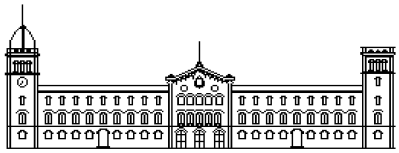


Treball fi de carrera

ENGINYERIA TÈCNICA EN INFORMÀTICA DE SISTEMES

Facultat de Matemàtiques
Universitat de Barcelona

TIEMPO REAL EN EL NÚCLEO LINUX

Jordi Peralta Dolz

Directors: Jaume Timoneda Salat i
Yago Isasi Parache
Realitzat a: Departament de Matemàtica
Aplicada i Anàlisi. UB

Barcelona, 10 de Febrer de 2009

ÍNDICE

CAPÍTULO 1
INTRODUCCIÓN

1.1 INTRODUCCIÓN

El origen y elaboración de este trabajo nace de la necesidad de comprender los mecanismos y algoritmos que el núcleo Linux estándar provee para facilitar un comportamiento válido para soluciones de tiempo real. En campos como la ingeniería aeroespacial, aeronáutica, industrial, telecomunicaciones, etc., se utilizan sistemas informáticos cuya respuesta temporal está regida por unos límites inamovibles, pues bien, está realidad es la que ha motivado la investigación del núcleo Linux en estos escenarios.
Desde hace unas décadas, en las implementaciones para sistemas críticos se opta por el uso soluciones hechas a medida, o bien, soluciones que comprenden sistemas operativos privativos. Algunos de estos sistemas operativos usan núcleos Linux modificados por los equipos de desarrollo de las propias empresas, obteniendo unos resultados más que satisfactorios para su uso en situaciones donde la corrección y la respuesta temporal es básica. Empresas como *WindRiver*, *LynuxWorks* o *FSMLabs* comercializan soluciones de tiempo real con micronúcleos derivados del núcleo Linux estándar.
Además del contexto comercial descrito, dentro de nuestras fronteras y en un marco de investigación como el universitario, el desarrollo y uso de soluciones de tiempo real con el núcleo Linux como protagonista es muy escasa. Universidades próximas como la Politécnica de Catalunya (*UPC*), en investigaciones realizadas con unidades de vuelo autónomo (*UVA*), por ejemplo, usan sistemas operativos Windows sin capacidad de tiempo real y, lógicamente, con el consiguiente pago por la licencia de uso del sistema.
A colación de lo mencionado merece la pena resaltar un hecho que también forma parte de las motivaciones de este trabajo que es, la rápida proliferación del sistema operativo GNU/Linux en una gran diversidad de ámbitos. Desde su aparición a principios de los 90 el sistema operativo GNU/Linux ha ido migrando de ámbitos universitarios a otros lugares que nada tienen que ver con el de origen. Esta rápida propagación se debe a muchas causas, pero una de las más relevantes es la, relativamente, fácil adaptación de su núcleo para tareas y soluciones que quedan lejos de las aportadas por un sistema operativo de propósito general.
Por tanto, el objetivo primordial de este trabajo es comprobar la viabilidad del uso del núcleo Linux en situaciones donde se requiere un comportamiento de tiempo real estricto. En el transcurso de este estudio, se ahondará en las estructuras fundamentales del núcleo y se intentará comprender tanto a nivel práctico como teórico toda relación existente con las modificaciones llevadas a cabo para obtener el comportamiento deseado. En conclusión, son objeto de estudio políticas de planificación, mecanismos de sincronización, sistemas de gestión de memoria y todas las estructuras y algoritmos necesarios para alcanzar la finalidad propuesta. En el siguiente apartado se exponen pormenorizadamente los objetivos perseguidos en la elaboración del presente trabajo.

1.2 OBJETIVOS

Como objetivos de este trabajo se establecen:

- Investigación y comprensión de la codificación del núcleo Linux estándar.
  Para asimilar de forma idónea las modificaciones que se llevan a cabo en el núcleo Linux estándar, con el fin de obtener un comportamiento que cumpla con los requisitos de tiempo real estricto, es necesario ahondar en la ordenación y distribución que se realiza del código del propio núcleo. Además del estudio de la distribución a nivel funcional del código, es necesario comprender las diferentes estructuras de datos confeccionadas exprofeso para el núcleo, los algoritmos que se implementan y el diseño seguido para su funcionamiento.

Sistemas de sincronización de tareas, debido al clásico problema de la concurrencia, son partes del núcleo que deben ser perfectamente asimiladas para entender los cambios que el código sufre tras la conversión en un núcleo de tiempo real estricto. De igual forma, problemas derivados del algoritmo de planificación de tareas escogido, como puede ser la inversión de prioridad, deben ser abordados con detenimiento y analizados para comprender su resolución.
A todo lo anterior se le añade el análisis, comprensión e investigación de todas las novedades implementadas en el núcleo Linux 2.6, versión objeto de este trabajo, que permiten un comportamiento de tiempo real ligero, sin necesidad de ningún tipo de modificación.
El estudio de términos como expropiación (*preemption*), prioridad, espera activa y pasiva, secciones críticas, árboles balanceados, orden de complejidad, planificador y activador, políticas de planificación, tiempo real ligero, y muchos otros conceptos que se encuentran dentro del contexto de codificación del núcleo, es una de las pretensiones del presente trabajo.

- Análisis y comprensión del código resultante de la conversión del núcleo en un sistema de tiempo real estricto.
  Todo el trabajo de asimilación del diseño, sistemas, estructuras y algoritmos del núcleo estándar sirven como base para realizar un exhaustivo seguimiento de las modificaciones llevadas a cabo para obtener un núcleo con capacidad de tiempo real estricto.
  El análisis de los nuevos desarrollos de sistemas de sincronización, como son los semáforos binarios que implentan una espera pasiva, conjuntamente con la transformación de rutinas de interrupción en threads que se ejecutan en modo núcleo, es fundamental para entender las estrategias seguidas para la consecución de un núcleo completamente expropiativo (*fully preemptive kernel*).
  Además de las anteriores *novedades* es necesario comprender perfectamente la gestión que el núcleo Linux hace del tiempo y de cómo la transformación usa nuevas aproximaciones para obtener una resolución temporal de micro, o incluso, nanosegundos.
  El estudio de estos conceptos junto con términos como inversión de prioridad, protocolo de techo de prioridad, protocolo de herencia de prioridad, jiffies, temporizadores de alta resolución, secciones críticas expropiativas, ticks dinámicos, y un largo etc., se hace imprescindible para alcanzar un conocimiento pleno de la solución de tiempo real estricto que con el núcleo Linux estándar se puede llevar a cabo.

Junto con los dos objetivos anteriores, que conforman el eje vertebrador del presente trabajo, existen otros, de carácter colateral, que también forman parte de la motivación original de su elaboración. El estudio del comportamiento de un sistema operativo de tiempo real estricto con núcleo Linux, el uso de herramientas de testing para la verificación de los tiempos de respuesta del sistema, el desarrollo de aplicaciones sobre el sistema operativo resultante y el conocimiento, comprensión y uso de las llamadas al sistema del estándar POSIX de tiempo real, son motivaciones subyacentes de la realización del proyecto.
Y, en último término, la elaboración de un núcleo Linux personalizado mediante su compilación a partir de las fuentes estándar encontradas en ***kernel.org***, conforman el conjunto de objetivos perseguidos en este trabajo final de carrera.

1.3 ORGANIZACIÓN DE LA MEMORIA

Esta memoria se compone de 5 capítulos y 5 apéndices.

- **Capítulo 1**

Es el presente capítulo y, en este se hace una breve introducción de la materia investigada y de las motivaciones e intereses que han conducido a la realización de este trabajo. También se lleva a cabo una breve descripción de la organización de la memoria.

- **Capítulo 2**

Se trata de un capítulo preeminentemente teórico, donde se presentan las principales definiciones del concepto de tiempo real. Además, se hace hincapié en la distinción entre sistemas de tiempo real y sistemas operativos de tiempo real, listando las características de cada uno. Más concretamente, en los sistemas operativos de tiempo real, se hace una recopilación de aquéllos requisitos que un sistema operativo debe cumplir para considerarse de tiempo real. Para finalizar el capítulo se enumeran una serie de ámbitos profesionales donde se usan los sistemas operativos de tiempo real.

- **Capítulo 3**

Es la columna vertebral del trabajo realizado. En este capítulo se detallan las estructuras, subsistemas y algoritmos que los núcleos 2.6 de Linux aportan y que, gracias a éstos, se consigue un escenario propicio para llegar a tener un sistema operativo de tiempo real.

Se inicia el capítulo con un profundo repaso de las características más novedosas del núcleo Linux actual que, como ya se ha comentado, favorecen un comportamiento de tiempo real ligero (*soft real-time*). En el estudio y comprensión de estas características, uno de los objetivos de este trabajo, se ha invertido más de un mes de intenso trabajo. Profundizar en los más ocultos recovecos de un sistema operativo no es una tarea trivial y, lógicamente, se ha requerido de un meticuloso estudio del código fuente y de la propia documentación que la organización kernel.org provee.

Tras este análisis, se exponen las diferentes modificaciones que el parche para tiempo real estricto lleva a cabo. De nuevo, se han tenido que invertir más de dos meses de concienzudo análisis de las nuevas estructuras, algoritmos y estrategias que se presentan con la conversión del sistema en un sistema operativo de tiempo real estricto.

Durante el desarrollo de este trabajo se han usado diferentes utilidades y herramientas para verificar con detenimiento los resultados esperados. El uso de la instrucción ***printk*** en ciertas partes del código del núcleo Linux modificado, herramientas de testing como ***hackbench*** o ***lmbench***, y la realización de scripts en ***bash*** y ***awk***, han estado presentes durante todo el tiempo de investigación de este trabajo.

En la parte final de este capítulo se detallan las acciones a seguir para la compilación del núcleo Linux modificado y se presentan una serie de resultados obtenidos con una utilidad de testeo que confirman el comportamiento del sistema como de tiempo real estricto. Esta última parte del capítulo también ha llevado más de un mes de trabajo, porque lo que se presentaba como una tarea bastante automática como era la compilación del núcleo, acabó siendo un trabajo de una considerable dedicación y atención a las opciones seleccionadas.

- **Capítulo 4**

Este capítulo es un compendio de alternativas a la solución presentada en el anterior capítulo que, también, siguen una política de licencias GNU/GPL.

Por otra parte, se ha querido exponer lo que se ha considerado una de las mayores limitaciones de la conversión del núcleo mediante la aplicación del parche, el algoritmo de gestión de memoria dinámica. En este apartado, se listan los más relevantes algoritmos de gestión de memoria dinámica para acabar presentando un nuevo algoritmo confeccionado por un equipo de ingenieros de la Universidad Politécnica de Valencia. El

estudio de este nuevo algoritmo también ha ocupado una gran parte del tiempo invertido en la realización de este trabajo final de carrera.

- **Capítulo 5**

En este último capítulo se exponen las conclusiones y las líneas de trabajo futuro que se desprenden del análisis realizado del núcleo Linux.

- **Apéndice A**

Presentación de los mecanismos de sincronización de los que hace uso el núcleo para gestionar las situaciones de concurrencia.

- **Apéndice B**

Explicación del problema clásico de inversión de prioridad en sistemas que usan políticas de planificación por prioridades. También se presentan los protocolos más conocidos que subsanan esta problemática, esto es, protocolo de techo de prioridades y protocolo de herencia de prioridad. Este último es el escogido por el parche de tiempo real.

- **Apéndice C**

Se realiza en este apéndice un repaso de las diferentes aproximaciones que se han realizado en los últimos años de temporizadores de alta resolución. Durante mucho tiempo se ha luchado por superar la barrera clásica de los 10ms de ciclo de reloj, que la gran mayoría de arquitecturas y sistemas operativos presentaban.

- **Apéndice D**

Presentación de las aplicaciones desarrolladas sobre el núcleo modificado, aprovechando, a su vez, el comportamiento de tiempo real estricto aportado. Estas aplicaciones han sido realizadas usando funciones típicas del estándar POSIX para tiempo real y, así, de esta forma, conseguir una cierta agilidad en el desarrollo de aplicaciones de tiempo real.

- **Apéndice E**

Contenidos del CD que se adjunta con la presente memoria.

CAPÍTULO 2
FUNDAMENTOS DE TIEMPO REAL

2.1 INTRODUCCIÓN

En las últimas décadas ha ido materializándose la necesidad de crear soluciones informáticas con unas características especiales para dar soporte a la, cada vez más, exigente realidad industrial. Ante esta necesidad, las soluciones clásicas no siempre dan la respuesta adecuada, debido sobretodo a la velocidad con la que los requisitos varían dentro del contexto actual de cambio continuo. Estas soluciones típicas, construidas habitualmente sobre dispositivos microelectrónicos y realizadas específicamente para el problema planteado, no pueden abarcar otro tipo de vicisitudes para las que, lógicamente, no han sido diseñadas.

Además de la creciente necesidad de tener sistemas más adaptables y reconfigurables, también se tiene la elevada proliferación de sistemas electrónicos portables y de alta disponibilidad. Dispositivos como las agendas electrónicas, los teléfonos móviles, los ordenadores de bolsillo, las soluciones en el campo de la automoción, aviónica, domótica, etc., han provocado que las soluciones adoptadas sean, cada vez más, algo más parecido a un sistema operativo que a una aplicación dedicada.
Si a lo expuesto hasta ahora se le suma que estimaciones hechas en 2000 ya presentaban que, más del 99% de la producción mundial de microprocesadores se utilizaba para sistemas empotrados, el panorama que se vislumbra queda más y más alejado de la solución tradicional para dar paso a una nueva e incipiente forma de respuesta.

Pues bien, esta nueva aproximación para dar solución a todas estas situaciones, además de a muchas otras, es el desarrollo de aplicaciones implementadas sobre sistemas operativos de tiempo real empotrados en plataformas hardware específicas.
A partir, de este punto se distinguirá entre sistemas de tiempo real y sistemas operativos de tiempo real. Los primeros se pueden ver como soluciones aportadas a un problema concreto e implementadas como un sistema empotrado, y los segundos se pueden entender como una plataforma sobre la que las soluciones se despliegan.
Habitualmente los sistemas de tiempo real caen en la categoría de sistemas dedicados, es decir, son diseñados pensando en una aplicación específica y previendo un tiempo de vida determinado. Pero, la ventaja de la solución por la actualmente se opta en contraposición a la tradicional, es que una vez el tiempo de vida de la aplicación ha finalizado, siempre se puede desarrollar otra nueva solución sobre el sistema operativo de tiempo real con la que satisfacer las nuevas especificaciones del problema.

En general, se puede afirmar que la misión de un sistema operativo de tiempo real es doble, por una parte suministrar al desarrollador una máquina virtual más fácil de manejar que el propio hardware y que oculte al usuario los aspectos de diseño de éste. Por otra parte debe ser gestor de una serie de recursos hardware y software, que son compartidos con los procesos y usuarios que estén trabajando simultáneamente en el sistema. Y todo esto con el agravante de estar ante un sistema reactivo, es decir, un sistema capaz de entender y reaccionar ante los eventos reales que se producen en su entorno.
En el resto del capítulo se abordarán las características más relevantes de los diferentes conceptos presentados en esta introducción, empezando por la definición de sistema de tiempo real y acabando por los usos que en la actualidad se tiene de los sistemas operativos de tiempo real.

2.2 DEFINICIÓN DE SISTEMA DE TIEMPO REAL

Antes de abordar los entresijos de los sistemas de tiempo real, cabe denotar que la gran mayoría de estos sistemas no tienen sentido fuera de soluciones empotradas y, además, con el uso de arquitecturas RISC(Reduced Instruction Set Computer) como la familia de procesadores ARM.
Existen muchas interpretaciones sobre la naturaleza de un sistema de tiempo real; sin embargo, todas tienen en común la noción de tiempo de respuesta, esto es, el tiempo que precisa el sistema para generar la salida a partir de alguna entrada asociada.
El Diccionario Oxford de computación proporciona la siguiente definición de un sistema de tiempo real:

"Cualquier sistema en el que el tiempo en el que se produce la salida es significativo. Esto generalmente es porque la entrada corresponde a algún movimiento en el mundo físico, y la salida está relacionada con dicho movimiento. El intervalo entre el tiempo de entrada y el de salida debe ser lo suficientemente pequeño para una temporalidad aceptable."

En esta definición el término "temporalidad" se refiere al contexto global del sistema, es decir, para una aplicación que gestione un láser industrial para tareas de alta precisión el tiempo de respuesta puede estar entorno a los microsegundos, mientras que una solución aplicada al ensamblado de piezas en una línea de montaje puede tener suficiente con un tiempo de respuesta de milisegundos o incluso segundos.
Otras definiciones de sistemas de tiempo real son la expuesta por Young(1982), la mencionada por Jim E. Cooling(1991) y finalmente, la aportada por el proyecto PDCS(Predictably Dependable Computer Systems-1995):

Young: *"Cualquier actividad o sistema de proceso de información que tiene que responder a un estímulo de entrada generado externamente en un período finito y especificado."*
Cooling: *"Sistemas de tiempo real son aquellos que deben producir respuestas correctas dentro de un intervalo de tiempo definido. Si el tiempo de respuesta excede ese límite, se produce una degradación del funcionamiento y/o un funcionamiento erróneo."*
PDCS: *"Un sistema de tiempo real es aquél al que se le solicita que reaccione a estímulos del entorno, incluyendo el paso de tiempo físico, en intervalos de tiempo dictados por el mismo entorno."*

Algunas de estas definiciones, en su sentido más amplio, albergan incluso a sistemas operativos de propósito general como pueden ser GNU/Linux o Unix. Cuando un usuario de cualquiera de estos sistemas operativos introduce un comando por la consola espera que la respuesta del sistema operativo sea dada en pocos segundos. De alguna forma, este sería el intervalo temporal del que se habla en las definiciones, pero, a diferencia de los auténticos sistemas de tiempo real, si la respuesta no se produce en el intervalo esperado no ocurre nada catastrófico. Este último aspecto es el que diferencia a un sistema de tiempo real de otro en el que el tiempo de respuesta es importante pero no crucial.

La corrección de un sistema de tiempo real depende no sólo del resultado lógico de la computación, sino también del tiempo en el que se producen los resultados.

Dentro del diseño de sistemas de tiempo real se distinguen habitualmente entre dos

tipologías bien diferenciadas:

- **Sistemas de tiempo real estricto(Hard real-time):**
  Son aquéllos en los que es absolutamente imperativo que las respuestas se produzcan dentro del tiempo límite especificado.
- **Sistemas de tiempo real no estricto(Soft real-time):**
  Son aquéllos en los que los tiempos de respuesta son importantes pero el sistema seguirá funcionando correctamente aunque los tiempos límite no se cumplan ocasionalmente. También se conocen como sistemas de tiempo real ligero o flexible.

Por ejemplo, un sistema de tiempo real estricto sería un sistema de control de freno de un avión. Los aviones disponen de un sistema multiprocesador que debe reaccionar tanto a las órdenes del piloto como al contacto de las ruedas con la pista de aterrizaje. Este sistema debe responder rápida y correctamente frente a eventos externos tomados periódicamente por sensores debido a la criticidad inherente del sistema.
En cambio, un sistema de control de ambiente(calefacciones, aires acondicionados) sería considerado como un sistema de tiempo real flexible(*soft*). Sistemas que gestionan parámetros de control facilitados por sensores como la humedad y la temperatura, se regulan de acuerdo a órdenes del usuario y la pérdida de un dato no implica un fallo del sistema.
En algunos casos se usan límites tanto flexibles como rígidos para caracterizar el comportamiento de un sistema. Es factible tener una respuesta a algún suceso de aviso que tenga un tiempo límite ligero de unos pocos milisegundos, con el fin de obtener una reacción óptima y eficiente, y un tiempo estricto de unas decenas o centenas de milisegundos, para garantizar que no se produzcan daños sobre el equipo o el personal. Entre ambos límites la utilidad de la salida disminuye.

2.3 CARACTERÍSTICAS DE UN SISTEMA DE TIEMPO REAL

Un sistema de tiempo real posee muchas características ya sean inherentes al sistema o bien impuestas. No todos los sistemas de tiempo real comparten las mismas características pero, si bien es cierto, existen un pequeño número de éstas que deben ser satisfechas o por el propio sistema o bien por algún lenguaje de propósito general que aporte las funcionalidades básicas como para implementarlas.

A continuación se presentan las características genéricas más relevantes de los sistemas de tiempo real:

- **Procesamiento concurrente**

Esta propiedad expresa el número de eventos manejados por el sistema, y está relacionada con su grado de paralelismo. El grado de simultaneidad depende de la técnica de repartición de recursos usada, por ejemplo, si el procesador es multitarea la técnica es pseudo-simultánea, ya que el procesador reparte su actividad para satisfacer las diversas peticiones existentes.
La existencia de instantes precisos en los que el sistema debe responder ante eventos reales introduce una mayor complejidad en el diseño del sistema. Aparecen los problemas de sincronización y exclusión mutua interactuando con imposiciones temporales, lo que complica la capacidad de predicción del comportamiento del sistema debido a las siguientes razones:

- El tiempo de reacción de los componentes que constituyen el sistema solo puede ser realmente evaluado a posteriori, una vez que se ha realizado el diseño y escogido una arquitectura hardware.
- Dada una arquitectura, la política de planificación influye en el comportamiento del sistema.
- Dada una arquitectura y una política de planificación, el reparto de tareas entre los distintos procesadores que constituyen el sistema influye sobre el comportamiento global del sistema.

Debido a estas razones la política de planificación solo puede ser escogida a priori en los casos más sencillos. Esta es una de las características principales para caracterizar un sistema de tiempo real.

- **Interacción con interfaces hardware**

Habitualmente los sistemas de tiempo real serán sistemas electrónicos, o con una fuerte componente electrónica, que estarán ligados a su entorno. Este entorno además se considera activo y cambiante. Los sistemas se encuentran en constante interacción mediante sensores y actuadores con el exterior y así controlar su comportamiento. Estos dispositivos interactúan con el computador mediante los registros de entrada y salida, y sus requisitos operativos son dependientes de dispositivo y computador. Los dispositivos pueden generar también interrupciones para indicar al procesador que se han realizado ciertas operaciones o que se han alcanzado ciertas condiciones de error.
En el pasado, la interacción con los dispositivos o bien se realizaba bajo control del sistema operativo, o bien requería que el programador de aplicaciones realizara traumáticas incursiones en el lenguaje ensamblador para controlar y manipular los registros e interrupciones. Actualmente, debido a la variedad de dispositivos y a la naturaleza de tiempo crítico de sus interacciones asociadas, su control debe ser directo, y no a través de una capa de funciones del sistema operativo. Además, los requisitos de calidad son buenos argumentos contra el uso de técnicas de programación de bajo nivel.

- **Funcionalidades de tiempo real**

El tiempo de respuesta es crucial en cualquier sistema de tiempo real. A pesar de esta notable afirmación, diseñar e implementar sistemas que garanticen una salida apropiada generada en un rango de tiempos adecuado bajo todas las condiciones posibles es harto complicado. Llevar a cabo esta empresa usando el máximo de recursos de cómputo del sistema es altamente improbable. Por todas estas razones, los sistemas de tiempo real se construyen habitualmente usando procesadores con considerable capacidad adicional, garantizando de esta forma que el comportamiento en el peor de los casos no produzca ningún retraso inoportuno durante los períodos críticos de operación del sistema.

Dada una adecuada potencia de proceso, se precisa soporte de lenguaje y de ejecución para permitir al programador:

- Especificar los tiempos en los que deben ser realizadas las acciones.
- Especificar los tiempos en los que las acciones deben ser completadas.
- Responder a situaciones en las que no todos los requisitos temporales se pueden satisfacer.

- Responder a situaciones en las que los requisitos temporales cambian dinámicamente. Estas situaciones reciben el nombre de modos de cambio.

Estos requisitos se llaman funcionalidades de control en tiempo real. Permiten que el programa se sincronice con el tiempo. Por ejemplo, en una central eléctrica es necesario incrementar el suministro de electricidad a los usuarios domésticos a las 5 de la tarde de lunes a viernes cada semana. Esto es consecuencia de la detección de un pico en la demanda cuando las familias vuelven a su casa desde el trabajo, encienden las luces, cocinan la cena, etc.

Un ejemplo de cambio se puede encontrar en los sistemas de control de vuelo de los aviones. Si un avión ha experimentado despresurización, existe una necesidad inmediata de dedicar todos los recursos de computación a la emergencia.
Con el fin de cumplir los tiempos de respuesta, es necesario que el comportamiento del sistema sea predecible.

- **Arquitectura distribuida**

Esta característica indica si el sistema puede ser implementado en un único procesador o debe ser distribuido entre varios procesadores. Estos procesadores pueden encontrarse en la misma placa, en un rack compartiendo bus, acoplados vía red, etc. Hay motivaciones, no excluyentes, que llevan a incorporar la interconexión de microprocesadores:

- El entorno del sistema: si hay una dispersión geográfica natural, donde para el correcto funcionamiento deben colaborar varias máquinas dispersas, suele requerir que los sistemas de control que manejan cada máquina se comuniquen.
- Prestaciones del sistema: aquí el tiempo de reacción debe ser mínimo y está impuesto por la funcionalidad del sistema, pudiendo ser alcanzado solamente por medio de un procesamiento paralelo.
- Confiabilidad del sistema: se puede exigir una arquitectura distribuida para que en caso de fallo de un procesador no se venga abajo todo el sistema, tomando otros su carga de trabajo, o degradándose paulatinamente.

