantes para el trabajo futuro.

En el caso específico del esquema propuesto en esta tesis, aunque no está atado a ningún esquema de planificación, la implementación que se utilizó en nuestro núcleo estuvo basada en prioridades estáticas. Esta decisión estuvo basada en el hecho de que la planificación con prioridades estáticas está contemplada en todos los estándares actuales para sistemas operativos y lenguajes de programación de tiempo real y constituye el esquema utilizado en la práctica por la generalidad de los sistemas operativos de tiempo real actuales. Sin embargo, como trabajo futuro puede contemplarse la implementación de un esquema de planificación de prioridades dinámicas tal como EDF.

Otra vertiente para trabajo futuro es el diseño de un hardware especializado que permita la implementación de este esquema directamente sobre el hardware (por ejemplo utilizando FPGAs). Como parte de este trabajo se podría contrastar la implementación nativa en hardware con cada uno de los esquemas de emulación propuestos e implementados en esta tesis. De especial interés puede ser comparar la implementación nativa contra la implementación utilizando enmascaramiento virtual y fin de interrupción automático (la variante de emulación más eficiente en el caso promedio).

Otra dirección para el trabajo futuro consiste en implementar el modelo integrado sobre sistemas operativos de tiempo real establecidos comercialmente tales como versiones de Linux para tiempo real, QNX o Windows CE. El propósito sería que la atención a la E/S de todos los dispositivos se haga mediante el empleo del modelo propuesto.

REFERENCIAS

[1] Luca Abeni, Giorgio Buttazo, "***Support for Dynamic QoS in the HARTIK Kernel***", Proceeding of the IEEE Conference on Real-Time Computing Systems and Applications, Cheju Island, South Korea, 2000.

[2] Luca Abeni, Ashvin Goel, Charles Krasic, Jim Snow, Jonathan Walpole "***A Measurement-Based Analysis of the Real-Time Performance of Linux***", Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02), 2002.

[3] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, Michael Young, "***Mach: A New Kernel Foundation for UNIX Development***", 93-113, *USENIX Association Conference Proceedings*, USENIX Association, June 1986.

[4] N.C. Audsley, A. Burns, M.F. Richardson and A.J. Wellings, ''***Hard Real-Time Scheduling: The Deadline Monotonic Approach***'', Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software, Atlanta, GA, USA (15-17 May 1991).

[5] George Anzinger y Nigel Gamble (MontaVista Software), "***Design of a Fully Preemptable Linux Kernel***", Embedded Linux Journal. September 6, 2000.

[6] F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser, "***Chorus Distributed Operating Systems***", M. Rozier, V. Abrossimov, 305-370, *USENIX Computing Systems, 1, 4*, Fall 1988.

[7] N. C. Audsley, A. Burns, M. F. Richardson y A. J. Wellings, "***Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling***". Software Engineering Journal, , 8(5), 1993.

[8] John Barnes, "***Programming in Ada 95***", Addison-Wesley, Reading, MA. 1995.

[9] John Barnes (Ed.), "***Ada 95 Rationale:The Language, The Standard Libraries***", Lecture Notes in Computer Science 1247, Springer-Verlag, Berlin 1997.

[10] S. K. Baruah, L. E. Rosier, and R. R. Howell. "***Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor***". Journal of Real-Time Systems, 2, 1990.

[11] M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner. "***Linux Kernel Internals***". Addison-Wesley, 1998. ISBN 0-2011-33143-8.

[12] Greg Bollella, Ben Brosgol, Peter C. Dibble, Steve Furr, James Gosling, Davis Hardin, Mark Turnbull, Rudy Belliardi, Doug Locke, Scott Robbins, Patrik Solanki, Dionisio de Niz, "***The Real-Time Specification for Java***", Addison-Wesley 2000.

[13] Per Brinch-Hansen, "***The Nucleus of a Multiprogramming System***", Communication of the ACM, Volume 13, Number 4 (1970), pages 238-241 and 250.

[14] Per Brinch-Hansen, "***Operating System Principles***", Prentice Hall (1973).

[15] Per Brinch-Hansen, "***The Evolution of Operating Systems***", In [16]

[16] Per Brinch-Hansen (Editor), "***Classic Operating Systems: From Batch Processing to Distributed Systems***", Springer-Verlag, New York, 2000.

[17] Dennis Brylow Niels Damgaard Jens Palsberg, "***Static Checking of Interrupt-driven Software***", Proceedings of ICSE'01, International Conference on Software Engineering, pages 47–56, 2001.