- **Fiable y seguro**

El fallo de un sistema implicado en la generación de electricidad podría redundar en el fallo de un sistema de soporte vital, como una unidad de cuidados intensivos.
La incorrecta o tardía gestión temporal de algún proceso crítico de una planta química podría desembocar en un irreversible daño ambiental o en una dramática pérdida de vidas humanas.
El error de tratamiento de algún sistema vital para una planta nuclear podría ocasionar situaciones parecidas a las ya vividas en la antigua Unión Soviética.
Todos estos ejemplos ilustran que el hardware y el software de un computador deben ser fiables y seguros, y más aún cuando se trata de sistemas críticos en los que la respuesta temporal es vital para el buen funcionamiento de la solución. Además, donde se precise interacción con un operador, deberá tenerse especial cuidado en el diseño de la interfaz, con el fin de minimizar la posibilidad de error humano.
El tamaño y complejidad de los sistemas de tiempo real potencian el problema de la fiabilidad; no sólo deben tenerse en cuenta las dificultades esperadas inherentes a la

aplicación, sino también aquellas introducidas por un diseño software defectuoso.

- **Implementación eficiente y entorno de ejecución**

Teniendo en cuenta que los sistemas de tiempo real son críticos respecto al tiempo, la eficiencia de la implementación será más importante que en otros sistemas. El uso de lenguajes de alto nivel no están permitidos en el desarrollo de soluciones de tiempo real, precisamente por la propia esencia de estos lenguajes, que permiten al programador abstraerse de los detalles de la implementación y concentrarse en la resolución del problema.

En este tipo de sistemas el programador debe preocuparse del coste de utilizar las características de un lenguaje particular. Por ejemplo, si se precisa que la respuesta a algún estímulo sea de un microsegundo, no tiene sentido usar una característica cuya ejecución precisa un milisegundo. Por motivos como este, lenguajes como Java no están del todo bien considerados en entornos donde la criticidad de la respuesta es muy elevada, es decir, en sistemas de tiempo real estricto y además con un muy bajo tiempo de respuesta.
Existen otras características secundarias como reconfigurabilidad, usabilidad, certificabilidad, capacidad de evolución, etc., que no son tan comunes o genéricas como las anteriores pero que, en determinadas situaciones, son igualmente críticas.

2.4 SISTEMAS OPERATIVOS DE TIEMPO REAL

Como se ha comentado en la introducción, los sistemas operativos, en general, tienen una doble misión, por un lado crear una capa de abstracción sobre el hardware en el que están instalados y por otro lado gestionar toda una serie de recursos hardware y software compartidos por todas las tareas.

Estas capacidades que aporta un sistema operativo conjuntamente con las características específicas de tiempo real(predictabilidad y determinismo), son las que, precisamente, lo hacen sumamente atractivo para su uso en estos sistemas. Una de las ventajas más relevantes del uso de sistemas operativos de tiempo real es la simplificación de la implementación de algoritmos en sistemas de tiempo real. El desarrollo de soluciones de tiempo real, habitualmente, suele presentar una complejidad notable y, además, puede llegar a complicarse más por el uso específico del hardware que la solución requiera. Esta complejidad comentada suele ser menor si la solución a implementar usa los servicios y mecanismos provistos por un sistema operativo de tiempo real.

Un sistema operativo de tiempo real debe ser modular y extensible. En sistemas empotrados, el núcleo del sistema debe ser pequeño ya que habitualmente será cargado en memorias ROM o RAM donde el espacio es limitado. El tamaño y la simplicidad son dos aspectos que un sistema operativo de tiempo real debe tener muy presente. Estos motivos, entre otros, son los que obligan a este tipo de sistemas operativos a estar compuestos por un microkernel(ver Figura 1) que provea solo los servicios esenciales, como planificación, sincronización y manejo de interrupciones.

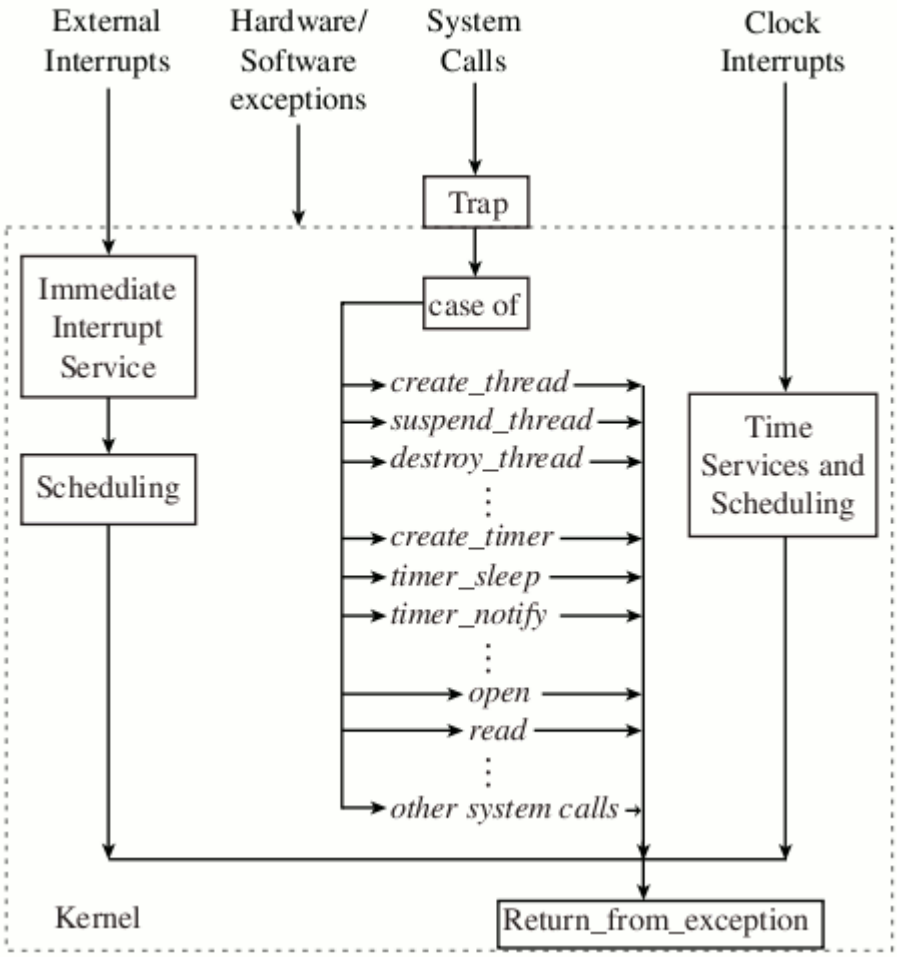


Figura 1: Estructura de un microkernel.

Otro hecho remarcable de los sistemas operativos de tiempo real es el beneficio de usar las primitivas o llamadas del sistema, ya que simplifican enormemente la utilización de diferentes algoritmos de planificación por parte del usuario.

Algunas aplicaciones de tiempo real requieren del uso de todas las funcionalidades que un sistema operativo de propósito general aporta. Un ejemplo de esto serían los sistemas de control del tráfico aéreo, que necesitan gestionar servicios de entrada y salida, ficheros, red, etc. Pues bien, un sistema operativo de tiempo real provee este tipo de servicios como componentes autónomos que pueden ser añadidos al microkernel. Estos componentes suelen recibir el nombre de proveedores de servicios del sistema(System Service Providers-SSP).

En sistemas operativos de tiempo real suelen convivir aplicaciones que deban seguir un comportamiento de tiempo real estricto, junto con otras que sean flexibles o que, ni siquiera sean de tiempo real.

El soporte para realizar extensiones a las funcionalidades aportadas por sistemas operativos de tiempo real, en sistemas UNIX/LINUX, recae en el estándar de tiempo real POSIX. Con el API(Application Program Interface) de tiempo real de POSIX se pueden extender las funcionalidades de la unidad mínima de proceso(thread) para tener un comportamiento aún más predecible.

2.5 REQUISITOS DE UN SISTEMA OPERATIVO DE TIEMPO REAL

Un sistema operativo de tiempo real debe presentar cinco requisitos especiales:

- **Determinismo**

Un sistema operativo es determinista si realiza las operaciones en instantes fijos y predeterminados o en intervalos de tiempo predeterminados. Cuando hay varios procesos compitiendo por recursos, incluido el procesador, ningún sistema será por completo determinista. El punto hasta el cual un sistema puede satisfacer las solicitudes de manera determinista depende, en primer lugar, de la velocidad con que pueda responder a las interrupciones y, en segundo lugar, de si el sistema posee suficiente capacidad para gestionar todas las peticiones en el tiempo requerido. Una medida útil de la capacidad de un sistema operativo para operar de forma determinista es el retardo máximo que se produce desde la llegada de una interrupción de alta prioridad hasta que comience el servicio de la rutina asociada. En un sistema operativo de tiempo real este tiempo puede ir desde unos pocos microsegundos a 1 milisegundo, en los sistema operativos que no son de tiempo real el retardo puede caer en un rango desde decenas a cientos de milisegundos.

- **Sensibilidad**

Es una característica semejante a la anterior, hace referencia a cuanto tiempo consume un sistema operativo en reconocer una interrupción, es el tiempo preciso para dar servicio a la interrupción después de haberla reconocido. Depende de:

- La cantidad de tiempo necesaria para iniciar la gestión de la interrupción y empezar la ejecución de la rutina de tratamiento. Si la ejecución de la rutina de tratamiento de la interrupción requiere un cambio de proceso ese tiempo será mayor.
- La cantidad de tiempo necesario para ejecutar la rutina de tratamiento de la interrupción.
- El efecto de anidamiento de las interrupciones. El servicio se retrasará si el sistema debe atender la llegada de otra interrupción más prioritaria.
- El determinismo y la sensibilidad forman conjuntamente el tiempo de respuesta a sucesos externos.

- **Control del usuario**

En general, es mucho mayor el control que tiene el usuario en sistemas operativos de tiempo real que en sistemas de tiempo compartido. En éstos últimos un usuario no puede otorgar prioridades a sus procesos, decidir sobre el algoritmo de planificación, qué procesos deben estar siempre residentes en memoria, etc.

- **Fiabilidad**

Es normalmente mucho más importante en sistemas operativos de tiempo real. Un sistema de tiempo real controla sucesos que están teniendo lugar en el entorno y en su propia escala de tiempos, las pérdidas o degradaciones en el sistema que los controla pueden tener consecuencias catastróficas, como ya se ha señalado en la introducción.

- **Tolerancia a fallos**

Un sistema operativo de tiempo real debe diseñarse para responder incluso ante varias formas de fallo. Se pretende que se pueda conservar la capacidad máxima y los máximos datos posibles en caso de fallo. Opciones como la de volcar el contenido de la memoria e aun archivo y abortar el programa ante la aparición de un fallo están totalmente

prohibidas. Un sistema operativo de tiempo real intentará corregir el problema o minimizar sus efectos antes de proseguir con la ejecución.
Asociado a la tolerancia de fallos se encuentra el concepto de estabilidad. Un sistema será estable si en los casos en los que es imposible cumplir todos los plazos de ejecución de las tareas se cumplen al menos los de las más críticas y de mayor prioridad.

2.6 CARACTERÍSTICAS Y PRESTACIONES DE UN SISTEMA OPERATIVO DE TIEMPO REAL

En aras a satisfacer los requisitos anteriores las características que los sistemas operativos de tiempo real deben ofrecer son:

- **Soporte para la planificación de procesos de tiempo real**

Un sistema operativo de tiempo real debe proporcionar soporte para la creación, borrado y planificación de múltiples procesos, cada uno de los cuales monitoriza o controla parte de una aplicación. Típicamente, en estos sistemas, es posible definir prioridades por procesos e interrupciones. En contraste, en un sistema de tiempo compartido, solo el propio sistema operativo determina el orden en que se ejecutan los procesos.

- **Planificación por prioridad**

Un sistema operativo de tiempo real debe asegurar que un proceso de alta prioridad, cuando esté listo para ejecutarse, pase por delante de un proceso de más baja prioridad. El sistema operativo deberá ser capaz de reconocer la condición, a menudo debido a una interrupción, pasar por delante del proceso que se está ejecutando y realizar un rápido cambio de contexto para permitir la ejecución de un proceso de más alta prioridad. Por ejemplo, un sistema operativo de tiempo compartido como es GNU/LINUX si tiene un proceso ejecutándose en modo núcleo debe esperar a que finalice su ejecución en este modo para luego activar el proceso más prioritario.

- **Garantía de respuesta ante interrupciones**

Un sistema operativo de tiempo real debe reconocer muy rápidamente la aparición de una interrupción o un evento, y tomar una acción determinista, esto es, bien definida en términos funcionales y temporales, para atender a ese evento. Debe responder tanto a interrupciones de tipo hardware como software. El propio sistema operativo debe ser interrumpible y reentrante. Las interrupciones son una fuente de introducción de indeterminismo y conducen a la aparición de altos tiempos de latencia.

- **Comunicación entre procesos**

Un sistema operativo de tiempo real debe ser capaz de soportar comunicaciones entre procesos de manera fiable y precisa, tales como semáforos, colas de mensajes y memoria compartida. Estas facilidades se emplean para sincronizar y coordinar la ejecución de los procesos, así como para la protección de datos y la compartición de recursos.

- **Adquisición de datos a alta velocidad**

Es necesario que el sistema sea capaz de manejar conjuntos de datos con un alta velocidad de adquisición. De esta forma, un sistema operativo de tiempo real proporciona medios para optimizar el almacenamiento de datos en disco, sobre todo a través de la entrada/salida con buffers. Otras características adicionales pueden ser la posibilidad de preasignar bloques de disco contiguos a archivos(almacenamiento secuencial) y dar

control al usuario sobre los buffers.

- **Soporte de entrada y salida**

Las aplicaciones de tiempo real típicamente incluyen cierto número de interfaces de entrada y salida. Un sistema operativo de tiempo real debe proporcionar herramientas para incorporar fácilmente dispositivos de E/S específicos. Para los dispositivos estándar la librería estándar debería ser suficiente. Deben además soportar entrada y salida asíncrona, donde un proceso puede iniciar una operación de E/S, y luego continuar con su ejecución mientras concurrentemente se está realizando la operación de E/S. En este aspecto cabe remarcar la existencia de procesadores específicos(canales de E/S, controladores de DMA, etc.) dedicados a realizar operaciones de E/S sin la participación de la CPU.

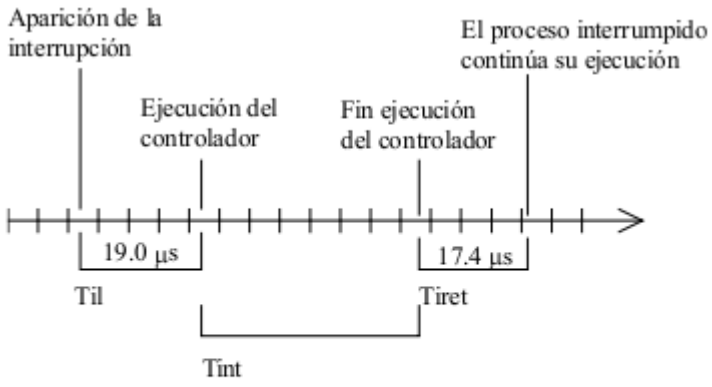
- **Control a cargo del usuario de los recursos del sistema**

Una característica clave de los sistemas operativos de tiempo real es la capacidad de proporcionar a los usuarios el control específico de los recursos del sistema, incluyendo la propia CPU, memoria y recursos de E/S. El control de la CPU se logra sobre la base de una planificación por prioridades en la que los usuarios pueden establecer los niveles de prioridad de los procesos. Además, se dispone de temporizadores de tiempo real y de funciones para su manejo con el fin de planificar eventos y períodos de espera. Un sistema operativo de tiempo real debe también facilitar el bloqueo de la memoria(locking), de esta forma se puede garantizar que un programa, o parte de éste, permanece en memoria, con el objetivo de poder realizar cambios de contexto de manera más rápida cuando ocurre una interrupción. Debería ser capaz de permitir al usuario la asignación de buffers y la posibilidad de bloquear o desbloquear archivos y dispositivos. El control que se ejerce en general es mucho mayor en los sistemas operativos de tiempo real que en los sistemas operativos genéricos. En estos últimos un usuario no tiene control sobre la prioridad de sus procesos, el algoritmo de planificación, el cierre de páginas virtuales en memoria, la reserva de espacio en disco, etc.

En el transcurso de la explicación de las diferentes características de un sistema operativo de tiempo real se ha podido observar que uno de los temas clave para un buen comportamiento determinista del sistema es la minimización del tiempo transcurrido desde la aparición de un estímulo externo y la correspondiente ejecución del código relacionado con dicho evento. Este tiempo, conocido como latencia, puede servir para caracterizar varios subsistemas de un sistema operativo y así poder analizar las prestaciones que éste puede proveer.

- **Latencia ante interrupciones**

Es el tiempo transcurrido desde la recepción de una señal de interrupción hardware hasta que la primera instrucción software del controlador de interrupciones es ejecutada. Hay que considerar que cuando aparece una interrupción debe guardarse en la pila el valor del contador del programa y, probablemente, la palabra de estado. Se debe acudir a una tabla de vectores de interrupción y cargar el valor correspondiente en el contador de programa. Todas estas operaciones son realizadas automáticamente por el hardware y consumen poco tiempo, salvo que las interrupciones, por alguna necesidad concreta del programa, se encuentren desactivadas temporalmente, en cuyo caso la latencia será mayor.

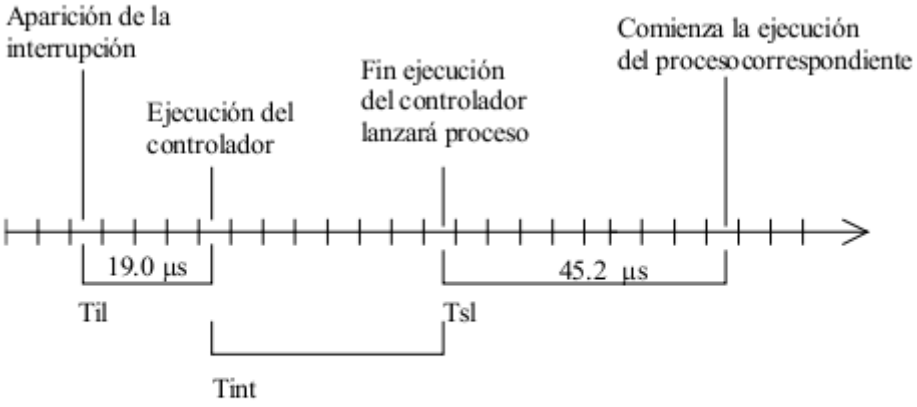


Til tiempo de latencia de la interrupción
Tint tiempo de procesamiento de la interrupción
Tiret tiempo de retorno tras la finalizacuón

Figura 2: Latencia de interrupción.

- **Latencia ante la planificación**

En muchas ocasiones, ante la aparición de un evento, será necesario lanzar un nuevo proceso. Se definirá la latencia en la aplicación como el tiempo transcurrido entre la finalización de un controlador de interrupción y la ejecución de un proceso provocada por esa interrupción. Este tiempo comprende las demoras necesarias para verificar qué proceso se debe planificar, realizar el cambio de contexto y, circunstancialmente, el tiempo que el sistema pudiera tener desactivada su capacidad de expropiación. Este tiempo es necesario para ajustar punteros y acceder a estructuras de datos exclusivas del sistema.



Til tiempo de latencia de la interrupción
Tint tiempo de procesamiento de la interrupción
Tsl tiempo de latencia de planificación

Figura 3: Latencia de planificación.

En ambos tipos de latencia se pueden estudiar los tiempos promedios, pero para situaciones donde la criticidad es alarmante se debería tener en cuenta el tiempo en los peores escenarios. Estos casos se darían, por ejemplo, cuando se estuviera ante una situación de simultaneidad de interrupciones, cuando la atención a éstas no estuviera activa, que la capacidad de expropiación del núcleo estuviera temporalmente deshabilitada, que el sistema estuviera sobrecargado, etc.

2.7 ÁMBITOS DE APLICACIÓN

Como se ha ido comentando a lo largo de este capítulo, los usos y aplicaciones de los sistemas de tiempo real son básicamente los sistemas de control, es decir, sistemas compuestos por un conjunto de sensores, una o varias unidades de proceso y un conjunto de actuadores. En estos sistemas la solución de tiempo real debe monitorizar a través de los sensores el estado del sistema, evaluando las posibles acciones de control y enviando a los actuadores las señales pertinentes en el instante adecuado.

Los ámbitos de uso más habituales son:

- Sistemas de producción industrial.
- Sistemas telemáticos.
- Sistemas de guiado, tanto en el ámbito de la automoción como en el aeronáutico.
- Sistemas quirúrgicos de alta precisión.
- Sistemas de control del tráfico, tanto aéreo como automovilístico.
- Sistemas de control militares(guiado de misiles, etc.)
- Robótica.
- Sistemas de navegación en dispositivos aeroespaciales.

CAPÍTULO 3
TIEMPO REAL EN EL NÚCLEO LINUX

3.1 INTRODUCCIÓN

Tras haber detallado las diferentes características y requisitos de los sistemas operativos de tiempo real, en el capítulo actual se abordarán los aspectos más relevantes del núcleo del sistema operativo GNU/LINUX y las modificaciones que se deben realizar para convertir a este sistema en un sistema operativo de tiempo real estricto.
La explicación se iniciará con la exposición de las claves que, a partir de la versión del núcleo 2.6, permitieron tener un comportamiento de sistema de tiempo real ligero. Las claves se presentarán conjuntamente con fragmentos de código del núcleo que aportarán una visión más detallada de las múltiples estructuras de datos que se tuvieron que crear y modificar con respecto a versiones anteriores.
Con las características propias del núcleo Linux 2.6 no se tiene suficiente como para garantizar un comportamiento de tiempo real estricto, motivo por el que, en el aparatado 3.3, se ahondará en las diferentes modificaciones que se deben realizar para obtener un auténtico sistema operativo de tiempo real. Estas modificaciones son debidas al parche que, los propios desarrolladores del núcleo Linux, mantienen e implementan. El resultado de la aplicación de este parche será tratado ampliamente en esta sección.
Finalmente, se detallarán las operaciones necesarias para compilar el núcleo Linux con el parche de tiempo real y se realizarán una serie de pruebas, tanto internas como externas, que demostrarán que el resultado es un sistema operativo de tiempo real estricto.

3.2 CLAVES DEL NÚCLEO 2.6 PARA UN COMPORTAMIENTO SOFT RT

El tránsito de la versión del núcleo Linux 2.4 a la versión 2.6 supuso una extraordinaria mejora en las características y prestaciones del sistema operativo GNU/LINUX. Las modificaciones llevadas a cabo fueron muchas, pero son solo cinco las que posibilitan tener un sistema operativo con un comportamiento de tiempo real ligero. Además, estas mejoras conforman la estructura básica necesaria como para otorgar a este sistema un comportamiento de tiempo real estricto. Esta última afirmación es el objeto de análisis del siguiente apartado (3.3).
Se pasará, ahora, a enumerar las claves del actual núcleo Linux para, posteriormente, explicar pormenorizadamente cuáles han sido las mejoras aportadas por cada una de éstas:

- Colas de planificación (Runqueues)
- Arrays de prioridad (Priority arrays)
- Planificador con complejidad O(1)
- Expropiación (preemption)
- Tiempo real (soft RT)

3.2.1 Colas de ejecución

Son las estructuras básicas de datos en el planificador.
Las colas de ejecución son las listas de procesos preparados para un único procesador además, contienen información de planificación por procesador.
Todo esto hace que las colas de ejecución sean las principales estructuras de datos de planificación por cada procesador.

La definición de las colas de ejecución se encuentra en el fichero fuente ***sched.c*** y no en la cabecera correspondiente para no permitir el uso de estas estructuras de datos al resto del código del núcleo Linux. Las colas de ejecución son única y exclusivamente estructuras de datos destinadas al uso del planificador.

```
struct rq {
        spinlock_t lock; /*protección de la cola de ejecución*/
        unsigned long nr_running; /*número de tareas listas*/
        unsigned long long nr_switches; /*contador de cambios de contexto*/
        unsigned long nr_uninterruptible; /*nº de tareas no interrumpibles*/

        /*tiempo del último intercambio de arrays*/
        unsigned long expired_timestamp;
        struct task_struct *curr; /*puntero a la actual tarea en cpu*/
        struct task_struct *idle; /*puntero a la tarea idle del procesador*/

        /*mapa de memoria de la última tarea ejecutada*/
        struct mm_struct *prev_mm;
        struct prio_array *active; /*array de prioridades activo*/
        struct prio_array *expired; /*array de prioridades expirado*/
        struct prio_array arrays[2]; /*definición de los arrays actuales*/
        atomic_t nr_iowait; /*nº de tareas esperando una operación I/O*/
        struct task_struct *migration_thread; /*thread migrado*/
        struct list_head migration_queue; /*cola migrada*/
        ...
```

Existen una serie de macros con las que poder obtener información de la cola de ejecución asociada a un procesador o a un proceso.
La siguiente macro retorna un puntero a la cola de ejecución asociada al procesador pasado como argumento:

#define cpu_rq(cpu) (&per_cpu(runqueues,(cpu)))

Esta, en cambio, devuelve la cola de ejecución del procesador actual:

#define this_rq() (&__get_cpu_var(runqueues))

La siguiente macro devuelve un puntero a la cola de ejecución donde la tarea pasada como argumento ha sido encolada:

#define task_rq(p) cpu_rq(task_cpu(p))

Solo existe una única estructura runqueue por procesador.

Esto en núcleos anteriores al 2.6 no era así, solo existía una única cola para todos los procesadores, y esto, lógicamente, implicaba una mala eficiencia.
Solo una tarea puede modificar una cola de ejecución en un tiempo dado y se realiza mediante la obtención del lock de la cola.
Con el fin de evitar *deadlocks*, los tokens de las colas se deben obtener en orden ascendente. Para garantizar este comportamiento existen funciones como:

```
double_rq_lock(rq1,rq2)
double_rq_unlock(rq1,rq2)
task_rq_lock(task,&flags) ;
```

```
/*
 * lock the rq a given task resides on and disable interrupts.
 * Note the ordering: we can safely lookup the task_rq without
 * explicitly disabling preemption.
 */
static struct rq *task_rq_lock(struct task_struct *p, unsigned long *flags)
        __acquires(rq->lock){
        struct rq *rq;
repeat_lock_task:
        local_irq_save(*flags);
        rq = task_rq(p);
        spin_lock(&rq->lock);
        if (unlikely(rq != task_rq(p))) {
                spin_unlock_irqrestore(&rq->lock, *flags);
                goto repeat_lock_task;
        }
        return rq;
}
```

Los *spinlocks* son usados para prevenir el uso simultáneo de colas de ejecución por parte de múltiples tareas. Estos locks funcionan como la llave de una puerta. Si una tarea se ha hecho con la llave y ha abierto la puerta, ésta es cerrada tras de si. Si otra tarea pretende abrir la puerta, al no haber obtenido la llave, deberá esperar a que la primera tarea salga y devuelva la llave. La espera recibe el nombre de ***spinning*** ya que la tarea se ve inmersa en un loop en el que constantemente está verificando si la llave ha sido devuelta(espera activa).

```
/**
 * Safely lock two runqueues
 * Note this does not disable interrupts like task_rq_lock,
 * you need to do so manually before calling.
 */
static void double_rq_lock(struct rq *rq1, struct rq *rq2)
        __acquires(rq1->lock)
        __acquires(rq2->lock){
        BUG_ON(!irqs_disabled());
        if (rq1 == rq2) {
                spin_lock(&rq1->lock);
                __acquire(rq2->lock);          /* Fake it out ;) */
        } else {
                if (rq1 < rq2) {
                        spin_lock(&rq1->lock);
                        spin_lock(&rq2->lock);
                } else {
                        spin_lock(&rq2->lock);
                        spin_lock(&rq1->lock);
                }
        }
}
```

3.2.2 Arrays de prioridad (Priority arrays)

Es la base de las mejoras del núcleo 2.6, y en concreto del comportamiento de complejidad O(1) del algoritmo de planificación.
Existen 2 arrays de prioridades por cola de ejecución:

```
struct prio_array *active;
struct prio_array *expired;
struct prio_array arrays[2];
```

Cada array contine una cola de procesos preparados por nivel de prioridad. También contienen un mapa de bits de prioridades, que es usado para la localización eficiente de la tarea preparada con un mayor nivel de prioridad en el sistema.

```
struct prio_array {
        unsigned int nr_active; /*nº de tareas en la colas*/
        DECLARE_BITMAP(bitmap, MAX_PRIO+1); /*mapa de bits de prioridades*/
        struct list_head queue[MAX_PRIO]; /*colas de prioridad*/
};
```

La definición de la estructura de datos ***struct list_head*** se encuentra en el archivo fuente ***linux-2.6.21.6/include/linux/list.h***, y se trata de una lista doblemente encadenada.
MAX_PRIO es el número de niveles de prioridad en el sistema. Por defecto, son 140. La definición se encuentra en la cabecera ***linux-2.6.21.6/include/linux/sched.h***

```
/**
 * La prioridad de un proceso va de 0..MAX_PRIO-1.
 * La prioridad de procesos RT es 0..MAX_RT_PRIO-1.
 * Tareas con políticas de planificación SCHED_NORMAL/SCHED_BATCH
 * se encuentran en el rango MAX_RT_PRIO..MAX_PRIO-1. Los valores
 * de prioridad están invertidos:
 * p->prio bajo significa mayor prioridad.
 *
 * El valor de MAX_USER_RT_PRIO permite que la prioridad máxima actual de
 * RT sea separada del valor exportado del espacio de usuario.
 * Esto permite que los threads del kernel establezcan su prioridad como
 * valor mayor que cualquier tarea de usuario. Nota:
 * MAX_RT_PRIO no debe ser menor que MAX_USER_RT_PRIO.
 */

#define MAX_USER_RT_PRIO        100
#define MAX_RT_PRIO             MAX_USER_RT_PRIO

#define MAX_PRIO            (MAX_RT_PRIO + 40)
...
```

En cuanto al mapa de bits, en el header ***linux-2.6.21.6/include/linux/types.h*** se puede observar la siguiente definición:

#define BITS_TO_LONGS(bits) (((bits)+BITS_PER_LONG-1)/BITS_PER_LONG)
#define DECLARE_BITMAP(name,bits) unsigned long name[BITS_TO_LONGS(bits)]

Por tanto, el tamaño del mapa de bits debería ser, como mínimo, el número de niveles de prioridad en el sistema. Como la arquitectura provee palabras de 32 bits, el mapa de bits tendrá 5 elementos de 32 bits cada uno, teniendo, pues, 160 bits.
Cada array de prioridades contiene un campo de mapa de bits que como mínimo tiene un bit por cada nivel de prioridad en el sistema.
Inicialmente, todos los bits están establecidos a 0. Cuando una tarea de una prioridad

determinada pasa al estado de lista, es decir, se establece su estado a ***TASK_RUNNING***, el bit correspondiente en el mapa de bits es establecido a 1. Por tanto, la búsqueda de la tarea con prioridad más alta en el sistema no es más que la búsqueda del primer bit con valor 1 en el mapa de bits.

Como el número de prioridades es estático, el tiempo para completar la búsqueda es *constante* y no depende del número de tareas listas en el sistema. Además, en cada arquitectura soportada por el núcleo Linux se implementa el algoritmo ***find first set*** con el objetivo de realizar una búsqueda lo más rápida posible en el mapa de bits.

Habitualmente, la función que realiza esta búsqueda se llama ***sched_find_first_bit()***, y se encuentra en ***linux-2.6.21.6/include/asm-generic/bitops/sched.h***

```
static inline int sched_find_first_bit(const unsigned long *b){
#if BITS_PER_LONG == 64
        if (unlikely(b[0]))
                return __ffs(b[0]);
        if (likely(b[1]))
                return __ffs(b[1]) + 64;
        return __ffs(b[2]) + 128;
#elif BITS_PER_LONG == 32
        if (unlikely(b[0]))
                return __ffs(b[0]);
        if (unlikely(b[1]))
                return __ffs(b[1]) + 32;
        if (unlikely(b[2]))
                return __ffs(b[2]) + 64;
        if (b[3])
                return __ffs(b[3]) + 96;
        return __ffs(b[4]) + 128;
#else
#error BITS_PER_LONG not defined
#endif
}
```

```
static inline unsigned long __ffs(unsigned long word){
        int num = 0;
        if ((word & 0xffff) == 0) {
                num += 16;
                word >>= 16;
        }
        if ((word & 0xff) == 0) {
                num += 8;
                word >>= 8;
        }
        if ((word & 0xf) == 0) {
                num += 4;
                word >>= 4;
        }
        if ((word & 0x3) == 0) {
                num += 2;
                word >>= 2;
        }
        if ((word & 0x1) == 0)
                num += 1;
        return num;
}
```

Como se ha comentado anteriormente, cada array de prioridades contiene un tipo abstracto de datos ***queue[]*** de tipo ***struct list_head.*** Una cola para cada uno de los niveles de prioridad existente en el sistema. Cada una de estas colas contiene todos los procesos listos para entrar en CPU. La búsqueda de la siguiente tarea es una simple elección del siguiente elemento en la lista.
Finalmente, la estructura ***prio_array*** contiene una variable entera sin signo que almacena el número de tareas preparadas.
Los 140 niveles de prioridad en el sistema, se dividen entre:

- ***[0, 99]*** Prioridad para tareas de tiempo real.
- ***[100, 139]*** Prioridad para tareas de usuario. Estos valores también son conocidos como *nice values,* y suelen ser representados, en modo usuario, por el rango de prioridades [-20, 19].

Para una prioridad concreta, los elementos de cada una de las colas siguen una política de planificación Round Robin.
Por tanto, estamos ante un sistema de colas multinivel con retroalimentación, donde la política entre arrays está regida por un algoritmo de prioridades expulsivas y la política en cada uno de los arrays es RR.
A medida que los procesos de la cola activa van consumiendo su rodaja de tiempo pasan a la cola de procesos expirados. Durante el paso, la nueva prioridad(si existe) al igual que la nueva rodaja de tiempo son recalculadas.
Cuando la cola activa está vacía los punteros entre las dos colas activa y expirada se cambian, haciendo que la expirada sea ahora la activa.
Usar dos arrays de prioridades por CPU, *active runqueue* y *expired runqueue* permite efectuar la transición entre "épocas" de timeslices en un tiempo constante.
El trabajo del planificador, tras ver todo esto, es fácil, seleccionar la tarea que ocupa la primera posición en el array de más alta prioridad.

3.2.3 Planificador con complejidad O(1)

Un algoritmo de planificación opera con un tamaño de entrada formado por el número de tareas que están preparadas para ser procesadas.
Para llevar a cabo el análisis de la eficiencia y el rendimiento de un algoritmo se usan medidas de complejidad temporal. Estas medidas no son más que el tiempo de cómputo necesario para la ejecución del algoritmo.
Usualmente el análisis de la eficiencia se realiza mediante el estudio de su comportamiento frente a condiciones extremas. El estudio del comportamiento asintótico del algoritmo puede llegar a facilitar información de su calidad, tanto de la escalabilidad como de la eficiencia y rendimiento que presenta.
Una familia de funciones que comparten un mismo comportamiento asintótico se dice que tienen un mismo ***Orden de Complejidad***. Estas familias se designan con la notación **O()**.
La notación **O()** ignora los factores constantes, desconoce si se hace una mejor o peor implementación del algoritmo, además de ser independiente de los datos de entrada del mismo. Es decir, la utilidad de aplicar esta notación a un algoritmo es encontrar el límite superior de su tiempo de ejecución, esto es, ***el peor caso***.
El planificador del núcleo Linux no contiene un solo algoritmo que se ejecute peor que O(1), es decir, con una complejidad constante. Como se ha reseñado en párrafos anteriores, independientemente del número de entradas que se tengan, una complejidad constante garantiza un tiempo concreto e inamovible de realización de las funciones del planificador(cambios de contexto, manejo de las interrupciones y *wakeup* de procesos).
Este comportamiento no se daba en núcleos inferiores a 2.6, por ejemplo, en la versión

2.4 el planificador tenía una complejidad lineal, **O(n)**, y por tanto, su rendimiento decaia con el crecimiento del número de entradas del sistema. En situaciones de carga elevada del sistema el procesador dedicaba más tiempo a las tareas del propio planificador que no a los procesos en si mismos. En resumen, los algoritmos que formaban parte del planificador del núcleo 2.4 tenían una importante carencia de escalabilidad.
Para finalizar con este apartado, mencionar que las claves de la consecución de una complejidad constante en el planificador del núcleo 2.6 se debe, básicamente, a tres estructuras ya analizadas, colas de ejecución, arrays de prioridad y mapas de bits.

3.2.4 Expropiación (preemption)

Los sistemas operativos multitarea pueden ser categorizados de dos modos diferentes: multitarea cooperativa y expropiativa.
Linux, como la gran mayoría de sistemas operativos Unix actuales, está basado en el segundo de los modos, esto es, multitarea expropiativa. En ésta, el planificador decide cuando una tarea debe ser interrumpida y expulsada de la CPU, y cuando una nueva tarea debe iniciar o reanudar su ejecución.
La expropiación se puede entender como el acto involuntario de suspensión de la ejecución de una tarea. El tiempo de uso de CPU de una tarea está predeterminado, y se conoce con el nombre de quantum o radaja(timeslice). La gestión de los quantums por parte del planificador establece un escenario muy favorable para la toma de decisiones globales y para la prevención del posible monopolio de la CPU a manos de alguna tarea "malintencionada".
En contraposición a los sistemas multitarea expropiativos, los cooperativos permiten la ejecución de una tarea hasta su finalización, sin tener en cuenta posibles problemas de inanición y handicaps similares. La expropiación del uso de la CPU se lleva a cabo durante los cambios de contexto. La codificación de los cambios de contexto dentro del núcleo Linux se encuentra en la función ***context_switch()*** que está definida en ***sched.c***

```
/* context_switch – cambia a un nuevo mapa de memoria y a un
 * nuevo estado de los registros del thread.*/
static inline struct task_struct *
                        context_switch(struct rq *rq, struct task_struct *prev,
                        struct task_struct *next) {
        struct mm_struct *mm = next->mm;
        struct mm_struct *oldmm = prev->active_mm;
        arch_enter_lazy_cpu_mode();
        if (!mm){
                next->active_mm = oldmm;
                atomic_inc(&oldmm->mm_count);
                enter_lazy_tlb(oldmm, next);
        }else
                switch_mm(oldmm, mm, next);
        if (!prev->mm) {
                prev->active_mm = NULL;
                WARN_ON(rq->prev_mm);
                rq->prev_mm = oldmm;
        }
#ifndef __ARCH_WANT_UNLOCKED_CTXSW
        spin_release(&rq->lock.dep_map, 1, _THIS_IP_); /* Desbloqueos de rq */
#endif
        switch_to(prev, next, prev); /* Estado de los registros y de la pila.*/
        return prev;
}
```

Esta función es llamada desde ***schedule()***, justo en el instante en el que un nuevo proceso ha sido seleccionado para entrar en CPU. Esta función realiza, básicamente, dos tareas:

- Llama a ***switch_mm()*** definida en ***include/asm/mmu_context.h***, donde se realiza la gestión del cambio del mapa de memoria de la tarea actual a la tarea siguiente.

- Llama a ***switch_to()*** definida en ***include/asm/system.h***, que lleva a cabo el cambio del estado del procesador del proceso actual al siguiente. Esta operación lleva implícito el hecho de salvar y restaurar la información de la pila y de los registros del procesador.

Con el objetivo de articular los cambios de contexto involuntarios en el núcleo se define un flag llamado ***need_resched.*** Este flag es activado por la función ***scheduler_tick()*** tras la finalización de la rodaja de tiempo del proceso actual, y por la función ***try_to_wake_up()*** en el instante en el que una tarea de mayor prioridad se encuentra en estado listo.
Este flag es verificado por el núcleo Linux cada vez que se retorna a modo usuario o se retorna del tratamiento de una interrupción. Si está activado se realiza una llamada al planificador antes de proseguir con el desarrollo normal.
Como nota, decir que en las actuales versiones del núcleo Linux el flag ***need_resched*** no es global, como ocurría anteriormente, sino que está definido bajo el nombre ***TIF_NEED_RESCHED*** en la estructura ***thread_info*** ubicada en la cabecera ***include/asm/thread_info.h***
Se puede distinguir entre dos tipos diferentes de expropiación:

- Expropiación en modo usuario.
- Expropiación en modo privilegiado.

3.2.4.1 Expropiación en modo usuario.

Este tipo de expropiación se da en el instante en el que se retorna a modo usuario y el flag ***need_resched*** está activado. Siempre que se retorna a modo usuario, ya sea de una interrupción o de una llamada al sistema, el flag es verificado, por si se debe planificar otra tarea o bien dejar ejecutando la actual.

3.2.4.2 Expropiación en modo núcleo.

A diferencia de otras variante de UNIX, el núcleo 2.6 está implementado siguiendo una política de prioridades expulsivas. Es posible expulsar una tarea de la CPU siempre y cuando el núcleo se encuentre en un estado seguro para planificar una nueva tarea. El núcleo Linux es capaz de llevar a cabo la expropiación de la CPU de una tarea mientras ésta no tenga bajo su control ningún **lock**. Los **locks** se pueden ver como delimitadores de zonas donde se puede realizar una expropiación o no. Dado que el núcleo Linux es SMP-safe, si una tarea no tiene ningún **lock** retenido, puede ser expulsada de la CPU en favor de otra. Para dar soporte a la característica expropiativa del núcleo se ha establecido un contador llamado ***preempt_count*** en la estructura ***thread_info***. Este contador que, inicialmente vale 0, se va incrementando a medida que la tarea se hace con un ***lock*** y, lógicamente, se decrementa cuando lo libera. Cuando se retorna del tratamiento de una interrupción a modo privilegiado, se evalúan dos propiedades, a diferencia de lo que ocurría en modo usuario, el flag ***need_resched*** y el contador ***preempt_count***. Si el flag está activo y el contador es 0, entonces una tarea más prioritaria está lista para hacerse con la CPU y, por tanto, el planificador será

invocado. Si el contador es diferente a 0, la tarea que está ejecutándose tiene en su poder algún ***lock*** y por tanto, no es seguro planificar otra tarea. En este caso se vuelve de la interrupción y se deja la misma tarea que estaba ejecutándose. Cuando la tarea actual libera todos los ***locks*** y el contador retorna 0, el código del unlock verifica el flag ***need_resched*** y, si está activo, el planificador es invocado. La expropiación en modo privilegiado también se puede realizar de forma explícita. Esta forma de expropiación siempre ha sido soportada ya que no necesita codificación adicional para asegurar o comprobar en que estado se encuentra el núcleo. Esta expropiación se puede llevar a cabo llamando explícitamente a la función ***schedule()*** o bien bloqueando la tarea en ejecución. Se asume que en la codificación en la que se invoca al scheduler se sabe si el estado es seguro, es decir, que no se tiene retenido ningún **lock**. En resumen, la expropiación en modo privilegiado sucede:

- Cuando un manejo de interrupción termina y antes de volver al modo usuario.
- Cuando el código del núcleo vuelve a ser expropiativo.
- Si una tarea en el núcleo invoca explícitamente a la función ***schedule()***.
- Si una tarea en el núcleo se bloquea con el consiguiente resultado de la invocación de ***schedule()***.

3.2.5 Tiempo real ligero (soft Real-Time)

El núcleo Linux 2.6 proporciona 2 políticas de planificación de tiempo real, ***SCHED_FIFO*** y ***SCHED_RR***. La política de planificación usada para tareas de tiempo compartido recibe el nombre de ***SCHED_OTHER***.

- ***SCHED_FIFO,*** tal y como indica su nombre, implementa un algoritmo first-in-first-out, o lo que es lo mismo, first-come-first-served. Toda tarea planificada siguiendo esta política tiene preferencia ante una tarea que siga una planificación ***SCHED_OTHER***. Una tarea con estado ***TASK_RUNNING*** y planificación ***SCHED_FIFO*** se ejecuta hasta que se bloquea, momento que pasa a estado ***TASK_INTERRUPTIBLE***, o bien, hasta que abandona voluntariamente el procesador, ***sched_yield()***. Se debe tener en cuenta que, siguiendo esta política de planificación, que no implementa ningún tipo de distribución temporal de uso de cpu, una tarea podría ejecutarse de forma indefinida. Una tarea planificada con este algoritmo puede ser expulsada de la CPU por otra tarea de mayor prioridad que siga una de las dos políticas de planificación de tiempo real. Si dos tareas de una misma prioridad han sido planificadas siguiendo un algoritmo FCFS el planificador intenta asignarles un timeslice con el objetivo de repartir de forma imparcial el uso de CPU pero, una tarea no dejará efectivamente el procesador hasta que no haga un yield() de forma explícita.

- ***SCHED_RR***, de forma similar a lo explicado con la política anterior, es la implementación de un algoritmo de planificación Round Robin. Las tareas que siguen este tipo de planificación hacen uso de la CPU durante un intervalo de tiempo predeterminado. Dos tareas de una misma prioridad siempre son planificadas siguiendo este algoritmo, y, como sucedía en la otra política de tiempo real, una tarea con mayor prioridad siempre pasa por delante de una tarea menos prioritaria. Bajo ningún concepto una tarea con menor prioridad expropia la CPU a una tarea con una prioridad mayor, ni siquiera en el supuesto de que ésta haya agotado su quantum de tiempo.

Ambas políticas de planificación de tiempo real posibilitan que el núcleo Linux pueda tener un comportamiento ***soft real-time***. En este tipo de comportamiento el núcleo trata de

planificar diferentes tareas dentro de un rango temporal determinado, sobretodo en referencia al límite temporal superior. A diferencia de lo que sucede en sistemas de tiempo real estricto, en la situación actual el núcleo no garantiza la consecución de la planificación dentro del rango temporal, sino que simplemente se intenta. Esto quiere decir que, en situaciones especiales, el límite temporal superior puede ser rebasado.
Las prioridades de tiempo real, como ya se ha comentado anteriormente, van desde 0 hasta el valor de ***MAX_RT_PRIO - 1***. Por defecto, ***MAX_RT_PRIO*** es 100 y, por tanto, el intervalo de prioridades de tiempo real que existe en el núcleo es [0, 99].
Ocurre algo similar con las tareas que siguen una política de planificación ***SCHED_OTHER***. En estas tareas el valor conocido como ***nice***(modo usuario) se encuentra entre el valor ***MAX_RT_PRIO*** hasta ***MAX_RT_PRIO+(40-1)***, es decir, dentro del espacio global de niveles de prioridad el rango para tareas de tiempo compartido es [100, 139]. La traducción de estos valores en modo usuario es [-20, 19].

Existen otras mejoras como, por ejemplo, la optimización realizada en el balanceador de carga dentro del marco de un sistema multiprocesador(symmetrical multiprocessing system), o, incluso, la asignación dinámica de prioridades a tareas de tiempo compartido. Pero, teniendo en cuenta que el objetivo perseguido es confeccionar un sistema operativo de tiempo real que, a la postre, estará empotrado en algún dispositivo monoprocesador, el tratamiento de mejoras en SMP, o el cálculo dinámico de prioridades para tareas de tiempo compartido, están fuera del alcance de este trabajo.

Como conclusión remarcar que las claves expuestas son condición necesaria para tener un comportamiento de tiempo real flexible, pero no son condición suficiente para albergar un comportamiento estricto. En el código del núcleo Linux estándar, aquí evaluado, se pueden encontrar multitud de ejemplos que no respetan los requisitos mínimos necesarios para poder cumplir unas pautas temporales estrictas. Estos fragmentos y estructuras son los que, precisamente, trata de modificar el parche que se aborda en la siguiente sección.

3.3 PARCHE DE TIEMPO REAL PARA EL NÚCLEO LINUX

3.3.1 Introducción

GNU/Linux es un sistema operativo de propósito general preparado para ser soportado en un amplio número de arquitecturas. Durante los últimos años un grupo de desarrolladores del núcleo Linux, liderados por Ingo Molnar, han trabajado en un parche para dotar a este sistema operativo de capacidades de tiempo real estricto. A lo largo del desarrollo de este parche, muchas de las funciones implementadas han pasado a formar parte del propio núcleo Linux, de forma que, actualmente, ya son parte del núcleo que se puede descargar del sitio oficial http://www.kernel.org. Pero, aún y habiendo fusionado parte del trabajo del grupo de desarrolladores en el núcleo estándar, existe una gran parte de la funcionalidad que sigue residiendo en el parche, ya que está relacionada con el código fuente que depende de cada una de las arquitecturas a las que el núcleo da soporte.

3.3.2 Escenario actual

Multitud de aplicaciones necesitan un comportamiento predecible y determinístico en aras a satisfacer los requisitos impuestos por el tiempo real, mientras que otras solo necesitan satisfacer una mínima parte. Esta disparidad de necesidades es el motivo por el que un sistema operativo debe proporcionar un soporte genérico lo suficientemente amplio como para albergar todos los comportamientos posibles. Uno de los propósitos principales de un sistema operativo en el escenario presentado, no consiste en augmentar la velocidad

de las aplicaciones ni en mejorar el tiempo de respuesta ante un estímulo dado, sino que estriba en garantizar un comportamiento determinístico sea cual sea la carga del sistema.
Este no es el caso del núcleo Linux que, como eje vertebrador de un sistema operativo de propósito general, tiene como objetivo maximizar el rendimiento y minimizar el tiempo de respuesta. Además, estos objetivos suelen estar en conflicto, ya que el hecho de minimizar el tiempo de respuesta implica un trabajo computacional mayor en los cambios de contexto y, por tanto, un peor rendimiento. En cuanto al planificador, éste intenta tomar sus decisiones conociendo todos los requisitos del sistema y, dinámicamente, materializar estas decisiones preservando la eficiencia de la planificación.
De todo lo anterior se puede desprender que un núcleo Linux, perteneciente a un sistema operativo de propósito general, a lo sumo puede proveer un comportamiento de tiempo real ligero, ya que se lucha por tener un buen comportamiento y un alto rendimiento.
En contraposición, un sistema de tiempo real estricto requiere de unas garantías de coordinación y predictibilidad que van más allá de las que un sistema de propósito general puede proporcionar. El objetivo primordial de un sistema operativo de tiempo real estricto es minimizar el tiempo de latencia y posibilitar una planificación predictible. De hecho, un sistema de tiempo real estricto no tiene porque aportar un mayor rendimiento que un sistema de carácter general, pero sí debe cumplir unos inamovibles requisitos temporales en la producción de una salida correcta.
A pesar de todo lo mencionado anteriormente, el núcleo Linux, y de forma más global, todo el sistema operativo GNU/Linux, es muy atractivo para ser usado en sistemas empotrados debido, básicamente, a su bajo coste, su adaptabilidad, su configurabilidad y su portabilidad. GNU/Linux es usado en sistemas como PDAs, PocketPCs, teléfonos móviles, dispositivos de audio y video, etc. Incluso existen multitud de sistemas que usan el núcleo Linux para gestionar robots y vehículos industriales, pasando por sistemas de alta precisión como láseres. El déficit de auténticas capacidades de tiempo real en el núcleo Linux no ha sido, hasta ahora, una barrera para su elección en sistemas que operan en situaciones críticas.