[18] Alan Burns, Andy J. Wellings, "***Implementing Analyzable Hard Real-time Sporadic Tasks in Ada 9X***", ACM Ada Letters, Volume X/V, Number .1 Jan/Feb 1994.

[19] G. C. Buttazzo. "***Hartik: A real-time kernel for robotics applications***". Proceedings of the IEEE Real-Time Systems Symposium, Diciembre 1993.

[20] G. C. Buttazzo y M. Di Natale. "***Hartik: a hard real-time kernel for programming robot tasks with explicit time constraints and guaranteed execution***". Proceedings of the IEEE International Conference on Robotics and Automation, Mayo 1993.

[21] Douglas Comer, Timothy V. Fossum, "***Operating System Design, Vol.1: The Xinu Approach (PC Edition)***", Prentice-Hall, 1988

[22] J.O. Coplien, N. Kerth, J. Vlissides (eds.): "***Pattern Languages of Program Design 2***", (PLoPD2) Addison-Wesley, 1996 (a book publishing the reviewed Proceedings of the Second International Conference on Pattern Languages of Programming, Monticello, Illinois, 1995).

[23] Manuel Coutinho, José Rufino, Carlos Almeida, "***Control of Event Handling Timeliness in RTEMS***", Proceedings of the 17th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2005), Phoenix, AZ, USA, November 14 - 16, 2005.

[24] Helen Custer, "***Inside Windows NT***", Microsoft Press, 1992.

[25] A. Damm, J. Reisinger, W. Schwabl, y H. Kopetz. "***The Real-Time Operating System of MARS"***. Operating System Review, 23(3): 141-157, Julio 1989.

[26] Kevin Dankwardt, "***Real Time and Linux,(Part 1, 2 and 3)***", January/ February/ March 2002 issue of Embedded Linux Journal. 2002.

[27] Uwe Dannowski, Espen Skoglund, Volkmar Uhlig; "***Interrupt Handling***", Proceedings of the Second Workshop on Microkernel-based Systems, 2001.

[28] Peter C. Dibble. "***Real-Time Java Platform Programming***". Prentice Hall, 2002.

[29] Steven-Thorsten Dietrich, Daniel Walker, "***The Evolution of Real-Time Linux***", 7th Real-Time Linux Workshop: The Evolution of Real-Time Linux, Nov. 17, 2005.

[30] E. W. Dijkstra, "***The Structure of the THE Multiprogramming System" Communications of the ACM***, Vol. 11, No. 5 (May 1968), 341-346.

[31] J. R. Eykholt, S.R. Kleiman. S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, D. Williams, "***Beyond Multiprocessing: Multithreading the SunOs Kernel***", Proceedings of the summer 1992 USENIX Technical Conference, 1992.

[32] Bill O. Gallmeister, "***POSIX.4: Programming for the Real World***". O´Reilly Associates, Inc. 1995.

[33] Mike Hall, "***Windows CE 5.0 for real-time systems***", Embedded Computing Design, November 2005.

[34] David Harel and A. Pnueli. "***On the development of reactive systems***". Logics and models of concurrent systems, pages 477–498, 1985.

[35] Arnd C. Heursch, Alexander Horstkotte and Helmut Rzehak, "***Preemption concepts, Rhealstone Benchmark and scheduler analysis of Linux 2.4***", on the Real-Time & Embedded Computing Conference, Milan, November 27-28, 2001.

[36] Arnd C. Heursch, Dirk Grambow, Dirk Roedel and Helmut Rzehak, "***Time-critical tasks in Linux 2.6, concepts to increase the preemptablity of Linux kernel***", Linux Automation Konferenz, University of Hannover, Germany, March 2004.

[37] Dan Hildebrand. "***An architectural overview of QNX***". In *Proc. of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126, Seattle, WA, April 1992.

[38] Ted Hills, "***Structured Interrupts***" Operating Systems Review 27(1): 51-68, 1993

[39] Kevin Jeffay, Donald L. Stone, "***Accounting for Interrupt Handling Cost in Dynamic Priority Task Systems***", Proceedings of the IEEE Real-Time Systems Symposium, Raleigh-Durham, NC, December 1993. pp. 212-221.

[40] Kyong Jo Jung, Seok Gan Jung, Chanik Park. "***Stabilizing Execution Time of user processes by Bottom Half Scheduling in Linux***". 16th Euromicro Conference on Real-Time Systems (ECRTS'04) Catania, Italy. June 30 - July 02, 2004.

[41] Do-While Jones, "***Interrupt-Free Design***", Circuit Cellar Magazine, February 1994 pp. 36.

[42] M. Joseph and P. Pandya, ''***Finding Response Times in a Real-Time System***'', BCS Computer Journal, pp. 390-395 (Vol. 29, No. 5, Oct 86).

[43] Lawrence. J. Kenah y Simon F. Bate, "***VAX/VMS Internals and Data Structures***", Digital Press, Bedford, Mass, 1984.

[44] Mark H. Klein , Thomas Ralya , Bill Pollak , Ray Obenza , Michael González Harbour, "***A practitioner's handbook for real-time analysis***", Kluwer Academic Publishers, Norwell, MA, 1993.

[45] Steve Kleiman, Joe Eykholt, "***Interrupts as threads***", ACM SIGOPS Operating Systems Review Volume 29, Issue 2 (April 1995), Pages: 21 – 26, 1995.

[46] Hermann Kopetz. "***The time-triggered model of computation.***" In Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS´98), pages 168–177, Madrid, Spain, December 1998.

[47] Hermann Kopetz. "***Real-Time Systems: Design Principles for Distributed Embedded Applications***" , Kluwer Academic Publishers, 2002.

[48] Jean J. Labrose, "***MicroC/OS-II: The Real-Time Kernel***", Second Edition CMP Books 2002.

[49] Edward L. Lamie, "***Real-Time Embedded Multithreading: Using ThreadX and ARM***", CMP Books, 2005.

[50] William Lamie, "***Pardon the Interruption***", p58, Information Quarterly Vol. 3, Number 4, 2004.

[51] I. Lee, R. King, and R. Paul, "***RK: A Real-Time Kernel for a Distributed System with Predictable Response,***" Tech. Report MS-CIS-88-78/GRASP LAB 155 78, Dept. of Computer and Information Science, Univ. of Pennsylvania, Octubre 1988.

[52] John P. Lehoczky, Liu Sha and Ye Ding, "***The Rate-Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behaviour***", Proceeding IEEE Real-Time System Symposium, pp. 166-171, 1989.

[53] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. "***The design and implementation of an operating system to support distributed multimedia applications***". *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, September 1996.

[54] Luis E. Leyva-del-Foyo, Pedro Mejia-Alvarez, Dionisio de-Niz. "***Aligning Exception handling with design by contract in embedded real-time systems***

development", Proceedings of the IEEE ECOOP 2005 Workshop on Exception Handling in Object Oriented Systems", pp. 125-136, Glasgow, UK, July 25, 2005.

[55] Qing Li, Caroline Yao. "***Real Time Concepts for Embedded Systems***". CMP Books 2003.

[56] Jochen Liedtke, Ulrich Bartling, Uwe Beyer, Dietmar Heinrichs, Rudolf Ruland, and Gyula Szalay. "***Two years of experience with a µ-kernel based OS***". Operating System Review, 25(2):51–62, Apr 1991.

[57] Jochen Liedtke. "***Towards real microkernels***". Communications of the ACM, 39(9):70–77, Sep 1996.

[58] Jochen Liedtke. "***On µ-kernel construction***". In Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP´95), Copper Mountain Resort, Colo., Dec. 1995) ACM Press, New York, Dec. 1995, pp. 237-250.

[59] John Lion, "***Commentary of Unix Version 6 Source Code: Lions' Commentary on Unix - With Source Code***", Peer-to-Peer Communications, 1977.

[60] C. L. Liu and J .W. Layland, "***Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment***", JACM 20(1), pp. 46-61 (1973).

[61] Daniel Lohmann, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. "***On the configuration of non-functional properties in operating system product lines***". In Proceedings of the 4th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS '05), pages 19–25, Chicago, IL, USA, Mar. 2005. Northeastern University, Boston (NU-CCIS-05-03).

[62] Jukka Mäki-Turja , Gerhard Fohler , Kristian Sandström "***Towards Efficient Analysis of Interrupts in Real-Time Systems***". 11th EUROMICRO Conference on Real-Time Systems, York, England.. May 1999, (Work in Progress Publications).

[63] Jim Mauro, Richard McDougall, "***Solaris Internals, Core Kernel Architecture***". Sun Microsystems Press, 2001.

[64] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, John S. Quarterman, "***The Design and Implementation of the 4.4BSD Operating System***". Addison-Wesley, 1996.