Con el núcleo Linux 2.6, como ya se ha visto, se tiene la capacidad de expropiación. En resumen, cuando una tarea adquiere el estado de preparada, el núcleo verifica si su prioridad es mayor que la de la tarea que en ese momento se está ejecutando. Si esta verificación es positiva, el planificador es invocado para expulsar del procesador a la tarea menos prioritaria y planificar y activar a la tarea más prioritaria. La expropiación ha sido mejorada para dar un mejor tiempo de reacción y bajas latencias.
De todas formas, la actual implementación del núcleo Linux contiene fragmentos de código donde la capacidad de expropiación ha sido desactivada. En ocasiones, una tarea con una prioridad baja puede no ser expulsada de la CPU por otra de mayor prioridad por encontrarse en una sección crítica. Aún más, la expropiación es desactivada cuando el núcleo está ejecutando alguna rutina de un servicio de interrupción o bien algún proceso de interrupción aplazado. Lógicamente, para una aplicación de tiempo real estricto, este comportamiento aportado por el sistema operativo es absolutamente inaceptable. El objetivo del parche de tiempo real es minimizar la cantidad de código del núcleo que no es expropiativo. Un núcleo Linux compilado con el parche ofrece un comportamiento expropiativo con alguna pequeña salvedad que se comentará más adelante.
Otro handicap para la obtención de un comportamiento de tiempo real estricto era la resolución temporal a la que era capaz de llegar el núcleo Linux. Tradicionalmente, esta resolución no bajaba de 10ms (1ms en función de la arquitecutra), esto para ciertas aplicaciones de tiempo real no era suficiente. Actualmente ya se tiene una mayor resolución con la aportación realizada por el subsistema de temporizadores de alta

resolución (High-resolution timers HRT).

En un inicio, el parche expropiativo de Molnar no fue aceptado en la rama principal de desarrollo del núcleo, debido al gran número de cambios que introducía en, prácticamente, todo el código. Pero, poco a poco, y teniendo en cuenta que el propio Ingo Molnar era un desarrollador del núcleo se fueron introduciendo funcionalidades del parche en el código del núcleo mantenido en la rama principal. Este hecho propició que pasado un tiempo, la aplicación del parche sobre un núcleo estándar no generara cambios tan sumamente drásticos en el código fuente. En la actualidad, el parche puede ser descargado del sitio oficial http://www.kernel.org/pub/linux/kernel/projects/rt, de forma similar a lo realizado con el código fuente del núcleo Linux estándar.

3.3.3 Funcionalidades del parche

El parche de Molnar, referido de ahora en adelante como parche RT-preempt, incorpora tres funcionalidades diferentes al núcleo Linux, a saber:

- **Núcleo expropiativo completo.**
  Mediante la transformación de los spin locks del núcleo en mutex que soportan la herencia de prioridad, y convirtiendo las rutinas de servicio de interrupción en threads, prácticamente todo el código del núcleo es expropiativo.
- **Temporizadores de alta resolución.**
  Mediante la separación de los temporizadores de los timeouts e introduciendo un nuevo gestor de eventos de reloj, la resolución temporal de los eventos de reloj ya no está limitada a **jiffies** (tradicionalmente, un jiffy era el intervalo de tiempo que separaba dos ticks de reloj), sino al propio hardware de la máquina.

- **Ticks dinámicos.**
  Mediante la reprogramación de las acciones llevadas a cabo en los temporizadores y en las operaciones de entrada y salida gestionadas en los eventos de reloj, se consigue que la frecuencia de los eventos de reloj no esté limitada a ser periódica.

Aunque todos los componentes han sido implementados intentando minimizar la dependencia con la arquitectura usada, aún existe cierta parte del código que es dependiente de la plataforma.
También cabe destacar que la implementación de los temporizadores de alta resolución(HRT) y los ticks dinámicos, son proyectos independientes del desarrollo del parche RT-preempt. Estos proyectos son mantenidos por Thomas Gleixner y, periódicamente, su funcionalidad es añadida al parche de Molnar.

3.3.3.1 Núcleo expropiativo completo

Algunos de los escollos que se han tenido que superar para obtener un núcleo completamente expropiativo han tenido como protagonistas a algunas de las estructuras de sincronización del núcleo(ver Apéndice A).
A continuación se enumerarán y explicarán las características del parche RT-preempt que proporcionan un núcleo completamente expropiativo:

- Secciones críticas expropiativas.
- Herencia de prioridad para spin locks y semáforos del núcleo.
- Rutinas de servicios de interrupción expropiativas.

3.3.3.1.1 Secciones críticas expropiativas

Uno de los problemas más relevantes en el ámbito de las secciones críticas es la equivalencia entre el tiempo de liberación del lock y el tiempo de respuesta que presenta el sistema. La impredictibilidad de la latencia del sistema motivada por el uso de mecanismos de sincronización como los spin locks(ver Apéndice A) deviene en un grave problema para las aplicaciones de tiempo real. En el parche RT-preempt, este problema es resuelto convirtiendo los spin locks en semáforos binarios(mutex). Teniendo en cuenta que los mutex implementan una espera pasiva, las secciones críticas dentro del núcleo Linux pasan a ser expropiativas.

Existen spin locks en el parche RT-preempt que, a pesar de la transformación llevada a cabo, deben permanecer con la funcionalidad propia de los spin locks, esto es, implementando una espera activa. Se pueden encontrar ejemplos en el código del planificador y en la propia implementación de los semáforos binarios en los archivos fuente del núcleo. Esta es una de las salvedades referidas en el punto 3.3.2, donde la aplicación del parche no puede corregir el comportamiento de los mecanismos de sincronización.

Tras la compilación del núcleo con el parche el tipo de los spin locks cambia de ***spinlock_t*** a ***raw_spinlock_t***. Cuando un spin lock sea invocado como ***raw_spinlock_t*** se comportará como un mecanismo de sincronización con espera activa, deshabilitando la capacidad de expropiación del núcleo. En cambio, si es invocado como ***spinlock_t*** el mecanismo presentará una espera pasiva(mutex), posibilitando así la capacidad expropiativa.

En el parche RT-preempt también se presenta un nuevo estado de las tareas con el objetivo de permitir al planificador preservar el valor anterior del estado de la tarea en ejecución cuando existen situaciones de bloqueo. El siguiente fragmento de código clarificará la anterior explicación:

```
spinlock_t mi_lock1;
spinlock_t mi_lock2;

spin_lock(&mi_lock1);
current->state = TASK_UNINTERRUPTIBLE;
spin_lock(&mi_lock2);
mi_funcion();
spin_unlock(&mi_lock2);
spin_unlock(&mi_lock1);
```

Desde el momento en el que la función *spin_lock(&mi_lock2)* se puede comportar como un semáforo e implementar una espera pasiva, el valor *current->state* puede ser modificado y, por tanto, esto podría afectar a la ejecución de la función *mi_funcion().* Este es el motivo de la inclusión de un estado adicional en el parche RT-preempt llamado **TASK_RUNNING_MUTEX.**

Con las modificaciones que lleva a cabo el parche es ilegal invocar a la función ***spin_lock()*** mientras la expropiación o las interrupciones estén desactivadas, pero, aún así, existen situaciones en las que una tarea no puede progresar hasta que no adquiera un spin lock. En este tipo de casos la solución que aporta el parche es la de aplazar la operación requerida por la función ***spin_lock()*** hasta que la expropiación haya sido reactivada. En consecuencia, numerosas funciones han sido duplicadas remarcando su carácter de demora añadiendo a su nombre el sufijo **_delayed**.

El planificador puede ser afectado por las operaciones aplazadas. Un ejemplo sería, cuando una tarea expulsa a otra de menor prioridad pero no puede progresar ya que

necesita el lock que la tarea de menor prioridad ha adquirido en un instante previo. De esta forma, la tarea entraría inmediatamente en el estado de bloqueo esperando a que el lock fuera liberado. Con el fin de evitar expropiaciones innecesarias de este tipo el parche RT-preempt extiende uno de los flags usados en la cabecera **thread_info.h** llamado **TIF_NEED_RESCHED,** que informa de una necesaria replanificación de la tarea, a **TIF_NEED_RESCHED_DELAYED.** Este nuevo flag realiza la misma función que el anterior, pero espera hasta que la tarea esté lista para volver a modo usuario o bien hasta el siguiente paso por la función ***preempt_check_resched_delayed()***.

A continuación se muestra como queda modificada la lista de flags de la cabecera que se encuentra en la ruta ***include/asm/thread_info.h***:

```
/*
 * thread information flags
 * - these are process state flags that various assembly files may need to access
 * - pending work-to-be-done flags are in LSW
 * - other flags in MSW
 */
#define TIF_SYSCALL_TRACE              0       /* syscall trace active */
#define TIF_NOTIFY_RESUME              1       /* resume notification requested */
#define TIF_SIGPENDING                 2       /* signal pending */
#define TIF_NEED_RESCHED               3       /* rescheduling necessary */
#define TIF_SINGLESTEP                 4       /* restore on ret to user mode */
#define TIF_IRET                       5       /* return with iret */
#define TIF_SYSCALL_EMU                6       /* syscall emulation active */
#define TIF_SYSCALL_AUDIT              7       /* syscall auditing active */
#define TIF_SECCOMP                    8       /* secure computing */
#define TIF_RESTORE_SIGMASK            9       /* restore sigmask in do_signal()
*/
#define TIF_NEED_RESCHED_DELAYED       10      /* resched on ret to userspace */
#define TIF_MEMDIE                     16
#define TIF_DEBUG                      17      /* uses debug registers */
#define TIF_IO_BITMAP                  18      /* uses I/O bitmap */
#define TIF_FREEZE                     19      /* is freezing for suspend */
```

Entre las numerosas funciones afectadas por estas modificaciones está la función **cpu_idle()**, por ejemplo, donde el procesador solo podrá dormir si los dos flags no están activados.

```
void cpu_idle(void) {

        int cpu = smp_processor_id();
        current_thread_info()->status |= TS_POLLING;

        /* endless idle loop with no priority at all */
        while (1) {
                tick_nohz_stop_sched_tick();
                while (!need_resched() && !need_resched_delayed()) {
                        void (*idle)(void);

                        if (__get_cpu_var(cpu_idle_state))
                                __get_cpu_var(cpu_idle_state) = 0;
        ...
}
```

3.3.3.1.2 Herencia de prioridad para spin locks y semáforos.

Como consecuencia de la transformación de spin locks en semáforos, que lleva a cabo el parche RT-preempt, y ya sea con la expropiación habilitada o no, la inversión de prioridad(ver Apéndice B) puede estar presente en cualquier parte del código del núcleo Linux. Y esto es debido, principalmente, a la característica de la implementación que realizan los semáforos de la espera pasiva.

De las dos soluciones clásicas para el problema de inversión de prioridad, el núcleo Linux opta por implementar la herencia de prioridad, que consiste en que una tarea menos prioritaria, que tenga en su poder el lock de un recurso, herede la prioridad de la tarea más prioritaria que esté esperando a hacerse con el lock. La otra solución que es el protocolo de techo de prioridad no es usada ya que podría provocar que tras un instante de tiempo desde el arranque del sistema ninguna de las tareas que se hayan apoderado de un recurso pueda ser expulsada por otra de mayor prioridad (ver Apéndice B).

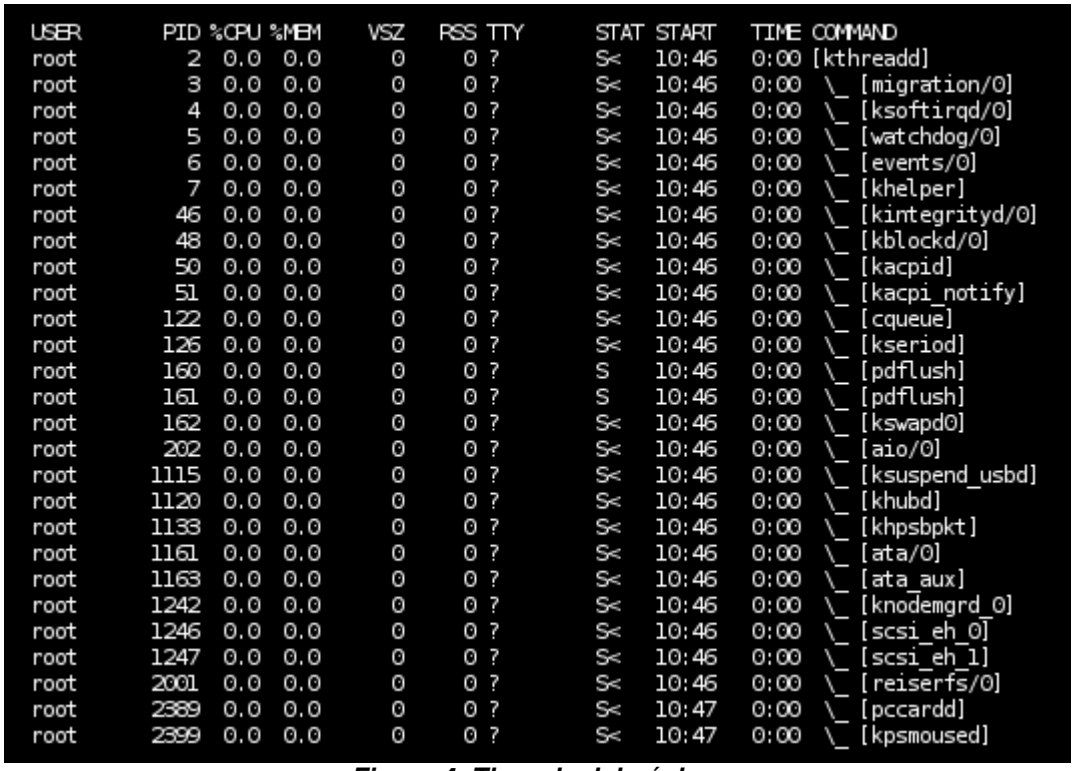
Los semáforos permiten que una o varias tareas accedan a una misma sección crítica, por tanto, un semáforo nunca está limitado al uso de un solo thread. En GNU/Linux, los semáforos son usados habitualmente para mantener la exclusión mútua en el uso de un recurso o bien para coordinar dos o más threads. En concreto, para la exclusión mútua, la característica de los semáforos de poder manejar múltiples threads produce un tiempo añadido innecesario que demora el tiempo de latencia del sistema. Es por este motivo que el parche RT-preempt introduce una nueva primitiva para el núcleo Linux llamada ***mutex***(ver Apéndice A). Debido a su diseño simple, un mutex es más ligero y rápido que un semáforo, y a diferencia de éste último, un mutex solo puede ser usado por un único thread. Esta propiedad del mutex es uno de los requisitos básicos para la implementación de la herencia de prioridad que el parche RT-preempt lleva a cabo. A partir del momento en el que el lock tiene una relación uno a uno con su propietario, la cadena de herencia de prioridad es simple y directa, y no se ramifica entre diferentes dueños que deban esperar a heredar nuevas prioridades.

3.3.3.1.3 Rutinas de servicios de interrrupción expropiativas.

Cuando se ejecuta un manejador de interrupciones o bien las operaciones aplazadas de la parte bottom-half de la línea principal del núcleo Linux, éste se encuentra en contexto de interrupción. Aunque el código que se ejecuta en un contexto de interrupción debe ser rápido y simple, el servicio y acceso a dispositivos de baja prioridad como, por ejemplo, discos duros, en el momento de gestionar la interrupción puede causar tiempos de latencia excesivamente largos para todas las tareas del sistema.

En general, dentro del núcleo Linux las tareas más prioritarias pueden verse severamente afectadas por las tareas de menor prioridad. Es por todo esto que el parche RT-preempt convierte las rutinas de servicio de interrupción en threads del núcleo.

Los threads del núcleo son usados para llevar a cabo operaciones en segundo plano. Por ejemplo, el vaciado de las caches de disco y la ejecución de tasklets son ejecutados intermitentemente por los threads del núcleo pdflush y ksoftirqd, respectivamente (ver Figura 4).



USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	2	0.0	0.0	0	0	?	S<	10:46	0:00	[kthreadd]
root	3	0.0	0.0	0	0	?	S<	10:46	0:00	_ [migration/0]
root	4	0.0	0.0	0	0	?	S<	10:46	0:00	_ [ksoftirqd/0]
root	5	0.0	0.0	0	0	?	S<	10:46	0:00	_ [watchdog/0]
root	6	0.0	0.0	0	0	?	S<	10:46	0:00	_ [events/0]
root	7	0.0	0.0	0	0	?	S<	10:46	0:00	_ [khelper]
root	46	0.0	0.0	0	0	?	S<	10:46	0:00	_ [kintegrityd/0]
root	48	0.0	0.0	0	0	?	S<	10:46	0:00	_ [kblockd/0]
root	50	0.0	0.0	0	0	?	S<	10:46	0:00	_ [kacpid]
root	51	0.0	0.0	0	0	?	S<	10:46	0:00	_ [kacpi_notify]
root	122	0.0	0.0	0	0	?	S<	10:46	0:00	_ [cqueue]
root	126	0.0	0.0	0	0	?	S<	10:46	0:00	_ [kseriod]
root	160	0.0	0.0	0	0	?	S	10:46	0:00	_ [pdflush]
root	161	0.0	0.0	0	0	?	S	10:46	0:00	_ [pdflush]
root	162	0.0	0.0	0	0	?	S<	10:46	0:00	_ [kswapd0]
root	202	0.0	0.0	0	0	?	S<	10:46	0:00	_ [aio/0]
root	1115	0.0	0.0	0	0	?	S<	10:46	0:00	_ [ksuspend_usbd]
root	1120	0.0	0.0	0	0	?	S<	10:46	0:00	_ [khubd]
root	1133	0.0	0.0	0	0	?	S<	10:46	0:00	_ [khpsbpkt]
root	1161	0.0	0.0	0	0	?	S<	10:46	0:00	_ [ata/0]
root	1163	0.0	0.0	0	0	?	S<	10:46	0:00	_ [ata_aux]
root	1242	0.0	0.0	0	0	?	S<	10:46	0:00	_ [knodemgrd_0]
root	1246	0.0	0.0	0	0	?	S<	10:46	0:00	_ [scsi_eh_0]
root	1247	0.0	0.0	0	0	?	S<	10:46	0:00	_ [scsi_eh_1]
root	2001	0.0	0.0	0	0	?	S<	10:46	0:00	_ [reiserfs/0]
root	2389	0.0	0.0	0	0	?	S<	10:47	0:00	_ [pccardd]
root	2399	0.0	0.0	0	0	?	S<	10:47	0:00	_ [kpsmoused]

Figura 4: Threads del núcleo.

Los threads del núcleo se diferencian de las tareas estándar por ejecutarse única y exclusivamente en modo núcleo, como parece lógico, por otra parte, y además carecen de espacio de direcciones. Pero, en cualquier caso y a pesar de estas diferencias comentadas, los threads del núcleo son planificables y expulsables al igual que las tareas normales.
A continuación se pasará a explicar cómo el parche convierte en threads tanto las operaciones críticas de la interrupción llevadas a cabo en su parte top-half, como las operaciones aplazadas realizadas mediante la parte bottom-half.

- **Conversión de manejadores top-half en threads del núcleo.**
  Un driver de un dispositivo registra un manejador de interrupción y activa una línea de interrupción para su gestión llamando a la función ***request_irq()***. Esta función, a su vez, invoca a la función ***setup_irq()*** que registra la estructura *struct irqaction*.
  Entre otras operaciones, la función ***setup_irq()*** llama a ***start_irq_thread()*** para crear un thread de núcleo que será el encargado de gestionar la línea de interrupción correspondiente. El trabajo que realiza el thread se implementa en la función ***do_irqd()***. Solo un único thread puede ser creado por línea de interrupción, y las interrupciones compartidas son gestionadas, también, por un solo thread.

  Desde el momento en el que los threads del núcleo son ejecutados en contexto de ejecución, los manejadores de interrupción son ahora susceptibles de planificación y expulsión.

  Casi todos los manejadores de interrupción se ejecutan como un thread del núcleo en el sistemas GNU/Linux parcheados con RT-preempt. En cualquier caso, existen situaciones donde el manejador debe ser servido en el contexto de interrupción.
  Mediante el establecimiento del descriptor de interrupción IRQ_NODELAY, el manejador es forzado a ejecutarse en el contexto de la interrupción y no como un

thread. Una interrupción IRQ_NODELAY puede afectar severamente a los tiempos de latencia del sistema, tanto al tiempo invertido en la planificación como al tiempo invertido en la gestión de la interrupción. El uso más notable de este tipo de interrupciones se halla en las interrupciones de reloj, precisamente, por su estrecha relación con la planificación de tareas y otras operaciones internas del núcleo. Esta es otra de las salvedades comentadas en el punto 3.3.2, donde la aplicación del parche aún no ha podido convertir todo el código del núcleo en un bloque absolutamente expropiativo.

- **Conversión de manejadores bottom-half en threads del núcleo.**
  En relación a este tipo de manejadores cabe destacar que el parche RT-preempt convierte el mecanismo *softirq* en threads del núcleo. Cada manejador tendrá su propio thread para realizar el servicio pertinente.

3.3.3.2 Temporizadores de alta resolución

De forma previa a la explicación, se presenta en la siguiente figura el actual sistema de gestión del tiempo en el núcleo Linux:

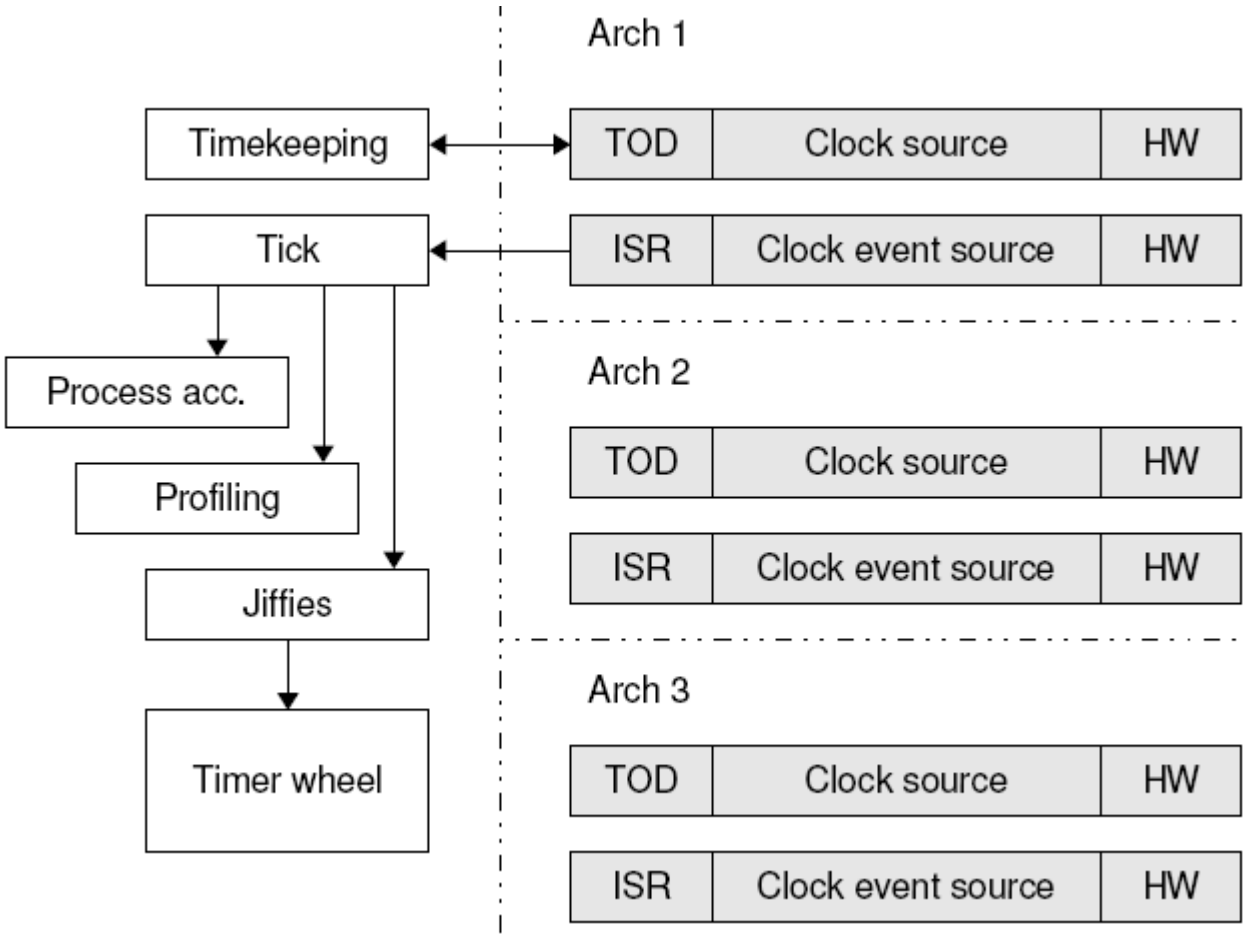


Figura 5: Sistema clásico de gestión del tiempo.

Las metas perseguidas en el diseño y elaboración de los ***hrtimers*** son:

- Simplicidad.
- Disponer de una estructura de datos no limitada a jiffies o cualquier otra granularidad. Toda la lógica del núcleo cuenta con una resolución de nanosegundos

con 64-bits de espacio.
- Simplificación del actual código del núcleo relativo a la gestión del tiempo.

Otro de los requisitos básicos de ***hrtimers*** es la inmediata acción de encolar y ordenar los temporizadores por tiempo de activación. Con el objetivo de materializar esta premisa se usan los árboles **RB** (Red-Black trees). Los árboles RB están disponibles en la librería del núcleo ***include/linux/rbtree.h*** y son usados en diferentes áreas críticas del sistema como, por ejemplo, en la gestión de memoria y en el sistema de ficheros. Los árboles RB se usan únicamente para la ordenación de los temporizadores en función del tiempo de activación, mientras que una lista separada aglutina el código a ejecutar con la finalidad de proporcionar un mejor acceso a los temporizadores encolados, sin tener que buscar en el árbol RB. Esta lista también se usa para almacenar los relojes de alta resolución, en aras a preservar el orden temporal de activación y tener listas de relojes pendientes y expirados.

El valor de los tiempos se almacenan en unos nuevos tipos de datos llamados ***ktime_t***, que usan como base el nanosegundo.
El tipo ***ktime_t*** está optimizado para poder utilizase tanto en arquitecturas de 32 bits como de 64. La tupla segundo, nanosegundo está permitida en ambas arquitecturas.