[65] Anthony Massa, "***Embedded Software Development with eCos***", Prentice Hall 2002

[66] Jeffrey C. Mogul, K. K. Ramakrishnan, "***Eliminating receive livelock in an interrupt-driven kernel***", ACM Transactions on Computer Systems (TOCS), v.15 n.3, p.217-252, Aug. 1997.

[67] Ralph Moore, "***Link Service Routines for better interrupt handling***", 22/12/05.

[68] F. Mueller, V. Rustavi, T. P. Baker, "***MiThOS – A Real Time Micro-Kernel Threads Operating System***", Proceeding of the IEEE Real-Time Systems Symposium", 1995.

[69] On-Line Applications Research Corporation (OAR). "***RTEMS C User's Guide, edition 4.6.2***", for rtems 4.6.2 edition, August 2003.

[70] "***OSEK/VDX Operating System, Versión 2.0 revision 1, 15.10, 1997***", <http://www.osek-vdx.org>.

[71] John Pagonis, "***Overview of Symbian OS Hardware Interrupt Handling***", Revision 1.0, March 2004

[72] John S. Quarterman, Avi Silberschatz y J. L. Peterson, "***4.2 BSD and 4.3 BSD as Examples of the UNIX System***", ACM Computing Survey, Vol. 17, Nro. 4, diciembre de 1985, págs. 379-418.

[73] L. Sha, R. Rajkumar and J. P. Lehoczky. "***Prioryty Inheritance Protocols: An Approach to real-Time Syncronization.***" IEEE Trans. On Computers, Vol. 39, 1990, pages 1175-1185.

[74] John Regehr , Usit Duongsaa, "***Preventing interrupt overload***", ACM SIGPLAN Notices, v.40 n.7, July 2005.

[75] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. L´eonard, and W. Neuhauser. "***Chorus distributed operating systems. Computing Systems***", 1(4):305–367, 1988.

[76] Sergio Ruocco, "***Real-Time Programming and L4 Microkernels***", Proceedings of the 2006 Workshop on Operating System Platforms for Embedded Real-Time applications, Dresden, Germany, July, 2006.

[77] Mark E. Russinovich, David A. Solomon, "***Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server 2003, Windows XP, and Windows 2000***", Microsoft Press, December 2004.

[78] Ken Sakamura ,"***µITRON 3.0 Specification***," I-01-02E, Version 3.02.00, TRON Association.

[79] Manas Saksena, "***Linux as a Real-Time Operating System***", TymeSys White Paper, 2003.

[80] Kristian Sandström, Christer Eriksson, and Gerhard Fohler "***Handling Interrupts with Static Scheduling in an Automotive Vehicle Control System***", Proceedings of Fifth International Conference on Real-Time Computing Systems and Applications, Hiroshima, Japan, Oct, 1998.

[81] Brand Selic,"***The Challenges of Real-Time Software Design***", Embedded Systems Programming, October 1996. (Article not available online.)

[82] W. Schröder-Preikschat. "***The Logical Design of Parallel Operating Systems***". Prentice-Hall International, 1994. ISBN 0-13-183369-3.

[83] Friedrich Schön, Wolfgang Schröder-Preikschat, Olaf Spinczyk, Ute Spinczyk; "***On Interrupt-Transparent Synchronization in an Embedded Object-Oriented Operating System***", 3rd International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000), pp. 270-277, 15-17 March 2000.

[84] Douglas C. Shmidth, Charles D. Cranor, "***Half-Sync/Half-Async***", en PLoPD2 [22], 1996.

[85] David A. Solomon, "Inside ***Windows NT Second Edition***", Microsoft Press, 1998.

[86] David Solomon and Mark Russinovich, "***Inside Windows 2000, Third Edition***", Microsoft Press, 2000.

[87] B. Sprunt. "***Aperiodic Task Scheduling for Real-Time Systems***." Ph.D. Thesis, Carnegie-Mellon University, August 1990.

[88] John A. Stankoviv, "***Misconception About Real-Time Computing***", Computer, Octubre 1988.

[89] John A. Stankovic y Krithi Ramamritham. "***The spring kernel. A new paradigm for real-time systems***". IEEE Software, Mayo 1991.

[90] John A. Stankovic y Krithi Ramamritham. "***What is Predictability for Real-Time Systems?***", Real-Time Systems, 2, 247-254 (1990).

[91] David B. Stewart. "***Twenty-Five-Most Commons Mistakes with Real-Time Software Development***", Proceedings of 1999 Embedded Systems Conference, San Jose, CA., Septiembre 2004.

[92] David B. Stewart and Gaurav Arora, "***A Tool for Analyzing and Fine Tuning the Real-Time Properties of an Embedded System***", IEEE Transactions On Software Engineering, Vol. 29, No. 4, April 2003.

[93] Daniel Stodolsky, J. Bradley Chen, Brian N. Bershad, "***Fast Interrupt Priority Management in Operating System Kernels***" USENIX Symposium on MicroKernels and Others Kernel Architectures", 1993.

[94] S. T. Taft, R. A. Duff (Eds), "***Ada 95 Reference Manual: Language and Standard Libraries***", LNCS 1246, Springer-Verlag, Berlin. International Standard ISO/IEC 8652:1995(E). 1997.

[95] Martin Timmerman "***Is Windows CE 2.0 a real threat to the RTOS World ?***", Real-Time Magazine 3Q98, 1998.

[96] TimeSys Corporation, "***A TimeSys Perspective on the Linux Preemptible Kernel***", 2003.

[97] K. W. Tindell, "***RTOS interrupt handling: commom errors and how to avoid them", Embedded Systems Programming Europe***", Junio 1999

[98] H. Tokuda y M. Kotera. "***A real-time tool set for the arts kernel***". In Proceedings of the IEEE Real-Time Systems Symposium. Diciembre de 1988.

[99] H. Tokuda y C. W. Mercer. "***Arts: A distributed real-time kernel***". Operating System Review, 23(3), Julio 1989.

[100] Andy Wellings, "***Concurrent and Real-Time Programming in Java***", John Wiley & Sons, 2004.

[101] Matthew Wilcox, "***I'll Do It Later: Softirqs, Tasklets, Bottom Halves, Task Queues, Work Queues and Timers***", Linux Conf AU, The University of Western Australia, Perth, Australia, 22nd to 25th January 2003.

[102] Clark Williams, "***Which is better -- the preempt patch, or the low-latency patch? Both!***", March 20, 2002. <http://www.linuxdevices.com/articles/AT8906594941.html> también en "***Linux Scheduler Latency***", in Embedded Systems (Europe) May 2002. También en Red Hat, March 2002.

[103] Jian Yang, Yu Chen, Huayong Wang, Bibo Wang, "***A Linux Kernel with Fixed Interrupt Latency for Embedded Real-Time System***", Proceedings of the Second International Conference on Embedded Software and Systems (ICESS'05), 2005.

[104] Wang, Yu-Chung and Kwei-Jay Lin, "***Some discussion on the low latency patch for linux"***, In: Conf. Proc. of the Second annual Real-time Linux Workshop, Orlando, Florida, November 27-28, 2000. Florida.

[105] Yuting Zhang and Richard West, "***Process-Aware Interrupt Scheduling and Accounting***", to appear in Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS), December 2006.

[106] Dedicated Sytems Encyclopaedia. http://www.dedicatedsystems.com/encyc/BuyersGuide/RTOS/Evaluations/docspreview.asp

Artículos Publicados

7.3 Artículos Publicados

Como resultado de este trabajo de investigación se han obtenido las siguientes publicaciones en congresos:

[1] Luis E. Leyva-del-Foyo, Pedro Mejia-Alvarez. "***Custom Interrupt Management for Real-Time and Embedded System Kernels***". ERTSI: Embedded Real-Time Systems Implementation Workshop, 25th IEEE International Real-Time Systems Symposium (RTSS04), Lisbon, Portugal. December 2004.

En este artículo se presentaron las dificultades del modelo tradicional de tratamiento de interrupciones en los sistemas de tiempo real (sección 2.6) y se introdujo nuestra propuesta de un modelo completamente integrado de tratamiento de interrupciones (sección 4.1) y tareas conjuntamente con las ecuaciones de su análisis (sección 4.2). El artículo reconoce las dificultades de implementación del modelo en el hardware de interrupción actualmente existente y propone el diseño de un Controlador de Interrupción a la Medida que se implemente utilizando FPGAs.

[2] Luis E. Leyva-del-Foyo, Pedro Mejia-Alvarez, Dionisio de-Niz, "***Predictable Interrupt Management for Real Time Kernels over conventional PC Hardware***", en "Proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium" (RTAS06), San Jose, California, United States, April 4 - April 7, pp. 14-23, 2006.