La funcionalidad que el núcleo ofrece para usar los ***hrtimers*** es:

- nanosleep
- itimers
- posix-timers

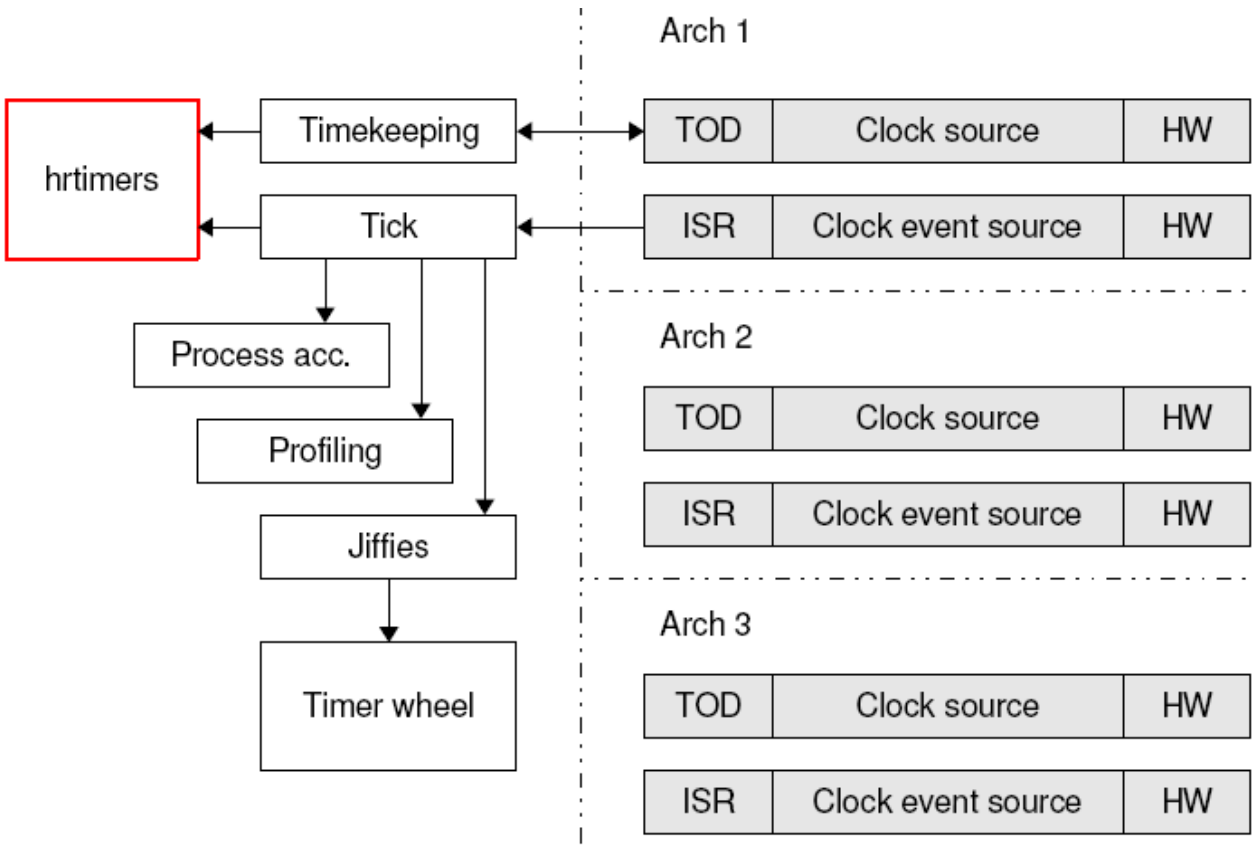


Figura 6: Sistema de gestión de tiempo con hrtimers.

Con todo lo expuesto, ***hrtimers*** soluciona diferentes handicaps existentes:

- Solventa la dependencia que se tiene con el tick del reloj.
- Crea un marco genérico para la programación de los eventos de interrupción.
- Remplaza la interrupción periódica de reloj por un temporizador.

En los ***hrtimers*** también se encuentra el desarrollo de John Stultz que proporciona un nuevo marco para el cálculo del tiempo diario (Generic Time of Day-GTOD). Este marco es independiente de la arquitectura y tiene una mejor desconexión del tick del sistema (ver Figura 7).

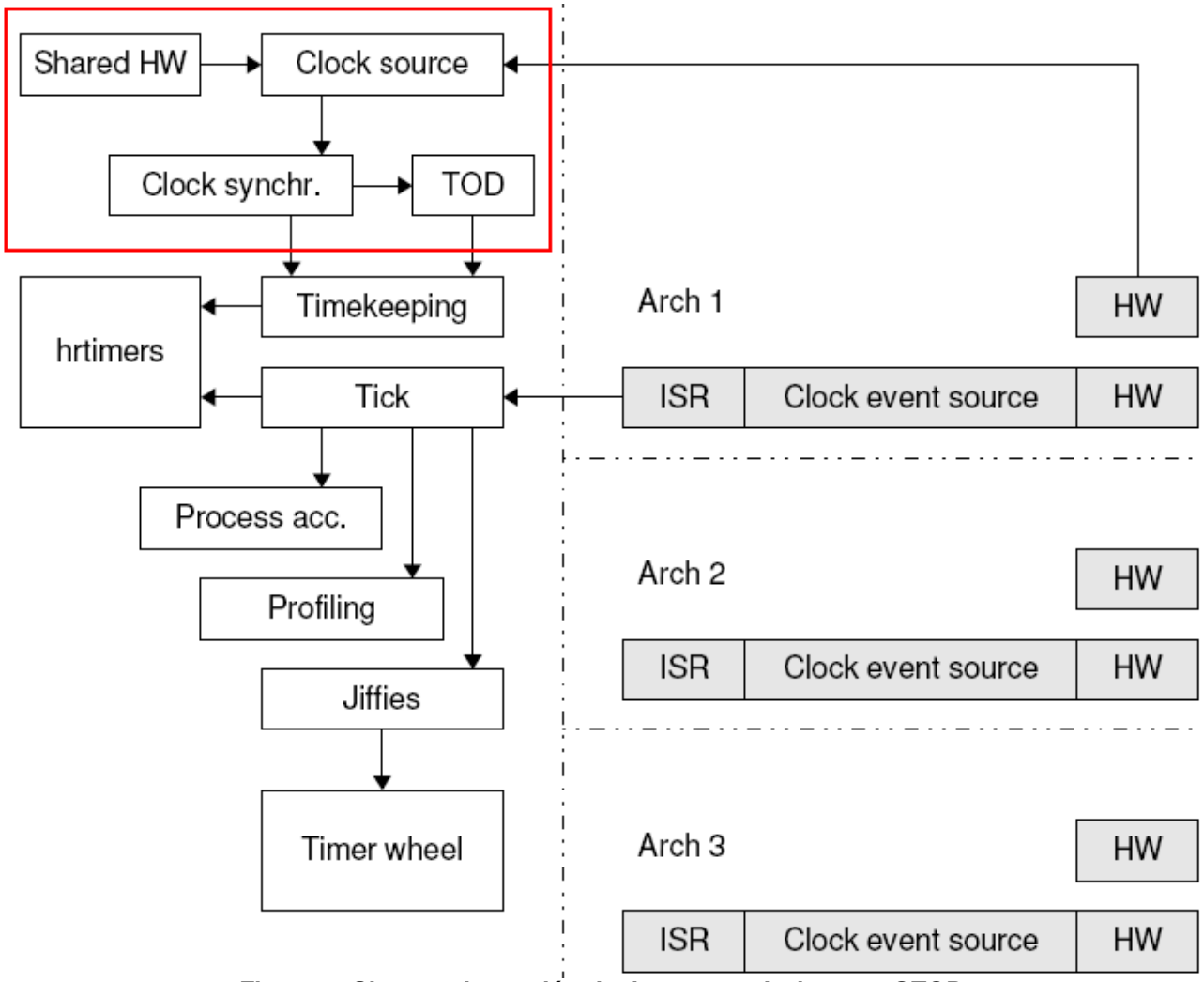


Figura 7: Sistema de gestión de tiempo con hrtimers y GTOD.

Los eventos de reloj, antes mencionados, conforman una infraestructura genérica para distribuir los diferentes eventos relacionados con un temporizador. Esta infraestructura también es independiente de la arquitectura y permite una mejor selección de hardware para los eventos.
Se usa un ***hrtimer*** por CPU para emular el tick de la misma. Esto permite actualizar los jiffies y realizar ajustes en NTP (*Network Time Protocol*), además de posibilitar el uso de ticks dinámicos, analizados en el siguiente apartado.

La Figura 8 muestra el resultado final del sistema de gestión de tiempo tras aplicar las modificaciones del parche RT-preempt.

En resumen, el parche RT-preempt en relación a los temporizadores de alta resolución proporciona:

- Parche base de hrtimers.
- Modificación de timeofday de John Stultz.
- Capa de abstracción de eventos de reloj.
- Tipos ktime_t y otras estructuras de datos añadidas.
- Bits de alta precisión para arquitecturas i386.

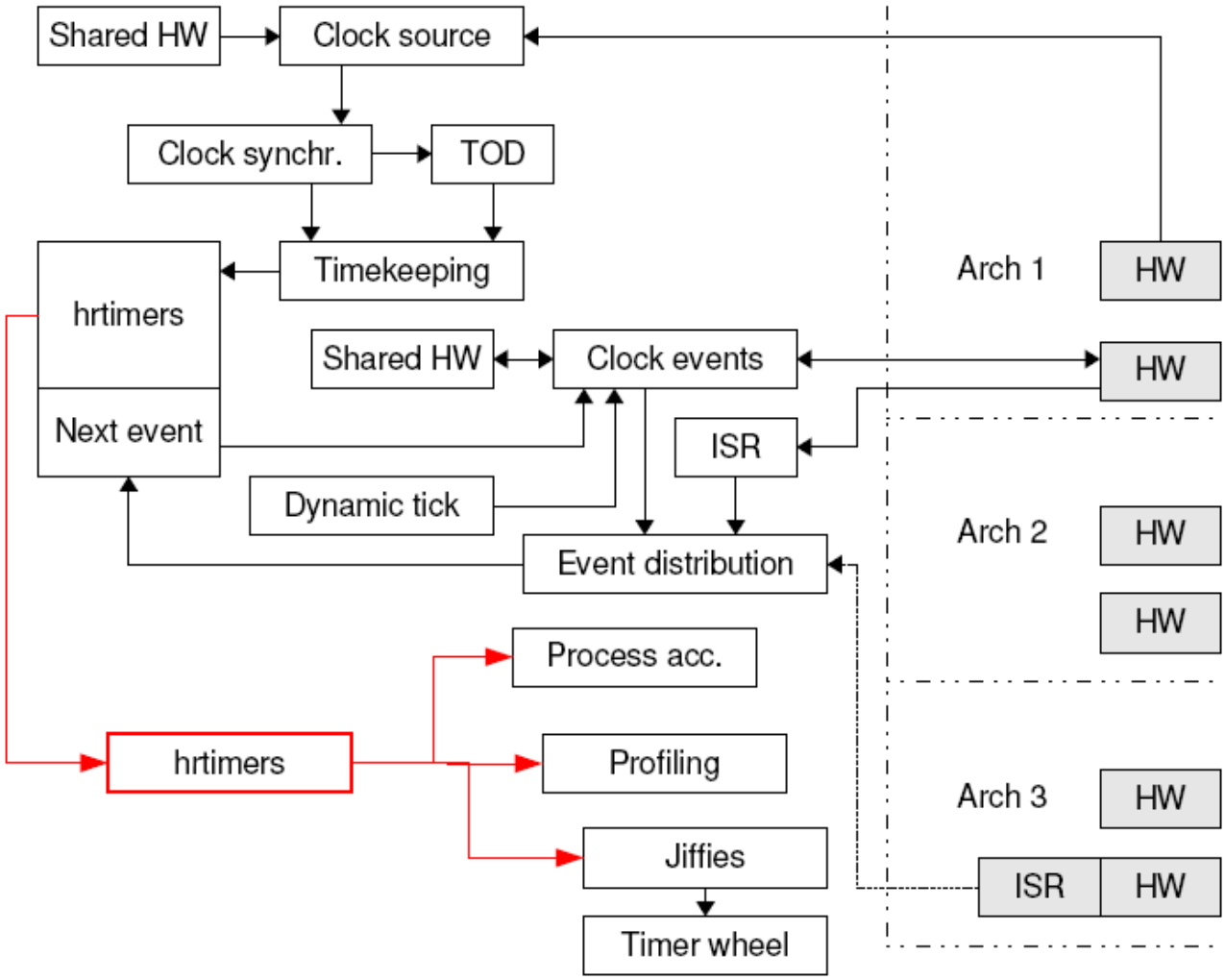


Figura 8: Sistema de gestión de tiempo final.

3.3.3.3 Ticks dinámicos

Los ticks dinámicos del sistema están basados en las modificaciones llevadas a cabo por el parche llamado "*Tiempos límite de planificación variables*" (*Variable Scheduling Timeout-**VST***). Esta modificación posibilita la supresión de los ticks de la CPU en los períodos en los que no tenga nada que procesar, es decir, el los períodos ***idle***.

Cuando el sistema entra en un estado de inactividad, la estrategia es encontrar el siguiente temporizador que tenga un tiempo de expiración lo más, razonablemente, lejano (al menos más de un tick en el futuro) como para poder desactivar las interrupciones de reloj periódicas. En este momento, la interrupción de reloj es reprogramada para que sea activada en el momento en que el temporizador expire.

La implementación de los ticks dinámicos que hace el parche RT-preempt no depende de

ningún otro parche y se realiza mediante la llamada **tick_nohz_stop_sched_tick()**.
Esta función es invocada desde la función **cpu_idle()** en ***arch/i386/kernel/process.c***, o bien desde la función **irq_exit()**, cuando un período de inactividad de CPU es interrumpido por una interrupción que no provoca una replanificación.
La función **tick_nohz_restart_sched_tick()** reinicia el tick periódico cuando el procesador ha finalizado su estado de inactividad.
La funcionalidad de los eventos de reloj de hrtimers para los ticks dinámicos está siempre disponible aunque no estén activados los temporizadores de alta resolución.

```
void cpu_idle(void){
        ...
        while (1) {
                tick_nohz_stop_sched_tick();
                while (!need_resched() && !need_resched_delayed()) {
                        void (*idle)(void);
                        ...
                        __get_cpu_var(irq_stat).idle_timestamp = jiffies;
                        idle();
                }
                local_irq_disable();
                trace_preempt_exit_idle();
                tick_nohz_restart_sched_tick();
                __preempt_enable_no_resched();
                __schedule();
                ...
}
```

3.4 COMPILACIÓN DEL NÚCLEO LINUX CON PARCHE RT-PREEMPT

En este apartado se detallarán las acciones realizadas para compilar un núcleo Linux estándar(Vanilla Kernel) con la funcionalidad aportada por el parche RT-preempt.
Antes de iniciar la explicación, hay que reseñar que se ha elegido un núcleo estándar, es decir, un Vanilla Kernel, porque este tipo de núcleos no han sufrido ningún tipo de modificación, hecho que no ocurre con los núcleos que se pueden encontrar en el interior de cualquier distribución del sistema GNU/Linux. La gran mayoría de distribuciones personalizan el núcleo Linux aplicando múltiples parches con el objetivo de facilitar nuevas funcionalidades al usuario final, ya sea para simplificar el uso del sistema o bien para hacerlo más atractivo.
La versión del núcleo escogido es la 2.6.21.6 y se puede descargar de la dirección siguiente:

http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.21.6.tar.bz2

La versión del parche RT-preempt es la misma, lógicamente, que la versión del núcleo estándar seleccionado:

http://www.kernel.org/pub/linux/kernel/projects/rt/older/patch-2.6.21.6-rt21

Una vez conseguido el fichero del núcleo se procede a su desempaquetado:

```
$ tar xjvf linux-2.6.21.6.tar.bz2
```

Se renombra el directorio que contiene todas las fuentes del núcleo descargado para denotar la que va ser su característica principal, esto es, ser un sistema operativo de tiempo real:

```
$ mv linux-2.6.21.6 linux-2.6.21.6-rt21
```

A continuación se aplica el parche RT-preempt:

```
$ cd linux-2.6.21.6-rt21
$ patch -p1 < ../patch-2.6.21.6-rt21
```

En este último paso es posible que, en función de la distribución de trabajo usada, no se tenga la utilidad *patch* instalada de antemano. En este tipo de situaciones habría que descargar e instalar la utilidad, o bien manualmente, o bien usando algún gestor de paquetes que el sistema pueda albergar.

A partir de este momento ya se está en condiciones de configurar las opciones deseadas del nuevo núcleo parcheado y de iniciar la compilación del mismo.
Para seleccionar las opciones que convertirán el núcleo Linux estándar en un núcleo con capacidad para tiempo real estricto, se debe, en primer lugar, limpiar cualquier rastro de compilaciones anteriores:

```
$ cd linux-2.6.21.6-rt21
$ make clean
$ make mrproper
```

y en segundo lugar, ejecutar la herramienta:

```
$ make menuconfig
```

En ambos casos cabe destacar que pueden aparecer ciertos contratiempos por no disponer del software necesario. Por ejemplo, el comando *make* no siempre está preinstalado en el sistema, entonces, se deberá hacer uso, nuevamente, de algún gestor de paquetes que el sistema tenga o bien, descargar e instalar manualmente el software necesario.

Para poder usar la opción *menuconfig*, que presenta las opciones del núcleo en modo texto con menús, se requiere la librería de desarrollo Ncurses que se encuentra en el paquete *libncurses5-dev*.

Cabe decir, que en casos donde se requiera recompilar el núcleo previamente instalado, se puede hacer uso del archivo de configuración de opciones existente. Para preservar estas opciones y añadir las de tiempo real, u otras, se debe copiar el archivo de configuración al directorio del núcleo a compilar:

```
$ cp /boot/config-`uname -r` .config
```

Sin embargo, esta opción no es recomendable si se ha cambiado de versión de núcleo, porque probablemente hayan cambiado las opciones de configuración.

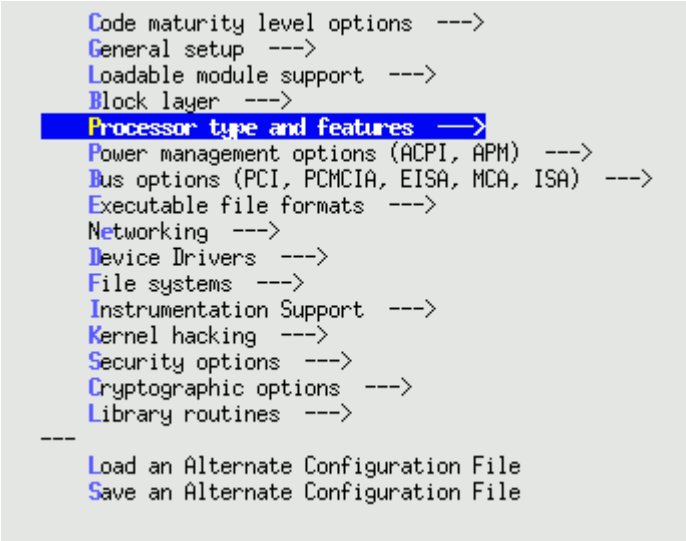


Figura 9: Menú principal de opciones.

Una vez dentro del menú de configuración de opciones(ver Figura 9), se debe seleccionar la opción *Complete preemption(Real-Time)* que se encuentra en *Processor type and features → Preemption Mode*. En las siguientes figuras se observa la secuencia de acciones a ejecutar:

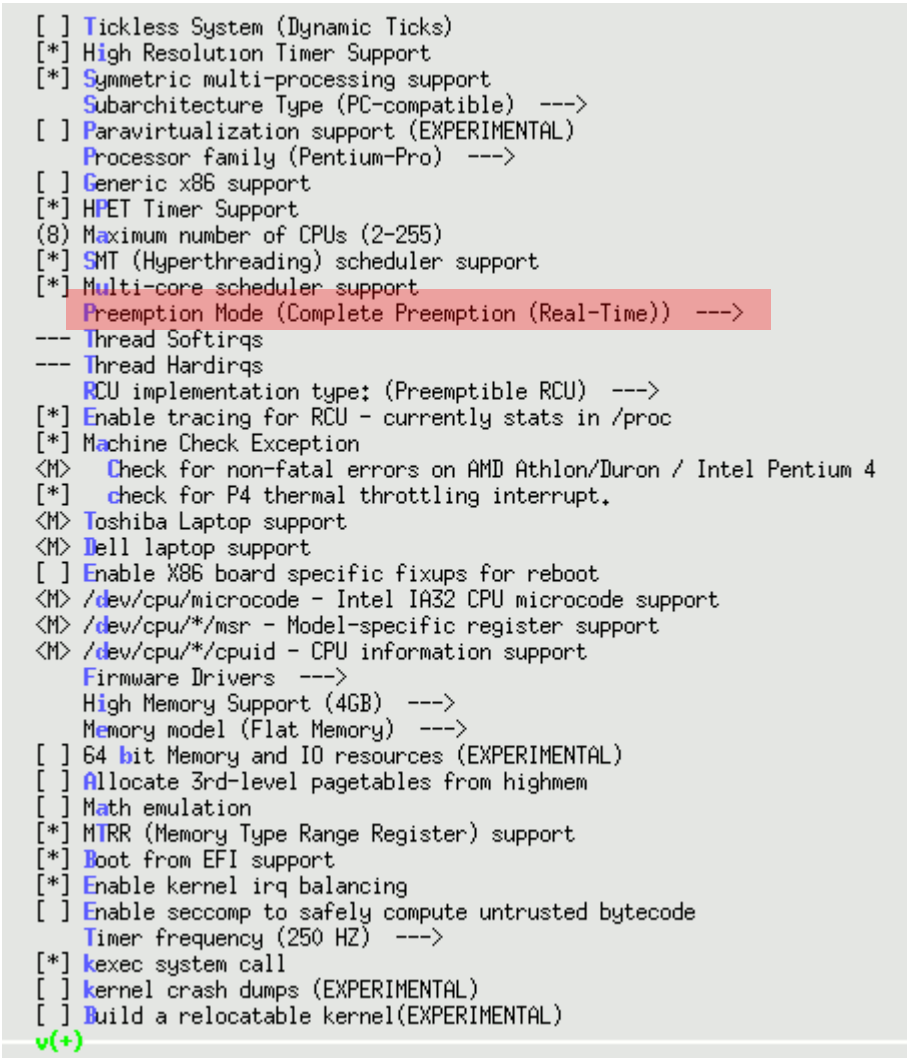


Figura 10: Opciones de configuración del procesador.

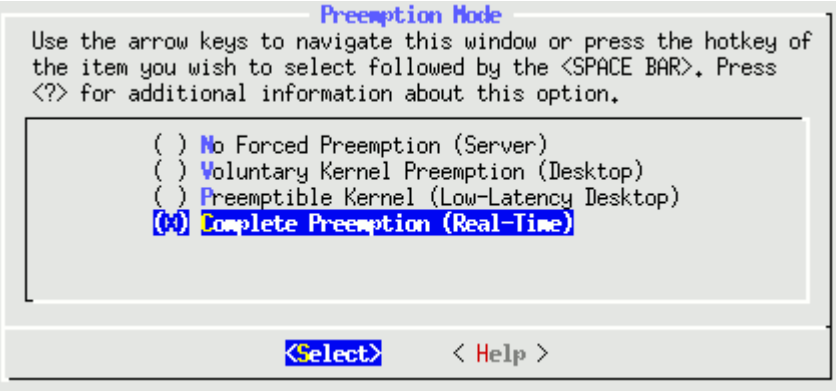


Figura 11: Opción de configuración para tiempo real estricto.

Existe la opción de activar una serie de herramientas para llevar a cabo actividades de depuración del núcleo, así como, realizar trazas de ciertas partes del código relativas a la capacidad de tiempo real. También existen herramientas de diagnóstico que pueden ser activadas. Todas estas opciones se encuentran en la categoría *kernel hacking* de la ventana principal de *menuconfig.* En la figura siguiente se pueden ver las opciones *Wakeup latency timing*, *Latency tracing* y *wakeup latency histogram*, que son las que posibilitan llevar a cabo las acciones previamente señaladas.

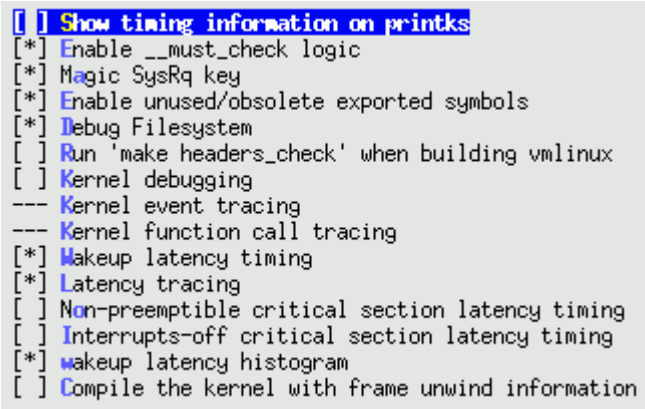


Figura 12: Opciones de depuración.

Antes de compilar el nuevo núcleo se crea el enlace */usr/src/linux* apuntando al directorio de las fuentes, ya que durante la compilación algunos programas buscan este directorio:

```
$ ln -s /usr/src/kernels/linux-2.6.21.6-rt21 /usr/src/linux
```

Y, a continuación, ya se está en disposición de compilar el núcleo y los módulos, en su caso, usando el comando *make*:

```
$ make
$ make modules_install
```

Este proceso puede tardar un tiempo considerable, en función del ordenador usado, y genera:

- Un núcleo comprimido : */usr/src/kernels/linux-2.6.21.6-rt21/arch/i386/boot/bzImage*
- El archivo de símbolos : */usr/src/kernels/linux-2.6.21.6-rt21/System.map*
- Numerosos archivos .o y .ko, que son módulos, y subdirectorios.