En este artículo se presentaron las dificultades del modelo tradicional de tratamiento de interrupciones en los sistemas de tiempo real (sección 2.6) así como el diseño del subsistema de administración de interrupciones basado en el modelo integrado (sección 4.3). Se introduce la implementación del modelo integrado utilizando los dos modos de emulación de enmascaramiento físico (con EOI explícito y con EOI Automático. Se presentan las ecuaciones para el análisis con estos modos de emulación (sección 5.4). Se muestran resultados experimentales que validan el comportamiento temporalmente predecible del modelo; así como, los valores de sobrecarga de conmutación de contexto y de la latencia de interrupción. Con este artículo se libera la efectividad del modelo integrado de las restricciones del hardware aunque la emulación presentada implica un costo en el desempeño promedio en el caso promedio.

[3] Luis E. Leyva-del-Foyo, Pedro Mejia-Alvarez, Dionisio de Niz, "***Predictable Interrupt Scheduling with Low Overhead for Real Time Kernels***", en "Proceedings of The 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications " (RTCSA'06), pp. 385-394, Sydney, Australia, August 16-18, 2006.

En este artículo se presenta la implementación del modelo integrado utilizando los modos de emulación con enmascarado virtual. Se presentan los algoritmos detallados del subsistema de administración de interrupción previamente introducido. Se presentan las ecuaciones que modelan los dos nuevos modos de emulación (enmascaramiento virtual con y sin EOI explícito); así como, los resultados experimentales que ponen de manifiesto que este modo de emulación permite obtener un comportamiento predecible temporalmente sin el inconveniente de la sobrecarga y afectación al comportamiento en caso promedio que presentaba la implementación con enmascaramiento físico.

[4] Luis E. Leyva-del-Foyo, Pedro Mejia-Alvarez, Dionisio de Niz, "***Real-Time Scheduling of Interrupt Request over Conventional PC Hardware***", Proceedings of the Seven Mexican International Conference on Computer Science (ENC 2006), pp. 27-36, Mexico. IEEE Computer Society 2006.

Este artículo resume e integra los resultados que se presentaron en los artículos precedentes. Se presenta la implementación del modelo integrado utilizando los modos de emulación con enmascarado virtual. Se presenta la idea de un subsistema de administración de interrupciones configurable para satisfacer distintos requerimientos de comportamiento en el peor caso y comportamiento promedio.

[5] Alain Tamayo-Fong, Luis E. Leyva-del-Foyo, Pedro Mejia-Alvarez, "***Component for Debugging Interrupt Based Systems***", Proceedings of the 7th International Conference on Control, Virtual Instrumentation, and Digital Systems (CICINDI-2006), Mexico City, Mexico, November 21-24, 2006. In Journal "*Research in Computing Science*", ISSN 1665 9899. CIC-IPN, México 2006.

En este artículo se presenta una herramienta que se desarrolló con el propósito de depurar, validar y caracterizar el desempeño del subsistema de administración de interrupciones. La herramienta se desarrolló como componente reutilizable con valor por si mismo (independientemente del modelo integrado).

7.4 Impacto de las Publicaciones

En esta sección documentamos las referencias que hemos encontrado hasta la fecha a los resultados de este trabajo. Por cada referencia documentamos el artículo que hace la referencia, el contexto de la referencia y nuestros comentarios:

1. Referencias al primer artículo publicado: "***Custom Interrupt Management for Real-Time and Embedded System Kernels***" (**4 referencias**):

    1) Tullio Facchinetti, Giorgio Butazzo, Mauro Marinoni and Giacomo Guidi, "***Non-Preemptive Interrupt Scheduling for Safe Reuse of Legacy Drivers in Real-Time Systems***", Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS05), 2005.

        Leyva-del-Foyo and Mejia-Alvarez [8] proposed an integrated approach to interrupt management with related analysis, but their method requires special hardware support.

        Comentario: En los artículos posteriores (**[2][3][4]** de la sección anterior) se presentaron los algoritmos que permiten implementar el esquema integrado sin necesidad de un soporte especial del hardware.

    2) Manuel Coutinho, José Rufino, Carlos Almeida, "***Control of Event Handling Timeliness in RTEMS***", Proceedings of the 17th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2005), Phoenix, AZ, USA, November 14 - 16, 2005.

        In **[13]** there is an effort to integrate task scheduling and interrupt scheduling. Although an important issue, it is not always possible to have this type of approach, due to specific system limitations.

        Comentario: En los artículos posteriores (**[2][3][4]** de la sección anterior) se presentaron los algoritmos que permiten implementar el esquema integrado sobre un hardware convencional de modo que este ya dejo de ser una limitación.