Tras la compilación se deben llevar a cabo algunas tareas para proceder a la instalación del nuevo núcleo en el sistema GNU/LINUX.:

- Instalar el nuevo núcleo *vmlinuz*:

```
$ cp /usr/src/kernels/linux-2.6.21.6-rt21/arch/i386/boot/bzImage \
> /boot/vmlinuz-2.6.21.6-rt21
```

- Instalar el nuevo archivo de símbolos *System.map*:

```
$ cp /usr/src/kernels/linux-2.6.21.6-rt21/System.map \
> /boot/System.map-2.6.21.6-rt21
```

- Instalar el archivo de configuración del kernel:

```
$ cp /usr/src/kernels/linux-2.6.21.6-rt21/.config /boot/config-2.6.21.6-rt21
```

- Crear la imagen de arranque i*nitrd.img* para el nuevo núcleo con la utilidad *mkinitramfs*:

```
$ mkinitramfs -o /boot/initrd.img-2.6.21.6-rt21 /lib/modules/2.6.21.6-rt21
```

- Actualizar el enlace simbólico */vmlinuz*, borrando el anterior:

```
$ rm /vmlinuz
$ ln -s /boot/vmlinuz-2.6.21.6-rt21 /vmlinuz
```

- Actualizar el enlace simbólico */initrd.img*, borrando el anterior:

```
$ rm /initrd.img
$ ln -s /boot/initrd.img-2.6.21.6-rt21 /initrd.img
```

- Crear el enlace simbólico */boot/System.map* que inicialmente no existe:

```
$ ln -s /boot/System.map-2.6.21.6-rt21 /boot/System.map
```

- Configurar el gestor de arranque, en el caso que nos ocupa GRUB, para poder arrancar con el nuevo núcleo. Se edita el fichero */boot/grub/menu.lst* y se añade la entrada para el núcleo Linux de tiempo real:

```
title           Debian GNU/Linux, kernel 2.6.21.6-rt21 (on /dev/sda1)
root            (hd0,0)
kernel          /boot/vmlinuz-2.6.21.6-rt21 root=/dev/hda1 ro
initrd          /boot/initrd.img-2.6.21.6-rt21
savedefault
boot
```

Con esta serie de modificaciones y sin haber alterado las opciones previamente existentes el sistema operativo GNU/Linux tiene todo lo necesario para arrancar con el nuevo núcleo con capacidad hard real-time.

Para comprobar que todo el proceso de compilación e instalación ha sido satisfactorio, una vez arrancado el sistema y habiendo seleccionado el nuevo núcleo en el gestor GRUB, se puede ejecutar el siguiente comando en la shell,

```
$ uname -a
Linux hoth 2.6.21.6-rt21 #1 SMP PREEMPT RT Sun Nov 30 21:33:34 GMT
2008 i686 GNU/Linux
```

donde en la salida devuelta se observa claramente la característica **PREEMPT RT**.
También se puede ver el contenido del fichero *version* que se encuentra en el pseudo sistema de ficheros */proc*:

```
$ cat /proc/version
Linux version 2.6.21.6-rt21 (root@hoth) (gcc version 4.1.2 20061115
(prerelease) (Debian 4.1.1-21)) #1 SMP PREEMPT RT Sun Nov 30 21:33:34
GMT 2008
```

En ambos casos se puede observar que la opción de expropiación(**PREEMPT**) del núcleo y la opción de tiempo real(**RT**) están activadas.

Otra forma de verificar que todo ha ido bien es usando las herramientas que el propio núcleo ofrece. Por una parte, se puede observar que la variable global del núcleo *kernel_preemption* tiene, ahora, como valor 1:

```
$ cat /proc/sys/kernel/kernel_preemption
1
```

Por otra parte, usando las utilidades que se han activado en la opción *kernel hacking*, se puede observar si el comportamiento global del sistema es el correcto. El fichero *preempt_max_latency* proporciona una primera aproximación(en microsegundos):

```
$ cat /proc/sys/kernel/preempt_max_latency
341
```

En el fichero *latency_trace*, que se encuentra en */proc*, se muestra la peor de las latencias observadas desde la activación del *latency tracing*:

```
$ cat /proc/latency_trace
```

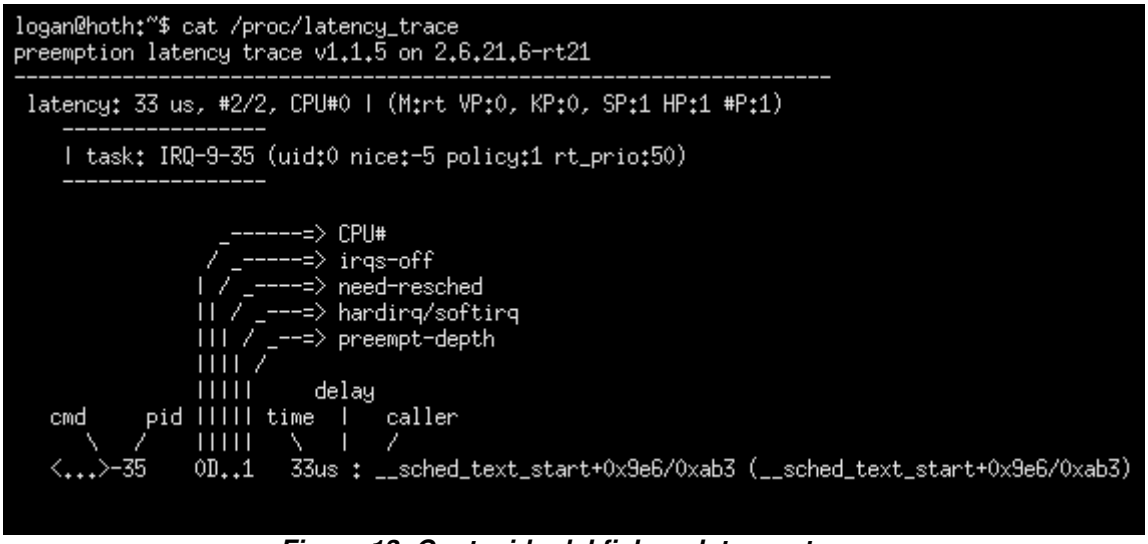


Figura 13: Contenido del fichero latency_trace

Como indicación para saber si el peor tiempo de latencia contenido en el fichero *latency_trace* está en el rango aceptado para un sistema operativo de tiempo real, se puede comparar con el resultado de la operación **10^5 / frecuencia de CPU.**

Por ejemplo, para una CPU de 1GHz de frecuencia cabe esperar una latencia máxima de 100 microsegundos o, en el caso del ordenador usado para elaborar este trabajo, cabría esperar una latencia máxima de 50 microsegundos (CPU de 2GHz, 10^5/2GHz = 50µs).

Por tanto, se observa que la latencia máxima está por debajo de lo esperado, hecho que implica que el comportamiento del sistema es el correcto.

Por último, se pueden usar histogramas de latencias, también instalados tras la activación de las opciones de *kernel hacking*. Estos histogramas se encuentran en el fichero */proc/latency_hist/wakeup_latency/CPU0*, en el caso del ordenador usado, ya que es monoprocesador.

Ejecutando en la shell el siguiente comando se obtiene una lista de la cantidad de tareas ejecutadas en el sistema y de sus tiempos de latencia (ver Figura 14):

```
$ grep -v '0$' /proc/latency_hist/wakeup_latency/CPU0
```

Con todo lo expuesto se puede estar en disposición de afirmar que el sistema operativo se comporta como un sistema de tiempo real esctricto.

```
#Minimum latency: 2 microseconds.
#Average latency: 7 microseconds.
#Maximum latency: 33 microseconds.
#Total samples: 649642
#There are 0 samples greater or equal than 10240 microseconds
#usecs samples
  2                25
  3            130647
  4              9978
  5             14436
  6              5035
  7             95063
  8            156128
  9            199353
 10             14644
 11              4138
 12              3461
 13              2414
 14              2923
 15              4438
 16              1983
 17              2465
 18              1083
 19               401
 20               396
 21               477
 22                79
 24                14
 25                 7
 26                 3
 27                 2
 28                 4
 29                 1
 30                 1
 31                 2
 33                 1
```

Figura 14: Resultado de /proc/latency_hist/wakeup_latency/CPU0

3.5 TESTING DE TIEMPOS DE RESPUESTA

El objetivo de este apartado es comprobar las capacidades de tiempo real estricto del núcleo Linux compilado con las funcionalidades del parche RT-preempt mediante una herramienta externa de testeo.
La utilidad escogida para efectuar las pruebas de comportamiento es *cyclictest*. Esta herramienta ha sido desarrollada por Thomas Gleixner, encargado del diseño y mantenimiento de los temporizadores de alta resolución, como se ha comentado en el apartado 3.2.

Con *cyclictest* se abordará la medición de los tiempos de latencia involucrados en las operaciones de dormir y despertar de threads con una prioridad elevada. La unidad de los tiempos de respuesta es de microsegundos.
Esta utilidad puede ser descargada de la siguiente dirección:

http://www.kernel.org/pub/linux/kernel/people/tglx/rt-tests/rt-tests-0.28.tar.gz

Una vez se disponga del paquete se debe descomprimir y compilar la herramienta en cuestión.

```
$ tar xzvf rt-tests-0.28.tar.gz
$ cd rt-tests
$ make cyclictest
```

Para comprobar la funcionalidad aportada por el parche RT-preempt se evaluarán tres núcleos diferentes:

- Un núcleo Linux 2.6.27-7-generic de una distribución Ubuntu 8.10 (Intrepid Ibex), que ya dispone de los temporizadores de alta resolución de los que se ha hablado en el apartado 3.3.3.2.
- Un núcleo Linux 2.6.18-6-686 de una distribución Debian 4.0r3 GNU/Linux sin temporizadores de alta resolución.
- Un núcleo Linux 2.6.21.6-rt sobre una distribución Debian GNU/Linux compilado con el parche RT-preempt.

Antes de pasar a la comparativa del comportamiento de estos tres núcleos se comentarán brevemente las opciones usadas de *cyclictest*.
El comando, junto con los parámetros utilizados, ha sido:

```
# ./cyclictest -n -q -t10 -p99 -i1000 -l10000 > test
```

Opciones:

- **-n**
  Con esta opción se usa el temporizador *nanosleep* en vez del *interval timers* de POSIX. En concreto, con esta opción, se llama a la función *clock_nanosleep* y no a la función estándar de POSIX.
- **-q**
  Ejecución silenciosa, solo se presenta por pantalla el sumario final de la ejecución.
- **-t#THREADS**
  Establece el número de threads que serán lanzados en la ejecución de *cyclictest*. En el caso de las pruebas realizadas en número de threads lanzados es 10.
- **-p#PRIO**
  Establece la prioridad con la que será creado cada uno de los threads. Al primero de los threads creados se le asignará la prioridad que acompaña al parámetro y al resto un valor decreciente a partir de la prioridad inicial (thread 1 con prioridad N, thread 2 con prioridad N-1, thread 3 con prioridad N-3, etc).
- **-i#INTV**
  Establece la base de tiempo de cada uno de los temporizadores de los threads (el valor por defecto es 1000 microsegundos).
- **-l#LOOPS**
  Establece el número de ciclos que los threads deberán ejecutar en la CPU. Por defecto es 0, es decir, un bucle sin fin. Esta opción es útil para automatizar tests con un número dado de pruebas cíclicas. *Cyclictest* termina cuando el primer thread consigue llegar al número de vueltas establecido.

Una vez comentadas las opciones usadas se procederá a mostrar los resultados obtenidos en los tres diferentes escenarios:

```
T: 0 ( 5526) P:99 I:1000 C:  10000 Min:     17 Act:   36 Avg:   34 Max:     563
T: 1 ( 5527) P:98 I:1500 C:   6672 Min:     17 Act:   36 Avg:   33 Max:     451
T: 2 ( 5528) P:97 I:2000 C:   5004 Min:     17 Act:   36 Avg:   32 Max:     357
T: 3 ( 5529) P:96 I:2500 C:   4003 Min:     17 Act:   32 Avg:   33 Max:     366
T: 4 ( 5530) P:95 I:3000 C:   3336 Min:     25 Act:   31 Avg:   34 Max:     311
T: 5 ( 5531) P:94 I:3500 C:   2860 Min:     20 Act:   36 Avg:   34 Max:     382
T: 6 ( 5532) P:93 I:4000 C:   2502 Min:     19 Act:   32 Avg:   33 Max:     221
T: 7 ( 5533) P:92 I:4500 C:   2224 Min:     22 Act:   32 Avg:   34 Max:     297
T: 8 ( 5534) P:91 I:5000 C:   2002 Min:     19 Act:   34 Avg:   33 Max:     220
T: 9 ( 5535) P:90 I:5500 C:   1820 Min:     25 Act:   32 Avg:   35 Max:     210
```

Figura 15: Resultados del núcleo 2.6.27-7

```
T: 0 ( 3178) P:99 I:10000 C:  10000 Min:     42 Act: 7860 Avg: 5930 Max:    8149
T: 1 ( 3179) P:98 I:10500 C:   9526 Min:     44 Act: 5347 Avg: 5913 Max:    8064
T: 2 ( 3180) P:97 I:11000 C:   9093 Min:     48 Act: 5845 Avg: 5928 Max:    8028
T: 3 ( 3181) P:96 I:11500 C:   8698 Min:     93 Act: 6333 Avg: 5930 Max:    8129
T: 4 ( 3182) P:95 I:12000 C:   8336 Min:     64 Act: 5826 Avg: 5759 Max:    8114
T: 5 ( 3183) P:94 I:12500 C:   8002 Min:     64 Act: 5328 Avg: 5936 Max:    8253
T: 6 ( 3184) P:93 I:13000 C:   7695 Min:     85 Act: 7811 Avg: 5945 Max:    8112
T: 7 ( 3185) P:92 I:13500 C:   7410 Min:     70 Act: 4310 Avg: 5930 Max:    8238
T: 8 ( 3186) P:91 I:14000 C:   7145 Min:     86 Act: 5803 Avg: 5970 Max:    8219
T: 9 ( 3187) P:90 I:14500 C:   6899 Min:     92 Act: 4803 Avg: 5938 Max:    8223
```

Figura 16: Resultados del núcleo 2.6.18-6

```
T: 0 ( 3549) P:99 I:1000 C:  10000 Min:     15 Act:   21 Avg:   22 Max:     38
T: 1 ( 3550) P:98 I:1500 C:   6668 Min:     13 Act:   21 Avg:   21 Max:     35
T: 2 ( 3551) P:97 I:2000 C:   5001 Min:     16 Act:   21 Avg:   21 Max:     32
T: 3 ( 3552) P:96 I:2500 C:   4001 Min:     15 Act:   21 Avg:   23 Max:     49
T: 4 ( 3553) P:95 I:3000 C:   3334 Min:     19 Act:   22 Avg:   21 Max:     37
T: 5 ( 3554) P:94 I:3500 C:   2858 Min:     15 Act:   21 Avg:   22 Max:     36
T: 6 ( 3555) P:93 I:4000 C:   2501 Min:     15 Act:   22 Avg:   21 Max:     30
T: 7 ( 3556) P:92 I:4500 C:   2223 Min:     15 Act:   21 Avg:   21 Max:     36
T: 8 ( 3557) P:91 I:5000 C:   2001 Min:     18 Act:   22 Avg:   22 Max:     33
T: 9 ( 3558) P:90 I:5500 C:   1819 Min:     18 Act:   22 Avg:   22 Max:     34
```

Figura 17: Resultados del núcleo 2.6.21.6-rt

Cabe denotar que en el caso de la distribución Debian con el núcleo 2.6.18, al no disponer de temporizadores de alta resolución, sigue presentando la limitación clásica de los núcleos Linux antiguos, es decir, su resolución no baja de 10ms (ver apartado 3.3.3.2). En consecuencia, para poder realizar la evaluación de los tiempos de latencia con "cyclictest" la opción -i ha tenido que establecerse en 10000 microsegundos para, de esta forma, obtener unos resultados comparables con las otras dos evaluaciones.

Como conclusión se muestra una tabla con los valores mínimo, máximo y promedio de los tiempos de latencia de planificación obtenidos en las diferentes pruebas realizadas.

Núcleo	Mínima(µs)	Máxima(µs)	Promedio(µs)
2.6.27-7	17	563	34
2.6.18-6	42	8253	5918
2.6.21.6-rt	13	49	22

En esta tabla se puede observar que el resultado obtenido en el núcleo parcheado con las modificaciones de tiempo real es el que presenta unos mejores tiempos de latencia y, por tanto, es el que puede garantizar el cumplimiento de los férreos requisitos temporales que caracterizan a las aplicaciones de tiempo real estricto.

CAPÍTULO 4
RESTRICCIONES Y ALTERNATIVAS

4.1 INTRODUCCIÓN

En el capítulo actual se presentarán dos materias diferentes que se ha creido conveniente apuntar para una, más exhaustiva, comprensión del planteamiento de este trabajo.
Por un lado, se expondrá el aspecto que, posiblemente, constituya uno de los escollos más relevantes a superar para poder usar el núcleo Linux en sistemas de tiempo real con una criticidad notable. Este handicap es el gestor de memoria dinámica que el núcleo Linux utiliza. En la sección 4.2 se explicarán los incovenientes y posibles peligros del uso de un gestor de memoria dinámica no diseñado explícitamente para sistemas de tiempo real, para, posteriormente, acabar presentando un nuevo gestor de memoria dinámica que cumple con todos los requisitos de determinismo necesarios de un sistema de tiempo real.
Por otro lado, en la sección 4.3 se enumerarán múltiples alternativas al uso del núcleo Linux modificado en sistemas de tiempo real.

4.2 GESTIÓN DE MEMORIA

Durante el desarrollo de este trabajo se han presentado las mejoras aportadas por el parche RT-preempt en un núcleo Linux estándar, y se han mencionado aquellas áreas que aún podrían estar en conflicto con un comportamiento de tiempo real estricto. Pero, no se ha hecho referencia a uno de los subsistemas vitales dentro de cualquier sistema operativo y al que no se le da ningún tipo de tratamiento en el parche *RT-preempt,* el sistema de memoria.
Es crucial para cualquier sistema operativo, ya sea de propósito general o de tiempo real, que en el instante en el que una tarea deba ser planificada y pase a ejecutarse, exista la memoria suficiente como para albergar su código, datos y pila. Es, entre otros, la resolución de este problema lo que se pretende abordar en este apartado.
Con el objetivo de contextualizar mejor la solución aportada, se procederá a realizar una breve introducción de los escollos reales a superar en la gestión de memoria, seguida de una somera presentación de diferentes gestores de memoria dinámica, para finalizar con la presentación de la alternativa a los gestores presentados, introduciendo el gestor Two-Level Segregated Fit (TLSF), diseñado e implementado por un grupo de ingenieros de la Universidad Politécnica de Valencia.

4.2.1 Definición del problema

Actualmente, el uso de memoria dinámica en sistemas de tiempo real es prácticamente inexistente. Por lo general, el único uso de memoria dinámica en los sistemas de tiempo real se produce durante su inicialización y terminación.
Cuando el uso de gestión de memoria dinámica es una necesidad impuesta, el único gestor considerado ha sido Binary Buddy, debido a su respuesta temporal logarítmica, $\mathbf{O(\log_2(n))}$.

Las causas que se apuntan para explicar esta situación suelen ser:

- Los gestores de memoria dinámica existentes o bien, no ofrecen una respuesta temporal determinista, o bien, en caso de ser determinista, no es lo suficientemente rápida.
- El problema de la fragmentación, puede provocar el fallo de la aplicación a pesar de que se disponga de suficiente memoria libre.

La gran mayoría de sistemas operativos modernos proveen facilidades que permiten la

implementación de la región *heap*, que es la utilizada por las tareas para alojar su memoria dinámica. El sistema operativo suele ofrecer un soporte mínimo consistente en una llamada al sistema para aumentar el tamaño del área *heap* del proceso. Por ejemplo, POSIX define la llamada al sistema ***brk()***. Sin embargo, el uso directo de esta primitiva por parte de los procesos no suele ser práctico ya que trabaja a nivel de página física, lo que significa que sólo permite el manejo de páginas físicas completas en el *heap*. El uso de una página completa por parte del proceso para almacenar una estructura compuesta por tan sólo unos pocos bytes se traduce en una gran ineficiencia. En su lugar, los procesos utilizan los llamados gestores de memoria dinámica. Estos gestores, normalmente implementados como bibliotecas, son los encargados de reservar memoria para la región de *heap* y posteriormente gestionarla, permitiendo al proceso realizar peticiones de tamaño menor a una pagina física.
Desde el punto de vista del proceso, el problema de la gestión de memoria está resuelto, ya que ésta se delega a los gestores de memoria dinámica. Sin embargo, los gestores de memoria se enfrentan a un problema de difícil solución, el problema de la fragmentación: una sucesión de peticiones de asignación y liberación de memoria provoca una progresiva partición del *heap* inicial en bloques de tamaño más pequeño. El gestor puede, después de un cierto tiempo de funcionamiento, no ser capaz de satisfacer una petición de asignación, a pesar incluso de que la suma de los tamaños de los bloques libres disponibles sea mayor que el bloque de tamaño requerido.
Hoy en día, existen dos formas de gestionar la memoria dinámica:

- Memoria dinámica explícita o manual: el programador tiene el control directo sobre cuando se asigna y se libera la memoria. Normalmente esto se realiza mediante llamadas explícitas a las funciones que ofrece el gestor de memoria (por ejemplo, malloc/free en el lenguaje C). Y, más concretamente, en sistemas que cumplan con el estándar POSIX de tiempo real se tiene la posibilidad de reservar en memoria el espacio necesario para alojar todas las páginas que la tarea necesite con llamadas como ***mlockall()***, o ***mlock()*** si son solo una pequeña fracción del total necesario.
- Memoria dinámica implícita o automática: el programa solicita memoria conforme la necesita, pero no indica explícitamente que bloques de memoria han quedado libres.

Desde el punto de vista algorítmico, la gestión de memoria dinámica es un problema de optimización, que consiste en minimizar la cantidad de memoria necesaria para satisfacer una serie de peticiones de asignación y liberación, pero esta secuencia de peticiones no es conocida a priori. Por tanto, la complejidad manifiesta de encontrar un algoritmo óptimo para la resolución del problema de la gestión de memoria dinámica ha provocado la necesidad de diseñar una gran cantidad de gestores, como por ejemplo, Best-Fit, First-Fit, Next-Fit, Worst-Fit, Binary Buddy, etc.

4.2.2 Gestores

A continuación se presentan los gestores de memoria dinámica más ampliamente usados en la actualidad.

4.2.2.1 First-fit

Este algoritmo implementa una política de primer ajuste, normalmente mediante el uso de una lista doblemente enlazada, donde se almacenan para su uso posterior los bloques libres. El inicio de la búsqueda del primer bloque adecuado para satisfacer una petición determinada varía según la implementación, a saber, ordenación por direcciones físicas, orden LIFO u orden FIFO.

4.2.2.2 Best-fit

El algoritmo Best-Fit consiste en una implementación de la política de mejor ajuste, utilizando una lista doblemente enlazada, al igual que en el algoritmo First-Fit. Sin embargo, esta estructura de datos no es la única opción existente, otros algoritmos como el AVL, explicado más adelante, utilizan estructuras avanzadas (árboles balanceados) para implementar la misma política con una respuesta temporal mejor.

Para implementar la política de mejor ajuste, este algoritmo realiza una búsqueda exhaustiva en la estructura de datos, seleccionando el bloque de tamaño más parecido a la petición recibida. En caso de existir varios bloques libres de dicho tamaño, la elección del bloque depende de la implementación del mecanismo(orden físico, FIFO o LIFO).

De este algoritmo cabe reseñar que el tiempo de cómputo es elevado debido, lógicamente, a la búsqueda exhaustiva realizada, pero, sin embargo, tiene un menor grado de fragmentación que los anteriores algoritmos.

4.2.2.3 Next-fit

Variante de First-Fit que implementa una política de ajuste siguiente, donde la búsqueda del bloque adecuado para la asignación no comienza siempre en la cabeza de la lista sino en la posición donde terminó la ultima búsqueda. Por lo demás este algoritmo tiene el mismo comportamiento que el algoritmo First-Fit.

4.2.2.4 Worst-fit

El algoritmo Worst-Fit es una implementación de la política de peor ajuste mediante una lista doblemente enlazada. Este algoritmo es muy parecido a Best-Fit, con la única diferencia de que asigna el bloque más grande encontrado que satisfaga la petición de memoria. Para ello este algoritmo realiza una búsqueda exhaustiva en la estructura de datos.

Destacar que, además del elevado tiempo de cómputo consumido, el grado de fragmentación presentado por este algoritmo es muy elevado.

4.2.2.5 Binary buddy

El gestor de memoria Binary Buddy es una de las múltiples variantes pertenecientes a la categoría de sistemas de colegas.

Este algoritmo utiliza un vector de listas de bloques libres, donde cada lista contiene solo bloques libres de un tamaño determinado (es decir, si una lista representa el tamaño 64 bytes, todos los bloques contenidos en esta lista tendrán un tamaño de 64 bytes). La única diferencia de este algoritmo con el resto de su categoría es la función que utiliza para insertar los bloques libres en las diferentes listas. En el caso de este algoritmo se utiliza la función de potencias de dos, es decir:

$$\mathbf{I = \log_2(T)}$$

donde **T** representa el tamaño del bloque que se quiere insertar en la estructura de bloques libres o buscar en el interior de dicha estructura, e **I** corresponde a la posición en el vector.