    3) John Regehr, "***Thread Verification vs. Interrupt Verification***", in Proceedings of the 2006 Federated Logic Conference (FLoC 2006), Seattle, August 10 - 22, 2006.

        Similarly, **Leyva-del-Foyo and Mejia-Alvarez's work** [8], the Nemesis OS [15], TimeSys Linux [21], and Solaris [14] all take the interrupts-as-threads approach.

    4) John Regehr, Nathan Cooprider "***Interrupt Verification via Thread Verification***", in Published by Elsevier Science. 2007.

2. Referencias al segundo artículo publicado: "***Predictable Interrupt Management for Real Time Kernels over conventional PC Hardware***" (**8 referencias**):

1) Ricardo Bacha Borges, Rômulo Silva de Oliveira, "***Análise Temporal da Implementação da Fila de Aptos como Lista Ordenada e Lista Desordenada***", Anais do XXVI Congreso da SBC. WSO, II Workshop de Sistemas de Operacionais, Campo Grande, MS, 14 a 20 de julio de 2006.

> Em **[Leyva-Del-Foyo; Mejia-Alvares; Niz 2006]** os autores descrevem uma técnica de implementação na qual as interrupções de hardware respeitam as prioridades das interrupções de software, resultando em um sistema mais previsível. Entretanto, no caso das interrupções do *timer* a técnica apresenta limitações pois a prioridade da interrupção é variável, isto é, depende das tarefas que precisam ser acordadas.
>
> **Comentario**: La prioridad del Manejador de interrupción del Reloj no tiene por que ser dinámica. Ella depende del algoritmo de planificación a elegir. Para el caso de EDF las prioridades serán dinámicas para cualquier tarea, sea activada por interrupción o por software, mientras que si se utiliza asignación un esquema de asignación de prioridades estáticas como RM entonces las prioridades de las tareas activadas por hardware como las activadas por software serán estáticas.

2) Dongwook Kang, Woojoong Lee, and Chanik Park, "***Dynamic Kernel Thread Scheduling for Real-Time Linux***", Eighth Real-Time Linux Workshop, Lanzhou University – SISE, P. R. China, October 12-15, 2006.

> In **[3]**, interrupts are also handled by interrupt service tasks whose role is the same as the IRQ threads. However, while real-time preemption patch allows all ISRs to start execution at the occurrence of IRQs, some IRQs are masked selectively and the corresponding ISRs are not executed in **[3**]. This is done by assigning priorities to IRQs and masking IRQs with the lower priorities. This reduces preemption latencies due to the waking up of ISTs or IRQ threads as well as interrupt handling in interrupt context. However, **[3]** only concentrates on prohibition of the preemption latency caused by interrupts, and does not suggest how to dynamically assign priorities to IRQs in detail.
>
> **Comentario**: El propósito del trabajo es proponer un esquema de administración de interrupciones que permita utilizar cualquier esquema de planificación y de asignación de prioridades para las ISRs reportado en la literatura y que sea adecuado para sistemas de tiempo real (no se sugiere ninguno en especial).

3) Yuting Zhang and Richard West, "***Process-Aware Interrupt Scheduling and Accounting***", to appear in Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS06), December 2006.

> **Leyva-del-Foyo** et al proposed a unified mechanism for synchronization and scheduling of both interrupts and processes [8]. In this work, interrupts are assigned dynamic priorities and scheduled like processes (or tasks) within their own contexts. In the absence of special hardware support, to integrate the scheduling of tasks and interrupts, a software-based abstraction layer must strategically mask the delivery of certain interrupts to ensure task predictability. This whole approach may, however, yield increased context-switching overheads to service interrupts.
>
> **Comentario**: En un artículo posterior (**[3]** de la sección anterior) se presentaron los algoritmos que permiten implementar el esquema integrado sobre un hardware convencional sin la sobrecarga de la conmutación de contexto mediante el uso del enmascaramiento virtual.

4) Mark Lewandowski, Mark Stanovich, Theodore Baker, Kartik Gopalan, An-I Andy Wang. "***Modeling Device Driver Effects in Real-Time Schedulability Analysis: Study of a Network Driver***". *Proceedings of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2007.