4.2.2.6 Half-fit

Este algoritmo es considerado el primer gestor de memoria dinámica diseñado explícitamente para ser utilizado en sistemas de tiempo real. Además, es el primer

algoritmo conocido que utiliza mapas de bits para la gestión de bloques libres, hecho que permite acelerar la realización de búsquedas.
La estructura de datos de este gestor consiste en un vector de listas; cada lista contiene bloques libres de tamaños comprendidos entre la potencia de dos del índice que ocupa dicha lista en el vector y la potencia de dos del índice de la siguiente lista, de forma parecida al algoritmo Binary Buddy. A diferencia de este último, Half-Fit siempre fusiona bloques libres físicamente adyacentes. Para un acceso rápido a las listas contenidas en el vector, el algoritmo Half-Fit utiliza dos funciones diferentes para traducir el tamaño de un bloque a un índice en su estructura de bloques libres. La primera de estas funciones se usa para insertar bloques libres en la estructura y es:

$$\mathbf{l = \log_2(T)}$$

Donde **T** indica el tamaño del bloque a insertar e **l** indica el índice correspondiente a la lista.
La segunda función es utilizada para encontrar una lista cuyo contenido siempre satisfaga la petición de memoria recibida (esta función no tiene en cuenta si la lista se encuentra vacía).

$$\mathbf{l} = \begin{cases} \mathbf{0 \text{ si } T = 1} \\ \mathbf{\log_2(T-1)+1 \text{ en cualquier otro caso}} \end{cases}$$

4.2.2.7 Árboles balanceados AVL

Los árboles balanceados AVL son una variante de los árboles binarios de búsqueda.
Un árbol AVL es un árbol binario de búsqueda al que se le añade una condición de equilibrio. Esta condición consiste en que la diferencia de altura de los subárboles izquierdo y derecho de cualquier nodo tiene que ser a lo sumo 1. Básicamente un árbol AVL permite realizar inserciones, extracciones y búsquedas. En esta última acción, al tratarse de un tipo de árbol binario, se cumple que todo subárbol a la izquierda de un nodo almcena valores menores al valor del nodo y todo subárbol a la derecha almacena valores mayores. Así pues cualquier búsqueda realizada en un árbol AVL muestra una respuesta temporal asintótica acotada por **$O(\log_2(n))$.**

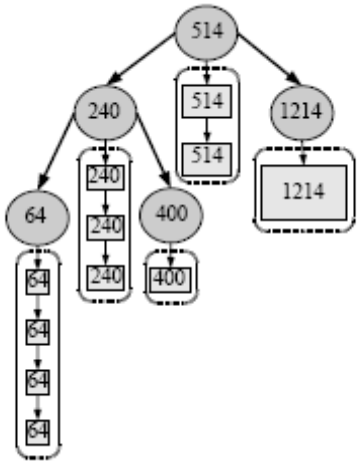


Figura 18: Árbol AVL.

Tal como se ve en la Figura 18, el árbol AVL es utilizado por este algoritmo para contener bloques de diferentes tamaños, cada hoja del árbol representa un tamaño dado y contiene bloques únicamente de dicho tamaño.

4.2.2.8 Dlmalloc

Actualmente este gestor de memoria es utilizado por las bibliotecas C y C++ de GNU.
El algoritmo dlmalloc utiliza una estructura híbrida para almacenar los bloques de memoria libres. Por una parte utiliza un vector, en el cual indexa los bloques libres de tamaño menor a una constante prefijada. Por otra parte utiliza una lista doblemente enlazada para bloques de tamaño mayor. Las primeras entradas del vector de bloques pequeños únicamente contienen bloques de un tamaño fijo, permitiendo una asignación inmediata de dichos bloques. El resto de entradas contiene rangos de tamaños, ordenados por tamaño dentro de cada lista. En la terminología del algoritmo, a los bloques indexados en la estructura se les conoce como “Bins”. La estructura indexa de manera exacta los bloques libres menores de 512 bytes mientras que los bloques de hasta 128 Kb son agrupados en rangos. El algoritmo dlmalloc utiliza diferentes estrategias de asignación según el tamaño demandado, en concreto:

- Para peticiones de tamaños pequeños (<= 64 bytes), el algoritmo se comporta como asignador cache, utilizando una política FIFO de asignación.

- Para peticiones de tamaños intermedios (> 64 bytes y < 512 bytes), el algoritmo es configurable para tener un comportamiento FIFO o bien Best-Fit.

- Para peticiones de tamaños grandes (>= 512 bytes), se utiliza una estrategia puramente Best-Fit.

- Para peticiones de tamaños muy grandes (>= 128 Kbytes), el algoritmo confía en las funciones de gestión de memoria del sistema operativo subyacente, por ejemplo la función **brk()** en el caso de los sistemas UNIX.

4.2.3 Solución

Tras la exposición del problema de la gestión de memoria dinámica en sistemas de tiempo real y haber evaluado los gestores de memoria más representativos con los que se cuenta en la actualidad, se está en disposición de abordar los puntos más relevantes del diseño del gestor de memoria dinámico Two-Level Segregated Fit (TLSF), desarrollado por la Universidad Politécnica de Valencia.
Las dos características básicas de diseño de TLSF son:

- Tiempo de respuesta rápido y totalmente predecible.
- Baja fragmentación.

Para lograr estos objetivos, TLSF sigue los criterios detallados a continuación:

- **Fusión inmediata**: cuando se libera un bloque de memoria el gestor lo une inmediatamente al bloque contiguo.
- **Umbral de división**: el tamaño de bloque de memoria más pequeño asignable es de 32 bytes. Esto es debido a que la memoria requerida habitualmente por la gran mayoría de aplicaciones se utiliza para almacenar estructuras de datos complejas y

no datos simples.

- **Estrategia Good-Fit**: el gestor siempre intenta asignar el bloque más pequeño lo suficientemente grande como para contener el bloque requerido. Esta estrategia es una evolución de Best-Fit que usa listas segregadas(vector de listas de bloques libres que cumplen una determinada condición, como por ejemplo tener un mismo tamaño o rango de tamaños), en contraposición a las doblemente enlazadas que usa el gestor ya analizado. Este hecho proporciona un comportamiento temporal predecible y una muy baja fragmentación.
- **No se realiza reubicación**: se asume que la memoria inicial disponible en la región de *heap* es un único bloque de memoria libre y, por tanto, no se cuenta con llamadas clásicas de sistema como **brk()**. TLSF ha sido diseñado para ofrecer un completo soporte para la gestión de la memoria dinámica, sin necesidad de usar ningún hardware de soporte de memoria virtual, MMU por ejemplo.
- **Estrategia idéntica para cualquier tamaño de bloque**: el fin perseguido es evitar tiempos de ejecución indeterminados, como puede llegar a ocurrir con gestores como *dlmalloc*.
- **Memoria no inicializada**: ni el espacio de memoria inicial ni los bloques liberados son inicializados a 0, como suele hacerse en otros gestores. Se parte de la base que el entorno de uso de TLSF es lo suficientemente confiable como para no invertir tiempo en la inicialización y así garantizar tiempos de respuesta rápidos.

Bajo la tutela y cumplimiento de los criterios enumerados, el gestor TLSF proporciona un tiempo de respuesta rápido y constante O(1), y una baja fragmentación, por tanto, es un claro candidato para formar parte de un sistema operativo de tiempo real estricto como puede ser el que constituye el núcleo Linux con las mejoras del parche RT-preempt.

4.3 SOLUCIONES ALTERNATIVAS

En la actualidad existen diferentes aproximaciones al tiempo real en el núcleo Linux. Una de las soluciones posibles es la evaluada en el punto 3.3(ver Figura 19), pero, lógicamente, no es la única y con esta comparativa se pretende dar una visión global de las otras arquitecturas disponibles para conseguir un tiempo real estricto.

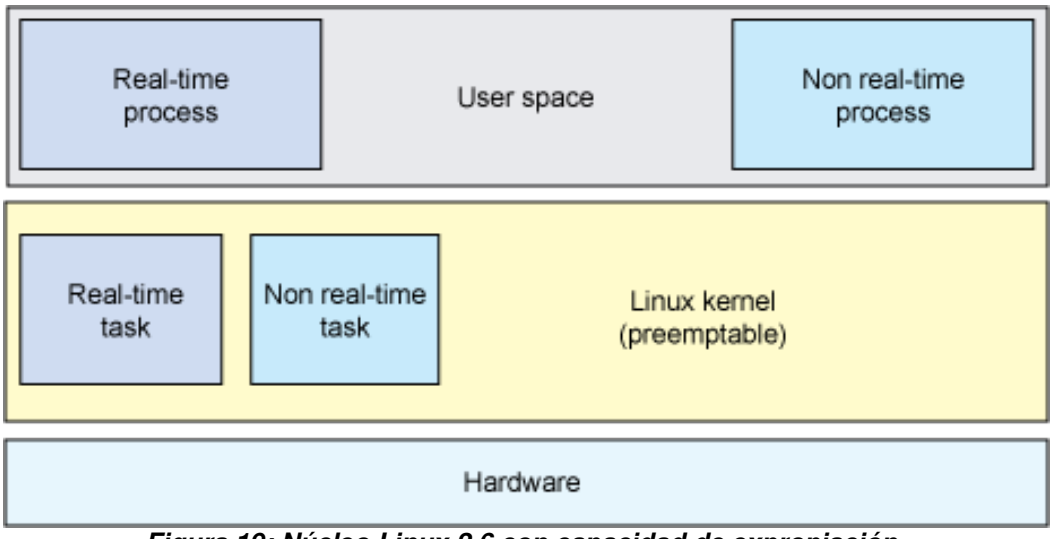


Figura 19: Núcleo Linux 2.6 con capacidad de expropiación.

Las soluciones que aportan capacidades de tiempo real al núcleo Linux son:

1. Aproximación del thin-kernel o micro-kernel.
2. Aproximación del nano-kernel.
3. Aproximación del resource-kernel.

4.3.1 Micro-kernel

Esta solución, tal y como indica su nombre, usa un segundo núcleo como interfície de abstracción entre el núcleo Linux y el hardware de la máquina(ver Figura 20).

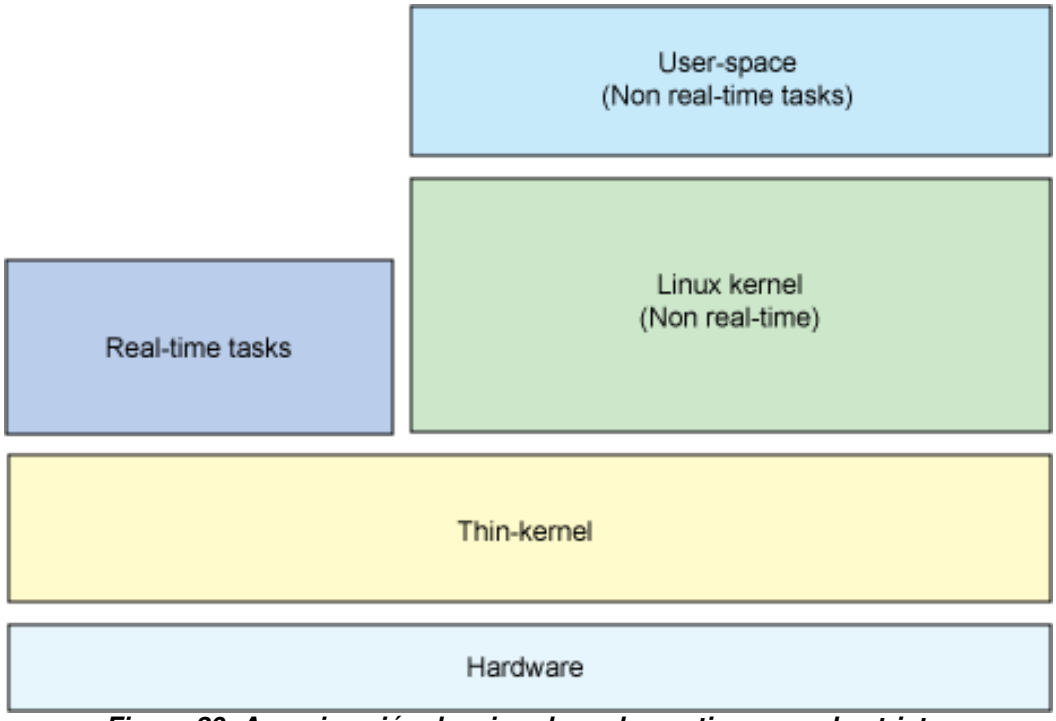


Figura 20: Aproximación de micro-kernel para tiempo real estricto.

La parte del núcleo que no es considerada de tiempo real se ejecuta en segundo plano como una tarea menos prioritaria del micro-kernel y alberga a todas aquellas tareas de tiempo compartido. En contraposición, todas aquellas tareas que son de tiempo real se ejecutan directamente en el micro-kernel.

Uno de los usos más relevantes del micro-kernel, amén de albergar a las tareas de tiempo real, es la gestión de interrupciones. Mediante esta gestión el micro-kernel garantiza que su propia ejecución sea prioritaria ante cualquier otra tarea del sistema, de forma que posibilita un comportamiento de tiempo real estricto.

Aunque esta aproximación tenga notables ventajas como la de dar soporte a tareas de tiempo real conjuntamente con un núcleo Linux estándar, también tiene ciertos inconvenientes. Uno de los más sobresalientes se encuentra, precisamente, en la independencia que existe entre tareas de tiempo real y de tiempo compartido. Esta separación añade una especial complejidad a nivel de depuración durante el proceso de desarrollo. Se debe tener en cuenta que, usando dos subsistemas distintos de planificación y activación de tareas, la realización de un profiling o de un test de performance de una tarea se complica considerablemente. Además, las tareas de tiempo compartido no disponen de un soporte completo de la plataforma Linux debido a la propia existencia del micro-kernel.

Ejemplos de esta aproximación serían soluciones como RTLinux, actualmente

perteneciente a Wind River Systems, RTAI (Real-Time Application Interface) y Xenomai.

4.3.2 Nano-kernel

Mientras que la aproximación del micro-kernel, analizada anteriomente, contaba con un pequeño núcleo capaz de gestionar tareas de tiempo real, la solución que aporta el nano-kernel va más allá y minimiza aún más el espacio usado para el núcleo. En este sentido, se podría entender la aproximación actual más como una interfície de abstracción hardware (HAL) que como un núcleo en si mismo.
El nano-kernel provee un sistema de compartición de recursos hardware entre diferentes sistemas operativos que se ejecutan en una capa superior (ver Figura 21).

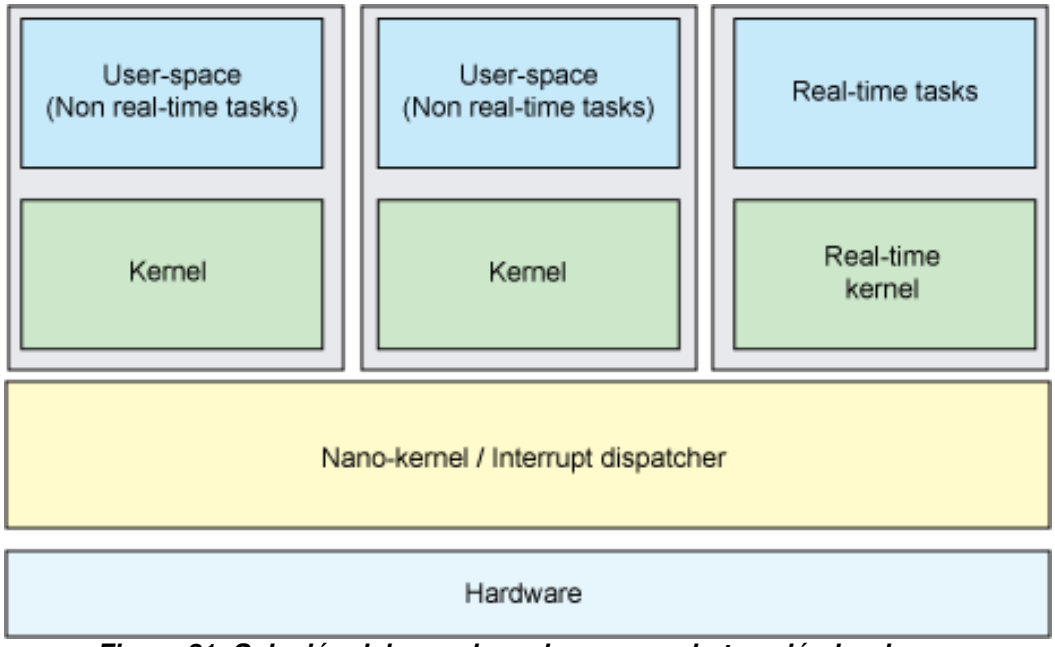


Figura 21: Solución del nano-kernel para una abstracción hardware.

Como el nano-kernel es una abstracción hardware puede suministrar una sistema de prioridades a los diferentes sistemas operativos que estén ejecutándose en capas superiores; este es, precisamente, el motivo del soporte para comportamientos de tiempo real estricto.
Esta aproximación es muy parecida a las soluciones que hoy en día se pueden encontrar en el mercado referentes a la virtualización de sistemas operativos. Sistemas como VMware ESX, Xen, etc., insertan una fina capa software justo por encima del hardware del ordenador, posibilitando la coexistencia de diferentes sistemas operativos de forma concurrente. Mediante un gestor de recursos hardware llamado “hypervisor” cada sistema operativo puede disponer de los recursos hardware necesarios de forma transparente y sin percibir la existencia de los otros sistemas en la máquina física real.
Como se puede observar la solución aportada por el nano-kernel es prácticamente la misma que la usada en los sistemas de virtualización.
El mayor referente de la aproximación del nano-kernel es el sistema operativo llamado ADEOS(Adaptative Domain Environment for Operating Systems). ADEOS soporta la ejecución concurrente de diferentes sistemas operativos. ADEOS posibilita, además, el control determinístico del flujo de interrupciones hardware usando una capa software antes de que el núcleo o núcleos Linux las procesen. En este control está implícita la interceptación, el enmascarado y la priorización de las propias interrupciones.

4.3.3 Resource-kernel

Otra de las arquitecturas que permite disponer de tiempo real estricto en el núcleo Linux es la conocida como aproximación resource-kernel. Esta solución se articula mediante la inserción de un módulo en el núcleo que permite efectuar reservas en el uso de diferentes recursos hardware. Los recursos disponen, a través de esta aproximación, de diferentes parámetros de reserva, entre los que sobresalen el tiempo de proceso requerido por una tarea y el tiempo límite de respuesta.

Esta arquitectura dispone de una API (Application program interface) que posibilita la gestión de los parámetros de reserva en las tareas desarrolladas.

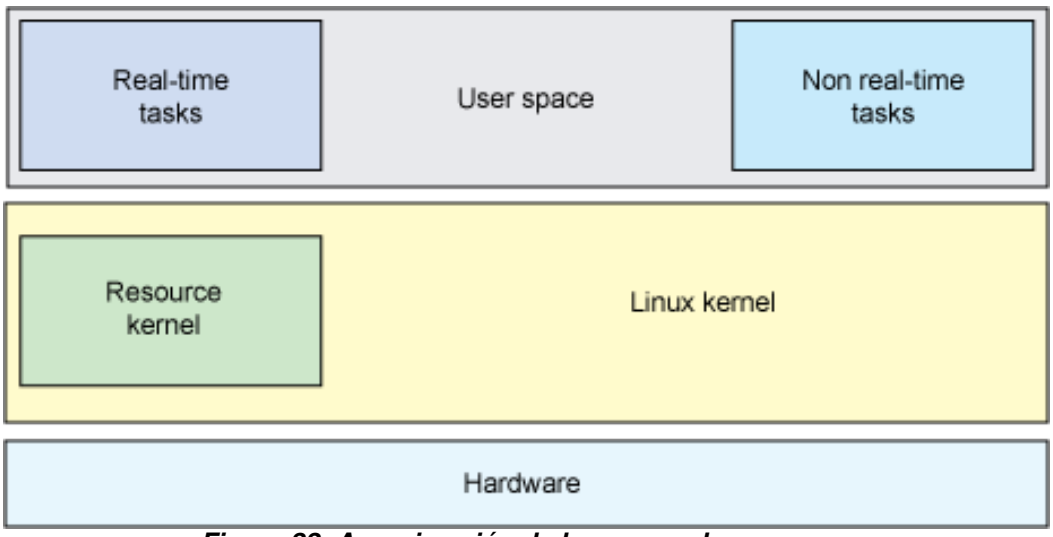


Figura 22: Aproximación de la reserva de recursos.

En la figura anterior se puede la solución aportada por esta aproximación.

El núcleo de recursos puede combinar las diferentes peticiones con el fin de definir una planificación que proporcione un acceso válido a los recursos teniendo en cuenta las restricciones definidas en cada una de las tareas, o bien devolver un error si no puede ser garantizado. Usando, por ejemplo, un algoritmo de planificación como EDF (Earliest-Deadline-First), el núcleo puede llegar a gestionar la carga dinámica de tareas en el sistema garantizando, lógicamente, el determinismo en los tiempos de respuesta ante cualquier evento.

Uno de los ejemplos de esta aproximación es la implementación hecha por la Universidad Carnegie Mellon a finales de los 90 llamada Linux/RK. Esta solución fue desarrollada hasta obtener un sistema operativo llamado TimeSys Linux/RT, que garantiza una reserva absoluta para una tarea o conjuntos de tareas del procesador, soporte para la herencia de prioridad, un núcleo absolutamente expropiativo y, todo lo esencial para posibilitar el desarrollo de aplicaciones que sigan un comportamiento de tiempo real estricto.

CAPÍTULO 5
CONCLUSIONES Y TRABAJO FUTURO

5.1 CONCLUSIÓN

Tras todo el desarrollo y análisis llevado a cabo en el presente trabajo se puede concluir que el núcleo Linux es un buen candidato para ser usado en infraestructuras tecnológicas donde la relevancia del cumplimiento de los tiempos de respuesta es vital para su correcto funcionamiento. Es de recibo decir, por otra parte, que la modificación del núcleo a través de la aplicación del parche RT-preempt en un sistema operativo de tiempo real estricto aún no se ha conseguido en su totalidad. Como se ha visto, aún existen ciertas partes del núcleo(acceso a estructuras de datos internas del núcleo y rutinas de servicio de interrupción que deben ser ejecutadas en contexto de interrupción) que no permiten tener un código completamente expropiativo. A pesar de este handicap, los resultados obtenidos en las pruebas realizadas, teniendo en cuenta que se han llevado a cabo sobre una arquitectura i386, que no es la más idónea para garantizar cumplimientos de intervalos de tiempo estrictos, verifican que el comportamiento determinista del núcleo es lo suficientemente sólido como para usarlo en plataformas de tiempo real.
Soluciones propietarias que usan el núcleo Linux como eje vertebrador del sistema, como VxWorks, por ejemplo, han demostrado sobradamente que con un breve pero intenso trabajo de mejora se pueden usar estos núcleos para misiones de tiempo real crítico, como ha sucedido en numerosas ocasiones en misiones espaciales tanto de la NASA como de la ESA.
En resumen, el núcleo Linux modificado con el parche RT-preempt y con una posible sustitución de su gestor de memoria dinámica por el presentado en el capítulo 4, podría constituir una muy buena aproximación a una solución de tiempo real estricto.

En relación a los objetivos presentados en el primer capítulo, cabe decir que se han conseguido en su totalidad. Se ha analizado, investigado y comprendido el código fuente del núcleo Linux, tanto en su forma estándar como en su forma modificada tras la aplicación del parche.

De los objetivos colaterales, es decir, del estudio del comportamiento de un sistema operativo de tiempo real estricto, del manejo de las utilidades más idóneas de testing, del uso de las librerías POSIX de tiempo real y de la familiarización con la compilación de un núcleo Linux modificado, hay que reseñar que todos se han alcanzado tras un intenso trabajo de investigación y estudio de manuales y lecturas de artículos relacionados.

5.2 TRABAJO FUTURO

Tomando como base el trabajo realizado se pueden llevar a cabo innumerables proyectos de los que se destacarán tres, que son de especial interés para el que suscribe.
En primer lugar se podría sustituir el gestor de memoria que implementa el núcleo estándar de Linux por el gestor desarrollado en la UPV, es decir, por el Two-Level Segregated Fit. Tras la sustitución y posterior compilación del núcleo se podrían realizar pruebas que constataran una mejoría en el determinismo del sistema.
En segundo lugar, y habiendo realizado el desarrollo anterior, se podría empotrar el sistema en una placa que siga el estándar PC104. Estas placas son de tamaño reducido y apilables, y están especialmente diseñadas para albergar sistemas dedicados, como adquisición de datos, control de sistemas robóticos, etc. La arquitectura que usa este tipo de estándar suele ser i386, por tanto, no se presentarían nuevos problemas. Sería interesante llevar a cabo este trabajo porque, habitualmente, estas plataformas llevan como discos duros memorias flash o, actualmente, discos de estado sólido, además, cada

placa aporta una funcionalidad a la infraestructura y, por tanto, se podrían controlar de forma exhaustiva los servicios que el núcleo compilado debería facilitar.
En último lugar, un ejercicio muy interesante sería empotrar el sistema anterior en una placa con microprocesador RISC(Reduced Instruction Set Computer). Un procesador ARM sería un buen candidato, y más concretamente, una plataforma IMX de FreeScale sería el marco de trabajo perfecto. Estas soluciones de la marca FreeScale son ideales ya que en su diseño cuentan con un microprocesador ARM, un subsistema de memoria, un subsistema de entrada/salida, y todos los componentes necesarios para la gestión y control del global del sistema.
Por último apuntar que con el anterior ejercicio se deberían usar utilidades de compilación cruzada por tratarse de otra tecnología a la usada en este trabajo.