> The second technique, followed in the current paper, is to defer most interrupt-triggered work to scheduled threads. Hardware interrupt handlers are kept as short and simple as possible. They only serve to notify a scheduler that it should schedule the later execution of a thread to perform the rest of the interrupt-triggered work. There are variations to this approach, depending on whether the logical interrupt handler threads execute in borrowed (interrupt) context or in independent contexts (e.g. normal application threads), and on whether they have an independent lower-level scheduler (e.g. RTLinux threads or vanilla Linux softirq handlers) or are scheduled via the same scheduler as normal application threads. The more general the thread scheduling mechanism, the more flexibility the system developer has in assigning an appropriate scheduling policy and priority to the device-driven threads. The job of bounding device driver interference then focuses on analyzing the workload and scheduling of these threads. This technique has been the subject of several studies, including [7, **4**, 17, 5], and is implemented in Windows CE and real-time versions of the Linux kernel.

5) Gerald Fry and Richard West, "***On the Integration of Real-time Asynchronous Event Handling Mechanisms with Existing Operating System Services***", in Proceedings of the International Conference on Embedded Systems and Applications (ESA'07), June 2007.

There are several extensions to COTS systems, such as RTLinux, RTAI, and Montavista, that attempt to enable support for real-time tasks [7, 9, 19, **8]**. These systems perform well and allow for best-effort tasks to coexist with real time applications.

6) Dongwook Kang, Woojoong Lee, and Chanik Park, "***Kernel Thread Scheduling in Real-Time Linux for Wearable Computers***". ETRI Journal, vol.29, no.3, June 2007, pp.270-280.

In **[5]**, all interrupt handlers are configured to have static priorities and are executed in a process context like the *IRQ threads* of Ingo Molnar's patch. Note that in the case of [5], the entire execution of an interrupt handler is conducted in the process context, whereas in the case of [1], only some parts of an interrupt handler are conducted in the process context. However, the priority inversion problem also appears in **[5]** due to its static assignment of priorities to interrupt handlers.

Comentario: El propósito del trabajo es proponer un esquema de administración de interrupciones que permita utilizar cualquier esquema de planificación y de asignación de prioridades para las ISRs reportado en la literatura que sea adecuado para sistemas de tiempo real. El trabajo no se limita a asignaciones de prioridades estáticas (esto es un error de interpretación de los autores de este artículo).

7) Peng Liu, Ming Cai, Tingting Fu and Jinxiang Dong, "***An EDF Interrupt Handling Scheme for Real-Time Kernel: Design and Task Simulation***", in Y. Shi et al. (Eds.): Proceedings of the 7th International Conference in Computational Science (ICCS 2007), Part IV, Beijing, China, May 27 - 30, 2007. Lecture Notes in Computer Science, Volume 4490, pp. 969-972, Springer-Verlag. Berlin Heidelberg, 2007.

In **[5]** an integrated scheme of tasks and interrupts to provide predictable execution times to real-time systems is presented

8) Peng Liu, Guojun Dai, Tingting Fu, Hong Zeng and Xiang Zhang, "***A Lazy EDF Interrupt Scheduling Algorithm for Multiprocessor in Parallel Computing Environment***", H. Jin et al. (Eds.): Proceedings of the 7th International Conference Algorithms and Architectures for Parallel Processing, (ICA3PP 2007), Hangzhou, China, June 11-14, 2007. Lecture Notes in Computer Science (LNCS) 4494, pp. 49–59, Springer-Verlag Berlin Heidelberg, 2007.

In **[3]** and [4] an integrated scheme of tasks and interrupts to provide predictable execution times to real-time systems is proposed. …
…Some systems have an integrated mechanism which schedules interrupts and tasks in a mixed way **[3]**[4].

3. Referencias al tercer artículo publicado: "***Predictable Interrupt Scheduling with Low Overhead for Real Time Kernels***", (**1 referencias**):

   9) Peng Liu, Guojun Dai, Tingting Fu, Hong Zeng and Xiang Zhang, "***A Lazy EDF Interrupt Scheduling Algorithm for Multiprocessor in Parallel Computing Environment***", H. Jin et al. (Eds.): Proceedings of the 7th International Conference Algorithms and Architectures for Parallel Processing, (ICA3PP 2007), Hangzhou, China, June 11-14, 2007. Lecture Notes in Computer Science (LNCS) 4494, pp. 49–59, Springer-Verlag Berlin Heidelberg, 2007.

   > In [3] and **[4]** an integrated scheme of tasks and interrupts to provide predictable execution times to real-time systems is proposed. …
   > …Some systems have an integrated mechanism which schedules interrupts and tasks in a mixed way [3] **[4].**

Del resto de los artículos (los más recientes) no hemos encontrado referencias.

Tabla de Contenidos