APÉNDICES

APÉNDICE A
MECANISMOS DE SINCRONIZACIÓN DEL NÚCLEO LINUX

El núcleo Linux comparte multitud de recursos que deben ser protegidos del acceso concurrente de los diversos threads que estén en ejecución. Si no existiera esta protección, lógicamente, los cambios realizados por cada uno de los threads en ejecución serían constantemente sobreescritos. Esta situación produciría resultados incorrectos y, muy posiblemente, permitiría el acceso a datos en un estado inconsistente. Una secuencia de código que accede y manipula datos compartidos recibe el nombre de *sección crítica*. Si dos threads en ejecución se encuentran de forma simultánea en una misma sección crítica se suele hablar de condiciones de carrera. La forma de evitar que se produzcan este tipo de problemas es sincronizando el acceso a la sección crítica, es decir, asegurar que cuando una tarea está ejecutando su sección crítica ninguna otra puede estar ejecutándola al unísono. Este tipo de sincronización debe cumplir unos requisitos básicos:

- **Exclusión mútua**: solo debe existir una única tarea ejecutando el código de la sección crítica.
- **Progreso**: una tarea que se encuentre fuera de la sección crítica nunca debe bloquear a otra tarea que quiera acceder.
- **Espera limitada**: una tarea que quiera acceder a la sección crítica no debe esperar indefinidamente.

En un sistema GNU/Linux uniprocesador existen multitud de fuentes de concurrencia y, por lo tanto, posibles condiciones de carrera:

- **Interrupciones**: una interrupción puede darse de forma asíncrona, en cualquier instante, y puede ocurrir en cualquier parte del código en ejecución.
- **Softirqs** y **tasklets**: durante una interrupción el manejador debe realizar, a menudo, un gran número de operaciones, pero, a su vez, uno de los objetivos perseguidos es que el código interrumpido vuelva a ejecutarse lo más pronto posible. La consecución de estas dos metas que están en contraposición es complicada y el núcleo Linux lo resuelve dividiendo el trabajo a procesar durante una interrupción en dos partes, *top-half* y *bottom-half*. *Top-half* se encarga de ejecutar el trabajo crítico y específico del hardware inmediatemente después de la interrupción. En cambio, *bottom-half* se encarga de aquel trabajo que puede ser aplazado sin afectar a las operaciones del núcleo. Pues bien, tanto las **softirqs** como los **tasklets** son los mecanismos encargados de llevar a cabo el trabajo más crítico de la parte *bottom-half* de una interrupción.
  El núcleo puede lanzar o planificar **softirqs** o **tasklets** en cualquier instante y se puede dar en cualquier parte del código que se esté ejecutando.
- **Expropiación del núcleo**: ya que el núcleo es expropiativo.
- **Durmiendo o con una sincronización con el modo usuario**: una tarea en el núcleo puede echarse a dormir e invocar al planificador.

Con el fin de evitar todos los problemas derivados de las diferentes situaciones de concurrencia en las que pueden incurrir las tareas que se ejecuten en el sistema, el núcleo aporta diferentes mecanismos de sincronización, entre los que destacan los *spin locks* y los *semáforos*. Estos son los dos mecanismos que se describirán a continuación, ya que su funcionamiento e implementación están muy relacionados con la obtención de un núcleo completamente expropiativo.

a. Spin lock

Los spin locks son variables compartidas que funcionan como flags. Cuando un thread entra en una sección crítica intenta hacerse con el lock, si está libre se apodera de éste y continua con su ejecución. Si existe otro thread que está intentando acceder a la misma sección crítica, al encontrase con un lock reservado por otro thread, inicia un bucle cíclico del que solo saldrá cuando el primer thread libere el lock. Este tipo de espera se conoce como espera activa, y suele asociarse a cierta ineficiencia al consumir CPU por solo esperar. De la conducta de esperar en un bucle cíclico es de donde le viene el nombre a este mecanismo de sincronización (spinning). Cuando el thread que ha adquirido el lock abandona la sección crítica libera el spin lock; esto posibilita que el thread que había iniciado una espera activa se haga con el lock y pueda acceder al código compartido. Este mecanismo provee una protección necesaria para la concurrencia en arquitecturas que permiten multiproceso.

En sistemas monoprocesador que siguen un algoritmo de prioridades expulsivas, esto es, con capacidad de expropiación de la CPU, los spin locks simplemente activan y desactivan la expropiación que se realiza en modo núcleo cuando son liberados o adquiridos, respectivamente. En consecuencia, el tiempo invertido por parte de un thread en hacerse con un spin lock es equivalente al tiempo de latencia de un sistema.

b. Semáforo

Los semáforos en el núcleo Linux son las estructuras que permiten la espera pasiva. Cuando un thread intenta hacerse con un recurso que está sincronizado mediante un semáforo y éste ya está en poder de otro thread, el primero es suspendido. Este thread volverá a estar en ejecución cuando el semáforo que protegía el uso del recurso compartido haya sido liberado. Como el thread en ejecución ha sido dormido en la pugna por el semáforo, se observa que el lock de este semáforo solo podrá ser obtenido en contextos de proceso, por tanto, manejadores de interrupciones y funciones aplazadas que se ejecutan durante una interrupción nunca podrán usar un semáforo. *A diferencia de los spin locks, los semáforos nunca desactivan la expropiación en modo núcleo*, en consecuencia, el tiempo invertido por la consecución del lock de un semáforo nunca afectará al tiempo de latencia del sistema.

Los semáforos permiten la creación de un número arbitrario de locks de forma simultánea. En el momento de la declaración, la variable ***count*** contendrá la cantidad de locks del semáforo. Habitualmente, esta variable se establece en 0 o 1, valores que representan la disponibilidad del recurso compartido o bien la ocupación del mismo, respectivamente. En estos casos, estos mecanismos de sincronización son llamados semáforos binarios o mutex. Pero, como se ha comentado, también se puede disponer de un semáforo con un número mayor de locks, supuesto en el que se permiten la entrada concurrente a diferentes threads en la sección crítica. En estos casos no prima tanto la exclusión mútua como la necesidad de respetar los límites declarados en el código para los diferentes hilos de ejecución.

APÉNDICE B
INVERSIÓN DE PRIORIDAD

La inversión de prioridad ocurre cuando una tarea de mayor prioridad es obligada a esperar a otra de menor prioridad. Un ejemplo simple de la inversión de prioridad se presenta en el momento en el que una tarea de menor prioridad se hace con el lock de un recurso compartido y una de mayor prioridad intenta hacerse con el mismo lock. La tarea de mayor prioridad pasa a un estado de bloqueo hasta que la tarea de menor prioridad acabe y libere el lock del recurso. En este instante, la tarea más prioritaria puede ser replanificada y apoderarse del lock.
Otro concepto relacionado es la inversión de prioridad no limitada. Esta se da cuando una tarea prioritaria debe esperar una cantidad de tiempo ilimitada a que otra de menor prioridad libere el lock del recurso. Se desprende de esta explicación que la duración de la inversión de prioridad dependerá, no solo del tiempo necesario para gestionar el manejo del recurso compartido, sino de las acciones no predecibles de la tarea poseedora del lock.
La siguiente figura muestra la secuencia de acciones de la inversión de prioridad no limitada:

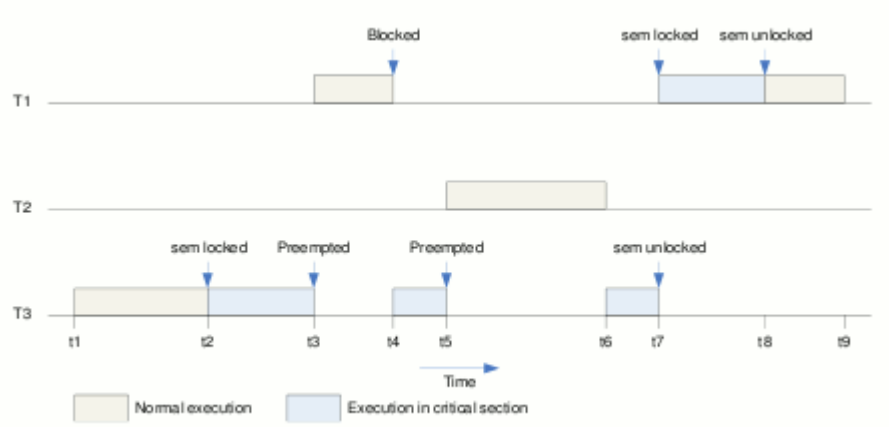


Figura 23: Inversión de prioridad no limitada.

- **t1:** T3 inicia su ejecución.
- **t2:** T3 entra en la sección crítica y se apodera del semáforo ***sem***.
- **t3:** T1 expulsa a T3 e inicia su ejecución.
- **t4:** T1 intenta entrar en la sección crítica pero se encuentra con el bloqueo ejercido por T3, en poder del semáforo ***sem***. T3 reanuda su ejecución.
- **t5:** T2 inicia su ejecución y expulsa a T3.
- **t6:** T2 suspende su ejecución por motivos no relevantes y T3 reanuda su ejecución.
- **t7:** T3 abandona la sección crítica liberando el semáforo. T1 expulsa de nuevo a T3, se hace con el lock del semáforo ***sem*** y entra en la sección crítica.
- **t8:** T1 abandona la sección crítica y libera el lock del semáforo.

Cabe destacar que en la actual línea principal de desarrollo del núcleo Linux, la expropiación está deshabilitada cuando las tareas protegen los recursos compartidos con spin locks. Esto previene la aparición de la inversión de prioridad. En consecuencia, T2 no puede expropiar el uso de CPU a T3 en t5 tal y como ocurre en la Figura 1, y por tanto, la inversión de prioridad no limitada es evitada. Pero, por esta misma razón T1 no podría expulsar a T3 en t3 y este esquema es más que inadecuado para el buen funcionamiento de ciertas aplicaciones de tiempo real debido, sobre todo, al impacto en los tiempos de

latencia que propiciarían.
A la práctica existen dos aproximaciones para la prevención de la inversión de prioridad permitiendo la expropiación a la vez: protocolo de techo de prioridad y protocolo de herencia de prioridad.

- **Protocolo de techo de prioridad.**
  Cada tarea tiene una prioridad estática por defecto .
  Cada recurso tiene un valor de techo estático que es la prioridad estática máxima de las tareas que usan el recurso.
  Cuando una tarea accede a un recurso adquiere una prioridad dinámica que es el máximo de su prioridad estática y el techo de prioridad del recurso.

- **Protocolo de herencia de prioridad.**
  La idea básica de la herencia de prioridad es que las tareas de menor prioridad que en un cierto momento estén en poder del lock de un recurso, hereden la prioridad de alguna otra tarea de mayor prioridad que esté pendiente de la liberación del recurso. El cambio de prioridad o la herencia efectiva por parte de la tarea menos prioritaria se inicia en el instante en el que la tarea más prioritaria se intenta hacer con el lock del recurso, y finaliza en el momento en el que la tarea menos prioritaria libera el lock. La siguiente figura muestra como la inversión de prioridad no limitada es solucionada por la aplicación del protocolo de herencia de prioridad.

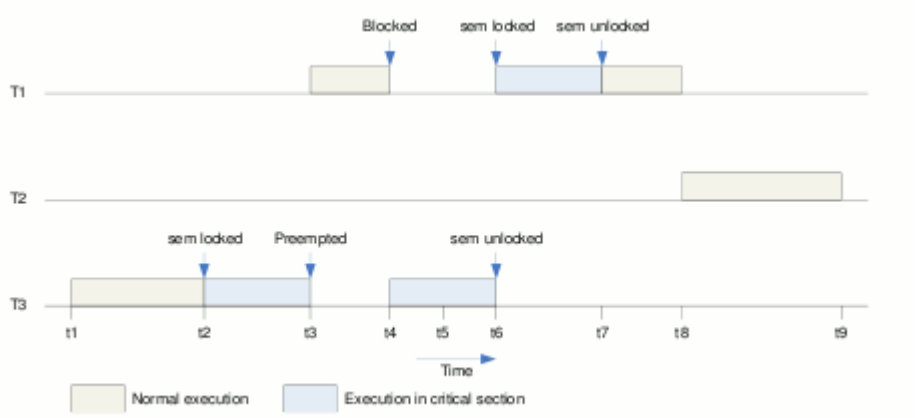


Figura 24: Herencia de prioridad.

La secuencia de acciones ahora es la siguiente:

- **t1:** T3 inicia su ejecución.
- **t2:** T3 entra en la sección crítica y se apodera del semáforo ***sem***.
- **t3:** T1 expulsa a T3 e inicia su ejecución.
- **t4:** T1 intenta entrar en la sección crítica pero se encuentra con el bloqueo ejercido por T3, en poder del semáforo ***sem***. A T3 se le asigna temporalmente la misma prioridad que a T1. T3 reanuda su ejecución.
- **t5:** T2 está listo para entrar en ejecución pero como ahora T3 tiene la misma prioridad que T1 y mayor que T2, este último no puede expropiar la CPU.
- **t6:** T3 abandona la sección crítica liberando el semáforo. En este momento su nivel de prioridad es bajado a su valor inicial. T1 expulsa a T3 y accede a la sección crítica haciéndose con el semáforo.
- **t7:** T1 abandona la sección crítica y libera el lock del semáforo.
- **t8:** T1 finaliza su ejecución y T2 incia la suya.

APÉNDICE C
APROXIMACIÓN HISTÓRICA A LOS TEMPORIZADORES DE ALTA RESOLUCIÓN

La primera solución que se implementó de un temporizador de alta resolución en GNU/Linux fue el componente UTIME perteneciente al proyecto de tiempo real de la Universidad de Kansas (KURT,1998). Esta extensión proveía bajo demanda al núcleo Linux de una resolución de microsegundos y de capacidad para un comportamiento de tiempo real. Esta aproximación se empezó a usar en núcleos Linux a partir de la versión 2.0 y hasta la versión 2.4.
La solución aportada por la Universidad de Kansas se basaba en la observación que, aunque las tareas de tiempo real se planificaban con un tiempo límite del orden de microsegundos, los eventos a planificar raramente ocurrían cada microsegundo. De este modo, UTIME implementaba un mecanismo por el que se permitía que las interrupciones de reloj ocurrieran en un microsegundo dado, pero no cada microsegundo.
Con el fin de conseguir una precisión de microsegundo, se tuvo que añadir un nuevo campo a la estructura de datos del temporizador llamado *fractional expiration*. Esta estructura también contenía el valor del tiempo límite del temporizador especificado en el campo *expiration*. Por tanto, el campo *fractional expiration* especificaba el microsegundo a partir del valor descrito en el campo *expiration* el temporizador finalizaría. En otras palabras, este campo permitía al usuario especificar cuantos microsegundos tras el valor de expiración el temporizador debía expirar. El campo *fractional expiration* con el paso del tiempo se renombraría y se acabaría llamando *usec* (microsegundo). Este mecanismo con el que obtener una resolución de microsegundos en el núcleo Linux se diseñó y se implementó en el gestor de tiempo del propio núcleo, siempre preservando la compatibilidad con el resto de funcionalidades clásicas existentes. El usuario podía aprovecharse de la resolución aportada por el componente UTIME a través de las siguientes funciones y métodos:

- Llamada al sistema ***nanosleep*** que permitía dormir a un proceso con precisión de microsegundos.
- Temporizador de intervalo que permitía establecer un temporizador **ITIMER_REAL** con precisión de microsegundos.
- Llamada al sistema ***select*** con granularidad de microsegundos.
- Llamada al sistema ***poll*** a nivel de microsegundos.
- Implementando un módulo personalizado para el sistema.

La siguiente solución para tener una mayor resolución en el núcleo Linux, un tanto más actual, fue la aproximación realizada en el proyecto de Temporizadores de Alta Resolución (HRT), introduciendo en escena temporizadores de resolución muy fina que soportaban el estándar POSIX.
El diseño de HRT fue llevado a cabo a partir del trabajo realizado en UTIME, de ahí que la parte de alta resolución del temporizador tuviera su propio campo en la misma estructura de datos. Además, los temporizadores de alta resolución eran almacenados en el gestor de tiempo hasta que se alcanzaba el valor de su campo de expiración. En el momento de su expiración, éstos eran movidos a una lista separada de temporizadores. Una interrupción de reloj independiente era la encargada de activar los temporizadores que componían esta lista. En resumen, HRT introducía una nueva interrupción para mantener una resolución con una granularidad de microsegundo.
La implementación de temporizadores de alta resolución en UTIME y HRT estaba fuertemente ligada a la gestión de tiempo genérica del núcleo Linux. Diversos estudios

(Gleixner et al. 2006) apuntaron que ambos proyectos se construían modificando el gestor de tiempo del sistema, propiciando un desafortunado tiempo extra en las acciones a realizar y una variación notable de los tiempos de latencia. En conclusión, los diferentes autores de los estudios acordaron llevar a cabo un módulo serapado y autónomo que diera soporte a la alta resolución en núcleos Linux.

Thomas Gleixner junto a Ingo Molnar realizaron una primera aproximación de este módulo independiente que permitiera al núcleo Linux superar la clásica barrera de los 10ms de resolución; y esta primera solución recibió el nombre de ***Ktimers***. El proyecto ***Ktimers*** introducía un nuevo subsistema para los temporizadores de alta resolución que no interactuaba con la gestión del tiempo que se llevaba a cabo dentro del núcleo Linux. ***Ktimers*** estaba basado en tiempos de desconexión con resolución de nanosegundos. Al igual que HRT, ***Ktimers*** también usaba su propia interrupción para activar sus temporizadores. Pero existía un último handicap, a saber, el subsistema usaba la interrupción de reloj que el propio sistema tiene por defecto, en vez de usar una fuente de interrupciones independiente para aportar la alta resolución. Por tanto, los ***Ktimers*** tampoco proporcionaban una mejor resolución que las anteriores aproximaciones comentadas.

El siguiente paso dado para la consecución de un sistema válido de temporizadores de resolución fina fue la extensión de los ***Ktimers*** por parte del propio Thomas Gleixner. En esta ocasión se llamaron ***hrtimers***. Con la presentación de los ***hrtimers*** se aportó, además, un nuevo componente que recibió el nombre de "*evento de reloj*". Junto con la incorporación del nuevo desarrollo hecho para el núcleo Linux de la aproximación del *reloj del día* (Stultz 2005), el componente "*evento de reloj*" constituyó la base de la nueva implementación de los temporizadores de alta resolución ***hrtimers***. Con el fin de conseguir una mejor resolución que los anteriores temporizadores, en los ***hrtimers*** se implementó una nueva interrupción de reloj independiente y no limitada por los ticks de reloj del sistema. Con esto, finalmente, se pudo alcanzar la meta fijada en un inicio, es decir, tener un subsistema absolutamente autónomo con la capacidad de aportar una resolución lo suficientemente fina como para posibilitar la existencia de tiempos de respuesta del orden de microsegundos, sin necesidad de modificar el propio gestor de tiempo del núcleo Linux.

APÉNDICE D
RUTINAS DE TIEMPO REAL

Con el fin de conseguir una cierta agilidad en la programación de aplicaciones de tiempo real, se han confeccionado un par de programas que hacen uso de las más relevantes llamadas al sistema del estándar POSIX de tiempo real.
Antes de pasar a la explicación detallada de lo que cada una de las aplicaciones hace, cabe comentar que durante la elaboración del código que se presenta se han realizado innumerables pruebas de forma que se consiguiera el fin perseguido.
De entre todas las pruebas realizadas, finalmente, se decidió añadir a este trabajo los dos ejemplos que se comentarán, sobretodo, por su enorme valor, tanto conceptual como gráfico, de lo que un sistema operativo de tiempo real estricto puede aportar.
Del resto de pruebas se puede mencionar que se ha trabajo con threads concurrentes, recreando diversas situaciones en las que uno era más prioritario, o bien, un conjunto de threads con una misma prioridad de tiempo real se querían apoderar del lock de un recurso compartido, etc. Pero ninguno de estos códigos ejemplificaba tan bien como los aportados las capacidades del sistema estudiado. Aún así, el conocimiento extraído del uso de estas operaciones no ha sido poco.

- **test_sdout.c**
  En este programa se pretende mostrar como un sistema operativo de tiempo real estricto es capaz de gestionar la recepción de señales con una resolución de microsegundos y reaccionar en un mismo nivel de granularidad.
  Básicamente, en la aplicación, se establece un temporizador de alta resolución que genera una señal de alarma cada 100µs durante un intervalo de 1 segundo. El manejador que se define para gestionar la señal envía un carácter (un punto) a la salida estándar, de forma que, al acabar el intervalo, se tengan 10000 caracteres en la salida estándar.
  Pues bien, esto que parece algo muy simple y con poco interés a nivel de programación, se torna interesantísimo cuando se comprueba que solo es capaz de alcanzar la cifra de caracteres comentada un sistema operativo de tiempo real estricto.
  De las pruebas realizadas sobre los mismos núcleos usados en el apartado 3.5 se desprende que, solo uno de los tres sistemas es capaz de gestionar temporizadores de tan alta resolución aportando, además, una característica que los otros no tienen, el determinismo. Este sistema, lógicamente, es el núcleo 2.6.21.6 con la modificación para tiempo real estricto. Solo en este sistema el número de puntos obtenido por la salida estándar es el esperado, es decir, 10000; en los otros dos núcleos la cifra ronda los 9900, hecho que implica que se perderían de media unas 100 alarmas, situación impensable en sistemas donde el tiempo de respuesta es crítico.
  De la realización de este pequeño ejemplo, bien comentado en el código por otra parte, se han aprendido a usar estructuras relacionadas con los temporizadores de alta resolución tratados en el capítulo 3, como ***itimerval*** , o funciones como ***setitimer()***, que sirve para establecer un temporizador de tiempo real, ***gettimeofday()***, que sirve para obtener la hora con precisión de microsegundos, ***nanosleep()***, usado para dormir a un proceso durante un intervalo de tiempo de microsegundos, etc.
  Como ayuda a la ejecución y explotación de este ejemplo se ha escrito un script en bash (test_chrt.sh) que realiza un par de bucles llamando al programa y almacena los resultados en ficheros de texto. En estos se puede ver con claridad el comportamiento

determinista de los diferentes núcleos. En el primer bucle se ejecuta la aplicación sin otorgarle prioridad de tiempo real, en cambio, en el segundo bucle se le asigna prioridad 99 y un algoritmo de planificación FIFO mediante la utilidad ***chrt.***

- **test_wave.c**
  En esta segunda aplicación el comportamiento determinista del núcleo modificado con el parche de tiempo real aún es más gráfico que en la anterior. En este programa se envía una señal de onda cuadrada a un osciloscopio que esté conectado al puerto paralelo del ordenador.
  Durante la ejecución de las pruebas de la aplicación se deben variar los valores de la prioridad asignados al proceso, de forma que, solo para prioridades altas la onda es representada de forma correcta, al no ser expulsada por ninguna otra tarea del sistema. Para prioridades bajas, con el simple movimiento del ratón, la definición y demora de la onda se hace patente en el display del osciloscopio.
  Lógicamente, la ejecución de esta aplicación requiere de la conexión de un osciloscopio en el puerto paralelo, o bien, en el puerto serie, ya que el código está preparado para admitir dicha conexión.
  Con el desarrollo de este programa y de las pruebas realizadas se han aprendido a usar estructuras para la planificación como ***sched_param***, estableciendo algunas de sus prioridades como ***sched_priority***, y además se han usado funciones como ***sched_setscheduler()*** y ***clock_nanosleep()***, para establecer la política de planificación a seguir y para dormir una tarea unos pocos nanosegundos, respectivamente.

APÉNDICE E
CONTENIDO DEL CD

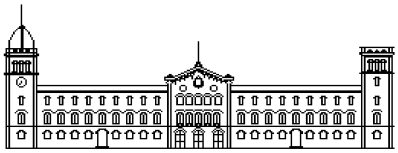
- **Memoria del trabajo final de carrera en formato pdf.**
- **Código fuente del núcleo estándar de Linux 2.6.21.6**
- **Código fuente del núcleo Linux 2.6.21.6 modificado con RT-preempt.**
- **Aplicaciones y scripts presentados en el apéndice D.**

BIBLIOGRAFÍA

- Bovet, D. P. y M. Cesati. *Understanding the Linux Kernel* (3ª ed.). O'Reilly, 2005
- Burns, A. y A. Wellings. *Real-Time Systems and Programming Languages* (3ª ed.). Addison Wesley, 2001
- Dinkel, W y otros. *KURT-Linux User Manual*. Universidad de Kansas, 2002
- Engen, M. *Capabilities for the AVR32 Linux Kernel*.. Unversidad de Noruega, 2007
- Gleixner, T. y Douglas Niehaus. *Hrtimers and beyond – transformation of the linux time(r) system*. 2006
- Kernighan, B. y D. Ritchie. *The C Programming Language*. Prentice Hall, 1988
- Kroah-Hartman, Greg. *Linux Kernel in a Nutshell*. O'Reilly, 2006
- Liu, Jane. *Real-Time Systems*. Prentice Hall, 2000
- Love, Robert. *Linux Kernel Development* (2ª ed.). Novell, 2005
- Siever, E. y otros. *Linux in a Nutshell*. O'Reilly, 2005
- Silberschatz, A. y otros. *Operating System Concepts*. John Wiley and Sons, 2001
- Stallings, William. *Operating Systems: Internals and Design Principles* (5ª ed.). Prentice Hall, 2005
- Tanenbaum, A. *Operating Systems: Design and Implementation*. Prentice Hall, 1997
- Yaghmour, Karim. *Building Embedded Linux Systems*. O'Reilly, 2003

Recursos en línea:

Código fuente y documentación del núcleo Linux estándar.
http://www.kernel.org
Documentación técnica sobre el sistema operativo GNU/Linux
http://www.ibm.com/developerworks/linux
Proyecto de documentación del sistema operativo GNU/Linux
http://www.tldp.org
Lista de correo de los desarrolladores del núcleo Linux
http://www.lkml.org
Documentación de sistemas de tiempo real. Open Source Automation Development Lab
http://www.osadl.org
Wiki del núcleo Linux de tiempo real
http://rt.wiki.kernel.org
Documentación del gestor de memoria dinámica TLSF
http://rtportal.upv.es



ENGINYERIA TÈCNICA EN INFORMÀTICA DE SISTEMES

UNIVERSITAT DE BARCELONA

Treball fi de carrera presentat el dia de de 200
a la Facultat de Matemàtiques de la Universitat de Barcelona,
amb el següent tribunal:

Dr. President

Dr. Vocal

Dr. Secretari

Amb la qualificació de: