UNIVERSIDAD DE CANTABRIA

DOCTORADO EN *Ciencia y Tecnología*

Tesis Doctoral

Soporte de Aplicaciones de Tiempo Real en Dispositivos Móviles

PhD Thesis

Support for Real-Time Applications on Mobile Devices

Realizada por

Alejandro Pérez Ruiz

Dirigida por

Mario Aldea Rivas

Michael González Harbour

Escuela de Doctorado de la Universidad de Cantabria

Santander 2020

Agradecimientos

Con estas palabras quiero agradecer a todos aquellos que de un modo u otro me han ayudado a llegar hasta este punto.

Quiero empezar dando las gracias a Mario y a Michael. Sin su apoyo, conocimientos y disposición para resolver los inconvenientes que han ido apareciendo durante el camino no estaría escribiendo estas palabras.

He tenido la suerte de trabajar en un grupo donde he pasado muy buenos momentos, y estoy convencido de que eso me ha permitido disfrutar de una de las mejores etapas de mi vida. Son muchas las personas del grupo las que han contribuido a crear el buen ambiente que ha hecho que esta etapa haya sido tan buena, pero me gustaría hacer una mención especial a Javi y Kakel por haber tenido siempre un rato para "bajar al café o a la botella de agua".

No podía olvidarme de todos los integrantes del despacho 3050 que han pasado incontables horas conmigo. Los primeros fueron César, Juan y Laura. Ellos han sido los que me guiaron en mis primeros pasos en el departamento; muchas gracias a los tres. Después, han ido llegando más personas (Noelia, Jesús, Miguel, Baldu...) que han conseguido de un modo u otro que nuestro despacho haya sido un sitio agradable. Mi última etapa de la tesis la he compartido con David, los Diegos y Ricardo. Sin miedo a equivocarme puedo decir que los últimos cinco del 3050 hemos pasado grandes momentos. No me quería olvidar de Fonso que aunque no haya sido integrante del despacho ha tenido muchas charlas conmigo y siempre me ha ayudado con cualquier tema que estuviese en su mano.

Quiero dar las gracias a mi familia, ya que este trabajo es también parte de ellos por haberme hecho quién soy. Siempre han confiado en mí y me han apoyado para que llegue hasta al final.

Finalmente, gracias Almu por ser la compañera en este viaje. Si no te hubiese conocido no estaría aquí. Este trabajo también es tuyo.

Resumen

Los dispositivos móviles como teléfonos o tabletas inteligentes han experimentado en los últimos años mejoras muy significativas tanto a nivel de hardware como de software. En la actualidad esta clase de dispositivos ofrecen potentes sistemas operativos enfocados principalmente a las necesidades del gran público. Entre los distintos sistemas operativos disponibles el más extendido es Android, y tal ha sido su expansión que se ha adaptado a dispositivos como relojes inteligentes, televisores o sistemas multimedia de vehículos, para los que no fue diseñado inicialmente. Sin embargo, a diferencia de lo que ha ocurrido con los sistemas embebidos tradicionales, no se han llevado a cabo grandes desarrollos que permitan usar sistemas como Android en entornos industriales o médicos, donde normalmente las aplicaciones utilizadas necesitan cumplir con una serie de requisitos temporales.

En esta tesis se ha llevado a cabo un estudio para encontrar las soluciones existentes que permiten utilizar aplicaciones con requisitos de tiempo real en Android. Tras analizar las soluciones existentes no se ha hallado ninguna que sea fácilmente portable y mantenible. Por lo tanto, se ha propuesto una solución que se aprovecha de las arquitecturas multinúcleo de los dispositivos móviles actuales y de los mecanismos disponibles en Android/Linux para aislar núcleos del procesador. De este modo es posible ejecutar en los núcleos aislados aplicaciones de tiempo real con pocas interferencias que puedan afectar a su respuesta temporal.

Bajo el objetivo principal de conseguir ejecutar aplicaciones de tiempo real en Android, en esta tesis se ha propuesto el uso de lenguajes de programación como C y Ada utilizados habitualmente en sistemas con requisitos temporales. Como estos lenguajes no se pueden utilizar en Android de forma nativa para aplicaciones de tiempo real, se han propuesto mecanismos que permitan su correcto funcionamiento.

Adicionalmente, en esta tesis se han desarrollado una serie de protocolos de sincronización no bloqueante para que las aplicaciones de tiempo real desplegadas en Android en núcleos aislados del procesador se puedan co-

municar sin bloqueos ni retrasos con otras aplicaciones del sistema que no tengan requisitos temporales.

Abstract

In recent years, mobile devices such as smartphones or tablets have experienced very significant improvements, both in terms of hardware and software. Nowadays, this class of devices offers powerful operating systems mainly focused on the needs of the general public. Among the different operating systems available, Android is the most widespread. Such has been its expansion that it has been adapted to devices such as smartwatches, televisions or car multimedia systems, for which it was not initially designed. However, unlike what has happened with traditional embedded systems, no major developments have been made to allow the use of systems such as Android in industrial or medical environments, where applications normally need to meet a series of timing requirements.

We started this work by carrying out a study to find existing solutions that allow the use of applications with real-time requirements in Android. After analysing these existing solutions, no one has been found that is easily portable and maintainable. Therefore, a novel solution has been proposed that takes advantage of the multicore architectures of current mobile devices and the mechanisms available in Android/Linux to isolate processor cores. By applying this solution, it is possible to run real-time applications on the isolated cores with little interference on their response times.

Under this main objective of executing real time applications in Android, we have proposed the use of programming languages such as C and Ada commonly used in systems with timing requirements. As these languages cannot be used natively in Android for real-time applications, mechanisms have been proposed to allow their correct operation.

Additionally, a series of non-blocking synchronization protocols have been developed, through which real time applications deployed in Android on isolated processor cores can communicate without blockages or delays with other non-real-time applications executing in the same system.

Índice general

Índice de figuras

Índice de tablas

Introducción

1

> *El comienzo es la parte más importante de la obra.*
>
> **— Platón**
> (Filósofo griego)

En las últimas décadas los sistemas computacionales han incrementado su potencia y disminuido su tamaño de una forma vertiginosa. Una prueba evidente de esto son los dispositivos móviles actuales, tales como tabletas o teléfonos inteligentes. Desde el primer teléfono móvil comercializado en 1983 hasta los teléfonos inteligentes actuales se ha conseguido introducir en ellos microprocesadores con altas capacidades, pantallas de gran resolución, módem para comunicaciones utilizando redes 5G o sensores fotográficos con cientos de millones de píxeles. La evolución de los teléfonos inteligentes ha provocado que cada vez sean usados en más ámbitos, convirtiéndose en el dispositivo electrónico más utilizado por miles de millones de personas en todo el mundo. En los últimos años las ventas de teléfonos inteligentes se han disparado y ya superan con fuerza a las de los ordenadores personales, tal y como se puede ver en la Figura 1.1.

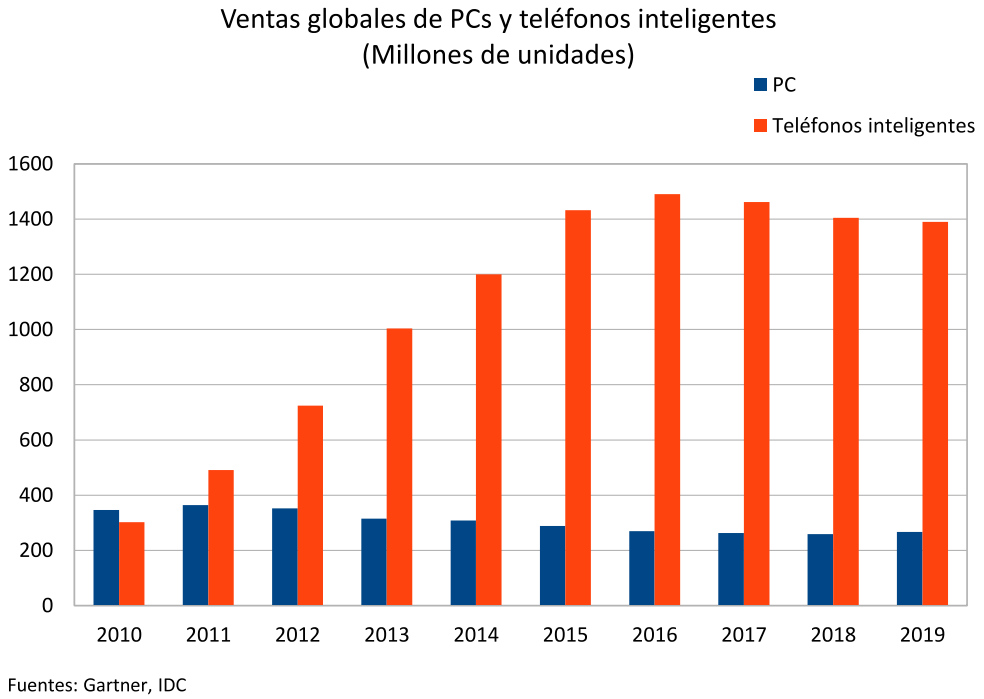

Fig. 1.1: Ventas globales de ordenadores personales y teléfonos inteligentes en los últimos años.

El aumento en ventas, y en especial la mejora en el hardware y el software que acompaña a los teléfonos inteligentes, ha despertado el interés en la industria y en la comunidad científica para utilizar estos dispositivos en distintos ámbitos que se alejan de su concepción original, la cual era ofrecer un sistema adaptado a un hardware con muy pocos recursos junto a un consumo energético limitado.

Un ámbito donde se ha producido un creciente interés es el del tiempo real, donde se busca tener sistemas temporalmente predecibles. En el entorno médico se han llevado a cabo estudios para utilizar el sistema operativo Android [1] [2] [3] [4], lo mismo ocurre en el campo militar [5] [6] o incluso en la industria aeroespacial [7]. En todos estos ejemplos existe una necesidad de que el sistema en su conjunto responda en mayor o menor medida con unos tiempos acotados y predecibles para su buen funcionamiento, es decir, estamos ante lo que se conoce como sistemas de tiempo real.

El surgimiento de estos nuevos ámbitos de aplicación requiere explorar soluciones que permitan el uso del sistema operativo Android con aplicaciones que posean requisitos temporales.

El resto de este capítulo de introducción está organizado como se describe a continuación. En la Sección 1.1 se describe qué son los sistemas de tiempo real. A continuación, en la Sección 1.2 se introduce brevemente en qué consisten los sistemas operativos de tiempo real y cuáles son sus características principales. Posteriormente, en la Sección 1.3 se habla de cómo está definido el sistema operativo Android a nivel de software y cómo ha sido su evolución en los últimos años. En la Sección 1.4 establecemos cuáles son los objetivos perseguidos en este trabajo. Y finalmente, en la Sección 1.5 exponemos la organización de la memoria de la presente tesis.

1.1 Sistemas de tiempo real

Los sistemas de tiempo real son sistemas informáticos que se hallan en constante interacción con el entorno que les rodea. Además, para su correcto funcionamiento es imprescindible que no solo los resultados generados sean correctos, sino que las respuestas a los estímulos del entorno se produzcan dentro de un plazo de tiempo determinado [8].

Esta clase de sistemas está presente en multitud de escenarios de nuestro día a día. Los podemos hallar en decenas de automatismos de un coche (ABS, airbag, inyección, etc.), en sistemas de fabricación integrada, en la distribución de la energía eléctrica, en el control de edificios, en telefonía móvil o en sistemas multimedia. Su uso está tan extendido que actualmente

el número de estos sistemas en funcionamiento es superior al de sistemas computacionales de propósito general.

No todos los sistemas de tiempo real operan bajo el mismo nivel de criticidad. Por lo tanto, estos se pueden clasificar en función de la criticidad asociada a los plazos temporales de respuesta [9] [10]:

- **Tiempo real estricto (*hard real-time*):** Todas las respuestas que debe dar el sistema tienen que terminar dentro de un plazo temporal especificado. Si el plazo de tiempo para dar una respuesta o realizar una acción concreta no se cumple puede haber consecuencias catastróficas en el sistema. Un ejemplo de esto puede ser el control de vuelo automático de una aeronave: un retraso en la realización de una acción puede ocasionar un accidente con consecuencias muy graves.

- **Tiempo real laxo (*soft real-time*):** Los tiempos de respuesta son importantes pero el sistema puede seguir funcionando aunque algún plazo temporal no se cumpla. Normalmente en esta clase de sistemas los fallos en el cumplimiento de los plazos temporales desembocan en una degradación del rendimiento. Un ejemplo de un sistema de tiempo real laxo es el de una videoconferencia donde aunque de forma esporádica se produzca la perdida de algún plazo temporal la comunicación se puede seguir manteniendo.

En muchas ocasiones, independientemente del nivel de criticidad, los sistemas de tiempo real son utilizados para resolver problemas de gran tamaño y complejidad. Por esta razón en esta clase de sistemas se utiliza hardware cada vez más potente y con mayor número de unidades computacionales. Normalmente los sistemas complejos están compuestos por un conjunto de tareas que son ejecutadas de forma concurrente para el control global del sistema. Tanto para gestionar el hardware como las distintas tareas, generalmente se utilizan sistemas operativos con características de tiempo real, que se encargan de ejecutar y planificar todas las tareas sobre el hardware disponible, además de posibilitar el uso de los dispositivos de entrada/salida.

1.2 Sistemas operativos de tiempo real

En los sistemas informáticos un sistema operativo es el software encargado de gestionar los recursos hardware y ofrecer los servicios que necesitan las aplicaciones para funcionar [11] [12]. Dentro de los sistemas operativos podemos encontrar aquellos que son de tiempo real, que han sido diseñados para no solo ofrecer corrección lógica en los resultados computados, sino

para ofrecer garantías temporales en la ejecución de las aplicaciones que gestionan [13] [14].

Muchos de los sistemas operativos modernos, independientemente de su naturaleza temporal, tienen dos modos de funcionamiento: modo kernel y modo usuario [11]. El sistema operativo se ejecutará en modo kernel (también llamado modo supervisor) cuando necesite realizar cualquier operación directamente sobre el hardware. El resto del software se ejecuta en modo usuario, donde solo un pequeño subconjunto de instrucciones máquina están disponibles. En los sistemas operativos de tiempo real el modo kernel tiene que ofrecer servicios y funciones para las aplicaciones con requisitos temporales (políticas de planificación, protocolos de sincronización, gestión del tiempo...).

Los sistemas operativos diseñados para ejecutar aplicaciones con requisitos temporales deben dar soporte a las siguientes dos necesidades para su correcto funcionamiento:

- **Planificación de tareas**: El sistema operativo debe tener la capacidad de determinar la tarea que debe ejecutar en cada uno de los procesadores del sistema. Esto se consigue gracias a los algoritmos de planificación implementados en el sistema. De forma bastante extendida nos encontramos con que esta clase de algoritmos utilizan políticas de planificación basadas en el tiempo o en prioridades [15].

- **Mecanismos de comunicación fiables entre las tareas**: Cuando tenemos diversas tareas ejecutándose concurrentemente es necesario que existan primitivas que permitan la comunicación y sincronización de manera fiable asegurando la integridad de los datos compartidos entre las distintas tareas.

Los sistemas operativos de tiempo real se pueden clasificar como sistemas de tiempo real estricto o laxo. Como ya se comentó en la sección anterior cuando nos referimos a tiempo real estricto la perdida de plazos temporales no está permitida, en cambio en los sistemas operativos de tiempo real laxo se pueden tolerar algunas perdidas de plazo, aunque es deseable que esto ocurra de la manera más infrecuente posible.

Actualmente existen numerosas implementaciones de sistemas operativos de tiempo real [16] [17]. Muchas de estas implementaciones siguen algunos de los siguientes estándares y especificaciones:

- **POSIX (*Portable Operating System Interface*)[18]:** Este estándar define una interfaz portable para las aplicaciones que se ejecutan sobre los

sistemas operativos UNIX. Su principal objetivo es que las aplicaciones puedan ser portadas a nivel de código fuente entre distintos sistemas operativos realizando solo la recompilación del código de la aplicación. Existen diversos sistemas operativos de tiempo real que son compatibles con el estándar POSIX, algunos ejemplos son MaRTE OS [19], VxWorks [20], PikeOS [21], e incluso las versiones más actuales de los sistemas basados en Linux incorporan numerosas características de tiempo real.

- **ITRON (*Industrial TRON Specification OS*)[22]:** Esta especificación desarrollada en Japón está diseñada para sistemas operativos de tiempo real sobre dispositivos embebidos. El diseño se basa en una política de adaptación flexible del kernel para cada una de las aplicaciones, de modo que se reduce la sobrecarga en los tiempos de ejecución y el uso de memoria. Existe también otra especificación denominada μITRON basada en ITRON pero con un número más reducido de funciones para que pueda implementarse en procesadores cuyo rendimiento y memoria sean limitados. Algunos sistemas operativos de tiempo real que tienen compatibilidad con el API definido en el estándar μITRON son RTEMS [23] y eCOS [24].

- **AUTOSAR (*AUTomotive Open System ARquitecture*)[25]:** Este estándar busca crear una arquitectura software abierta para las unidades de control electrónico utilizadas en el ámbito de la automoción. Define interfaces para las aplicaciones y el entorno en tiempo de ejecución y además establece una metodología de desarrollo. Algunos sistemas operativos que utilizan este estándar son RTA-OS [26] y VSTAR OS [27].

El avance que se ha producido en el software y el hardware en las última décadas ha provocado que en algunos entornos industriales no se ejecuten solo aplicaciones cerradas muy limitadas, sino que se utilicen sistemas de tiempo real con más funcionalidades, mejores interfaces de usuario y que incluso puedan convivir con aplicaciones con distintos niveles de criticidad. El hecho de intentar integrar cada vez más funcionalidad en los sistemas ha generado la necesidad de tener sistemas particionados donde aplicaciones heterogéneas se ejecutan sobre una única plataforma computacional. En estos sistemas las aplicaciones están aisladas temporal y espacialmente. De este modo es posible ejecutar aplicaciones con distintos grados de criticidad. Un ejemplo de sistemas particionados es el propuesto por el estándar ARINC 653 (*Airlines Electronic Engineering Committee, Aeronautical Radio INC, 2010*) utilizado en el ámbito de la aviónica. Una implementación que utiliza el

estándar ARINC 653 es LithOS [28] basada en el hipervisor XtratuM [29]. El estándar no solo define las reglas para el aislamiento temporal y espacial de las aplicaciones, sino que además define interfaces para la comunicación entre las distintas particiones de un mismo nodo.

No solo los sistemas particionados utilizados con un hipervisor son la única solución para ejecutar aplicaciones con distintos niveles de criticidad en una misma unidad computacional. Actualmente los sistemas operativos como Linux han ido incorporando características de tiempo real que permiten ejecutar aplicaciones con mayores garantías temporales. De hecho prácticamente todo el kernel de Linux es expulsable. Obviamente, junto a estas aplicaciones con requisitos de tiempo real pueden coexistir otras aplicaciones clásicas de Linux.

Hasta la realización de esta tesis no existía una solución portable y fácilmente mantenible para ejecutar aplicaciones de tiempo real en sistemas operativos diseñados para dispositivos móviles. Debido a que el sistema operativo para teléfonos inteligentes más extendido en todo el planeta es Android, en la siguiente sección se realizará una introducción sobre él.

1.3 Android

La palabra Android se asocia inmediatamente con el sistema operativo que se ejecuta en millones de teléfonos, pero Android es más que un simple sistema operativo, es una plataforma completa para el desarrollo y ejecución de aplicaciones. La plataforma Android fue creada en noviembre de 2007 con el respaldo de Google y la fundación Open Handset Alliance (un consorcio de compañías de telecomunicaciones, hardware y software). Esta plataforma desde su nacimiento ha ido expandiéndose por todo el mercado global hasta llegar a ser la más utilizada en los teléfonos inteligentes alcanzando una cuota de mercado de más del 74 % en septiembre de 2020 [30].

Android está formado por diferentes capas de componentes software tal y como se ilustra en la Figura 1.2. La capa más baja se corresponde con el kernel de Linux, el cual proporciona el funcionamiento básico del sistema, como por ejemplo el manejo de la memoria, los procesos y los drivers o gestores de dispositivos. El kernel de Android tiene algunas particularidades que lo hacen diferir del kernel tradicional utilizado en las distribuciones Linux más populares. Por ejemplo, se incluye un gestor para eliminar procesos cuando hay poca memoria disponible llamado en inglés *Low-Memory Killer*. Este tiene en cuenta diferentes prioridades y umbrales para cada tipo de aplicación a la hora de elegir los procesos a eliminar cuando la memoria disponible

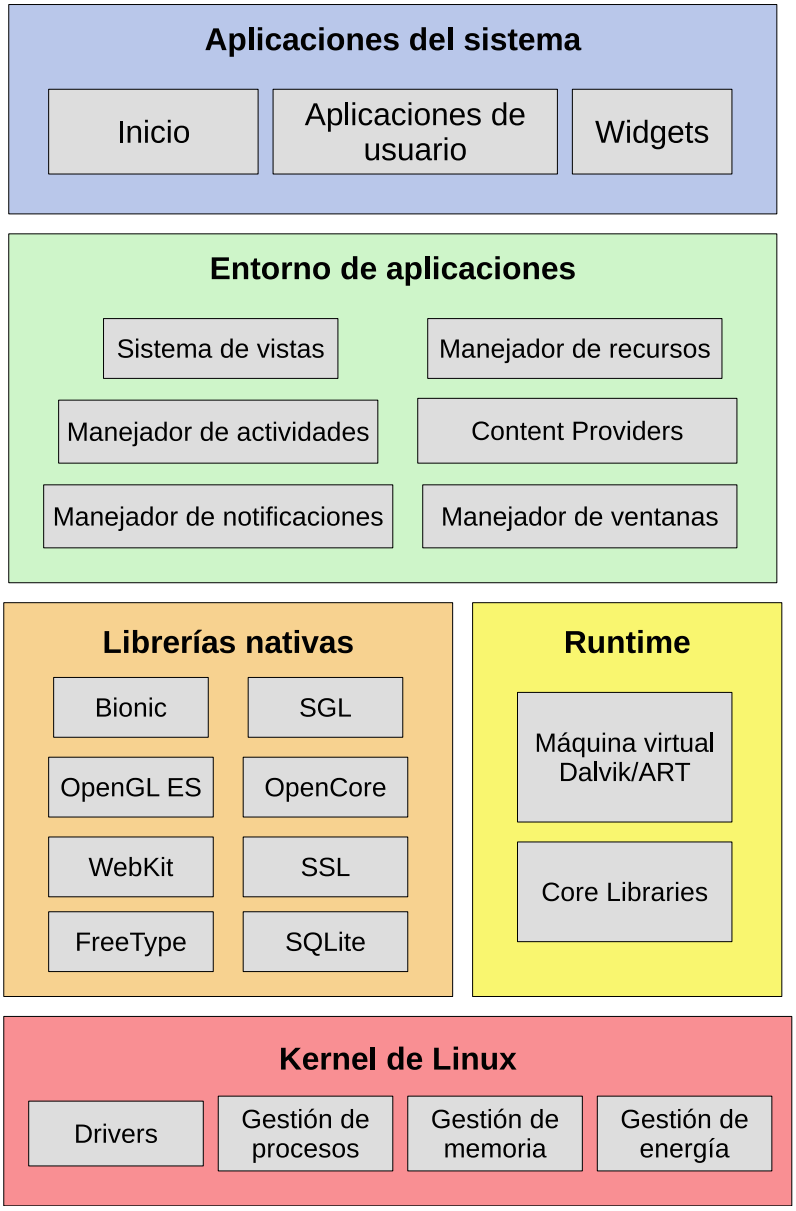

Fig. 1.2: Arquitectura software de Android.

disminuye. Otro cambio relevante de Android con respecto a Linux es el mecanismo utilizado para compartir memoria entre procesos bautizado como *ASHMEM*. Con dicho mecanismo se pretende mejorar la seguridad del sistema y evitar la pérdida de recursos en la memoria.

Encima de la capa compuesta por el kernel, hay otra capa que contiene un conjunto de librerías nativas (escritas en C o C++) que son compiladas para cada arquitectura hardware específica. Entre ellas se incluye la librería *Bionic*, una implementación de la librería estándar C específica de Android. *Bionic* sustituye a la versión de GNU de dicha librería, denominada *glibc*, que es la utilizada en las distribuciones Linux más populares.

Un componente clave de Android es su máquina virtual para ejecutar código Java. Las versiones actuales de Android incroporan una máquina virtual denominada ART (*Android RunTime*)[31], la cual es la sucesora de la ya obsoleta *Dalvik*. Se optó por sustituir la máquina virtual *Dalvik* porque realizaba la compilación en tiempo de ejecución, mientras que ART lleva a cabo la compilación solo una vez justamente después de la instalación de las aplicaciones y, por ende, se mejora la eficiencia y el consumo energético. En este sistema operativo la máquina virtual se encarga de ejecutar las aplicaciones compiladas en un formato específico (*Dalvik executable*). Además, cada aplicación se ejecuta como una instancia de la máquina virtual con su propio espacio de memoria y recursos.

Actualmente las aplicaciones desarrolladas para Android suelen estar escritas en los lenguajes de programación Java o Kotlin [32], ya que utilizando estos lenguajes los programadores se aprovechan de una serie de funciones (entorno de aplicaciones en la Figura 1.2) que facilitan el uso del hardware y de los recursos del sistema. A pesar de que Java o Kotlin son los lenguajes recomendados por Google para la creación de aplicaciones en Android, sería posible utilizar cualquier otro lenguaje de programación para el que tengamos un compilador compatible con Linux y el hardware del dispositivo que se esté utilizando.

Android está diseñado para ofrecer a los desarrolladores una plataforma modular y fácilmente extensible, de tal modo que los componentes de las aplicaciones puedan ser reutilizados desde otras aplicaciones. Para conseguir esto en este sistema operativo existen cuatro componentes principales que pueden conformar las aplicaciones escritas en Java o Kotlin:

- **Actividades**: Proporciona la interfaz gráfica de usuario. Las aplicaciones suelen tener varias actividades para establecer el aspecto gráfico.
- **Servicios**: Son los procesos que se ejecutan en segundo plano sin la necesidad de que exista una interacción con el usuario. Normalmente son utilizados para operaciones que perduran en el tiempo.
- **Receptor de anuncios**: Es utilizado para interceptar y manejar eventos del sistema o aquellos designados específicamente por el usuario.
- **Proveedores de contenido**: Permite compartir información almacenada entre las distintas aplicaciones instaladas en el sistema.

Aparte de los componentes descritos anteriormente Android ofrece a los desarrolladores tres mecanismos para que las aplicaciones o los procesos se puedan comunicar entre sí. Exactamente contamos con tres: *intents*, *binders*

y memoria compartida. Los *intents* son utilizados para mandar mensajes (*bundles*) entre aplicaciones. Los *binders* sirven para realizar llamadas a métodos que se encuentran en otras aplicaciones. Y finalmente Android ofrece una implementación propia para la memoria compartida que se denomina *ASHMEM*.

En la plataforma Android es posible implementar partes de las aplicaciones usando lenguajes como C y C++, es decir, que de forma oficial existen un conjunto de herramientas [33] que contemplan este uso. Esto permite codificar parte de las aplicaciones para que no se ejecute sobre la máquina virtual Java, de tal modo que pueda ser más eficiente.

1.3.1 Evolución de Android

Android es un sistema operativo que está en constante evolución. Desde su origen ha ido añadiendo nuevas características y funcionalidades que lo han convertido en un sistema operativo utilizado en dispositivos como teléfonos inteligentes, tabletas, televisores o incluso para el sistema multimeda de automóviles.

Este sistema operativo ha ido cambiando con cada nueva versión gracias al proyecto conocido con el nombre en inglés *Android Open Source Project (AOSP)* [1]. Dicho proyecto está liderado por Google y lo que hace es ofrecer el código fuente básico del sistema para que se pueda personalizar, crear variantes y portar a nuevos dispositivos. Sobre este código se deben añadir el resto de componentes que conforman la plataforma completa. Estos componentes no tienen que ser necesariamente de código abierto, ya que generalmente existirá código privativo (aplicaciones de usuario, firmware, drivers, etc). La parte del código que se hace pública con cada nueva versión de Android tiene licencia de software Apache 2.0, a excepción del kernel Linux que posee una licencia GNU GPL. Por lo tanto, el kernel al tener licencia GPL exige la publicación del código fuente y que todos los desarrollos derivados de él conserven esta misma licencia, en cambio la licencia Apache 2.0 es más permisiva porque no requiere que los trabajos derivados sean publicados bajo el mismo tipo de licencia, y tampoco exige la liberación del código fuente.

El proceso completo que se sigue en el desarrollo y despliegue de una nueva versión de Android en un dispositvo implica los siguientes pasos:

1. Liberación del código fuente base de la nueva versión por parte de Google.

[1] `https://source.android.com/`

Versión	Nombre	API	Porcentaje de uso
2.3.3-2.3.7	Gingerbread	10	0,3 %
4.0.3-4-0-4	Ice Cream	15	0,3 %
4.1-4.3	Jelly Bean	16-18	3,2 %
4.4	KitKat	19	6,9 %
5.0-5.1	Lollipop	21-22	14,5 %
6.0	Marshmallow	23	16,9 %
7.0-7.1	Nougat	24-25	19,2 %
8.0-8.1	Oreo	26-27	28,3 %
9	Pie	28	10,4 %

Tabla 1.1: Porcentaje de uso de las versiones de Android en mayo de 2019 [34]

2. Desarrollo de la capa denominada *HAL (Hardware Abstraction Layer)* para conseguir que el nuevo software funcione sobre el hardware específico del dispositivo.

3. Implementación de las funciones básicas del dispositivo, como por ejemplo llamadas o conectividad.

4. Añadir características extras y personalizar el sistema. Normalmente en este paso los fabricantes de dispositivos Android añaden su software propio y personalizan el aspecto de la interfaz gráfica de usuario.

5. Verificación del software y de la estabilidad del sistema a través de numerosos tests. En este paso se incluye la verificación de todos los estándares de conformidad.

Los pasos previos demuestran que desplegar una nueva versión de Android no es algo trivial y se necesita llevar a cabo un proceso complejo, ya que al fin y al cabo se realiza una combinación de los cambios implementados por Google con los que han realizado los fabricantes de los dispositivos. Esto en gran parte ha provocado que exista una fragmentación muy significativa en Android. Tal y como se ilustra en la Tabla 1.1, en mayo de 2019 la versión más reciente de Android que existía en esa fecha tan solo estaba desplegada en el 10.4 % de los dispositivos utilizados a nivel global.

Google conoce el problema que tiene con la fragmentación de su sistema operativo y ha intentando mitigarlo a través de algunos proyectos como el denominado Treble [35]. El objetivo principal de este proyecto es separar la implementación del software que depende del fabricante del código desarrollado por Google. Para lograr esto se introduce una nueva interfaz entre las dos partes del software que se quieren separar. De este modo, la actualización del sistema operativo se abstrae del hardware del dispositivo facilitando gran parte del proceso de actualización del software. Debido al vasto número de dispositivos y fabricantes que actualmente desarrollan dispositivos con el sistema operativo Android, proyectos como Treble necesitan tiempo y una coordinación intensa para utilizarse de manera homogénea. Además, Treble no facilita la actualización y la integración del código Java específico desarrollado por los fabricantes. Todo esto provoca que actualmente la fragmentación haya disminuido con respecto a años previos, sin embargo aún está muy presente en el ecosistema Android, lo que dificulta enormemente desarrollos o adaptaciones de nuevas funcionalidades que se quieran aplicar de forma global sobre todas las versiones de Android.

1.4 Objetivos de la tesis

Tal y como se ha expuesto anteriormente, el sistema operativo para dispositivos móviles más utilizado en la actualidad a nivel mundial es Android. Por otro lado, existe un interés importante en poder utilizar dispositivos basados en Android como plataformas para aplicaciones con requisitos de tiempo real. Al ser Android un sistema basado en Linux en el que gran parte de su código fuente es accesible, es posible analizarlo en profundidad para encontrar alguna solución que mejore su respuesta temporal. Por estos motivos el objetivo principal de esta tesis es identificar y desarrollar un conjunto de mecanismos y herramientas cuyo uso permita utilizar Android como plataforma sobre la que ejecutar aplicaciones con requisitos de tiempo real laxo.

Para cumplir con el objetivo principal se pretenden llevar a cabo las siguientes aportaciones:

- Evaluar las soluciones existentes que tratan de mejorar la predictibilidad temporal de Android. De este modo determinaremos cuáles son sus puntos fuertes y débiles.

- Buscar un mecanismo que permita aislar núcleos del procesador donde poder ejecutar aplicaciones de tiempo real con el menor número de interferencias posibles, de tal modo que las aplicaciones con requisitos

temporales puedan ejecutarse con ciertas garantías. Por consiguiente, se ejecutarán diferentes tipos de aplicaciones sobre las distintas unidades de procesamiento que ofrecen las arquitecturas multinúcleo. Todo esto es posible gracias al aumento de la potencia hardware que están experimentando los dispositivos que utilizan el sistema operativo Android.

- Para que un sistema operativo como Android pueda ser capaz de ejecutar aplicaciones con requisitos temporales es necesario tener lenguajes de programación que ofrezcan características de tiempo real. Para el uso del lenguaje de programación C, se identificarán las limitaciones para tiempo real que tiene Android cuando se utiliza su librería *Bionic* y se propondrá una alternativa. Además, se demostrará que es posible utilizar el lenguaje de programación Ada para el desarrollo de aplicaciones con requisitos temporales sobre Android.

- Desarrollar protocolos de sincronización no bloqueantes que permitan compartir datos entre las aplicaciones de tiempo real y las de no tiempo real en arquitecturas multinúcleo.

El cumplimiento de todos los objetivos previos persigue obtener una solución para cualquier versión actual Android donde sea posible ejecutar aplicaciones de tiempo real laxo. El principal requisito que debe tener la solución presentada en esta tesis es que sea fácilmente portable y mantenible en cualquiera de los dispositivos multinúcleo que son utilizados con Android actualmente. La constante evolución de Android convertiría en una labor muy compleja tener soluciones de tiempo real para este sistema que impliquen grandes modificaciones internas en el software, por este motivo nos hemos impuesto el requerimiento de que nuestra solución sea portable y fácilmente mantenible a lo largo del tiempo.

1.5 Organización de la tesis

En el Capítulo 2 de la presente tesis se exploran las soluciones que diversos autores han presentado a lo largo de los últimos años para añadir una respuesta temporal más predictible en Android. En primer lugar se resumen las principales limitaciones que tiene este sistema operativo para la ejecución de aplicaciones de tiempo real y cuáles son sus posibles soluciones. Posteriormente, se hace un repaso de cuál es la situación actual de los sistemas operativos basados en Linux para ejecutar aplicaciones con requisitos temporales. A continuación, se detallan las soluciones de tiempo real que existen

para Android dentro del ámbito académico y empresarial. Finalmente, se realiza una caracterización de todas las soluciones descritas anteriormente.

En el Capítulo 3 se expone nuestra solución para aislar núcleos en dispositivos multinúcleos que ejecutan sistemas operativos basados en Linux, de tal forma que en ellos se puedan ejecutar aplicaciones con requisitos temporales. Durante este capítulo se describen cómo son las actuales arquitecturas multinúcleo usadas en los dispositivos Android, cuáles son las políticas habituales de planificación de tareas en este tipo de arquitecturas, y posteriormente describimos cómo funcionan los algoritmos de planificación en el kernel de Linux/Android. A continuación, estudiamos cuáles son los mecanismos de aislamiento que existen en Linux, describimos su funcionamiento, analizamos cuáles son los más adecuados para nuestros objetivos, y proponemos cómo usarlos en conjunto de forma adecuada. Finalmente, se evalúa nuestra propuesta para ver la mejora temporal que nos pueden ofrecen los mecanismos de aislamiento.

En el Capítulo 4 se analizan las limitaciones que tiene el sistema operativo Android para ejecutar aplicaciones de tiempo real debido al uso de la librería *Bionic*. Se propone una alternativa para poder utilizar la librería tradicional *glibc* en Android y así eliminar las limitaciones. Además, se describen los pasos necesarios para hacer uso del lenguaje de programación Ada en el desarrollo de aplicaciones con requisitos de tiempo real en Android.

En el Capítulo 5 se lleva a cabo el desarrollo y evaluación de cinco protocolos de sincronización no bloqueante, que serán utilizados para permitir compartir datos entre aplicaciones de tiempo real y de no tiempo real coexistentes en un dispositivo móvil multinúcleo con el sistema operativo Android. De este modo, será posible tener en Android aplicaciones con requisitos temporales aisladas en un núcleo del procesador, y que además se puedan comunicar con el resto de aplicaciones del sistema sin perder su predictibilidad temporal.

Finalmente, en el Capítulo 6 se exponen las principales conclusiones y resultados de este trabajo de investigación y se describen brevemente algunas posibles líneas de investigación futuras.

Estado del arte

2

> *Aquellos que no recuerdan el pasado están condenados a repetirlo.*
>
> — **George Santayana**
> (Poeta, ensayista y novelista)

Android es un sistema operativo diseñado inicialmente para teléfonos inteligentes que está basado en una modificación del kernel de Linux y en software de código abierto. A pesar de su objetivo inicial este sistema ha ido evolucionando a lo largo del tiempo, y ya no solo se utiliza con teléfonos inteligentes sino que está siendo empleado en multitud de entornos con distintos tipos de requisitos tales como televisores, relojes, sistemas multimedia para automóviles, etc.

La ejecución de aplicaciones de tiempo real en Android lleva desde hace años despertando el interés de la industria y de la comunidad científica. Prueba de ello son los diversos estudios previos [36] [37] [38] [39] [40] [41] que han evaluado la viabilidad de Android para ser utilizado en entornos con requisitos temporales. Todos estos trabajos han concluido que la plataforma Android sin la aplicación de alguna modificación o mecanismo extra no es apropiada para la ejecución de aplicaciones de tiempo real. En el estudio realizado por Mongia y Madisetti [37] se muestran algunos experimentos donde los tiempos de respuesta de peor caso de las tareas con requisitos temporales se ven incrementados entre 1 *ms* y 500 *ms* debido a las interferencias generadas por el sistema operativo. Esta clase de interferencias son inasumibles para la mayoría de sistemas de tiempo real. Por ejemplo, en el ámbito del control de movimientos suele ser habitual tener requisitos temporales del orden de los pocos milisegundos. En el ámbito de la reacción a una orden humana los requisitos temporales se sitúan en el marco de las decenas de milisegundos.

El trabajo realizado por Maia et al. [36] identifica componentes claves de la plataforma Android que impiden utilizar Android en entornos de tiempo real. Los principales problemas que expone dicho trabajo son los siguientes:

- Android utiliza una implementación propia de la librería estándar C denominada *Bionic*. *Bionic* está optimizada para dispositivos móviles con recursos limitados y posee una licencia BSD que no obliga a que todo el código que se cree a partir de ella herede la condición de código abierto. Además, la librería *Bionic* proporciona y define las llamadas al

sistema y otras funciones básicas, pero tiene algunas carencias comparada con la librería tradicional, como por ejemplo la falta de protocolos de sincronización o limitaciones en los mecanismos de sincronización por exclusión mutua (mutexes), siendo esto un requisito indispensable para cualquier aplicación de tiempo real.

- La máquina virtual Java utilizada (Dalvik/ART) y los tiempos de respuesta del kernel de Android impiden tener tiempos de respuesta acotados.

- El kernel de Linux utiliza por defecto el planificador llamado "*completamente justo*" (*Completely Fair Scheduler*, en inglés). Este planificador proporciona fracciones temporales dinámicas entre las distintas tareas del sistema para intentar ser lo más justo con todas las tareas que deben ser ejecutadas. Esto es claramente incompatible con los requisitos de una aplicación de tiempo real en la que deben priorizarse actividades más urgentes frente a las menos urgentes.

Debido a las limitaciones citadas anteriormente, en el trabajo de Maia *et al.* se propusieron cuatro posibles soluciones para que Android sea adecuado para la ejecución de aplicaciones de tiempo real. En la Figura 2.1a se ilustra la primera solución propuesta, que consiste en ejecutar todas las aplicaciones con requisitos temporales sobre un kernel de Linux con características de tiempo real. La siguiente solución que se plantea consiste en añadir una máquina virtual con características de tiempo real (RT-JVM); de este modo gracias a un kernel de Linux de tiempo real se podrían ejecutar programas Java de tiempo real (ver Figura 2.1b). En la Figura 2.1c se ilustra la siguiente solución que propone utilizar los servicios de un kernel de tiempo real junto con una modificación de la máquina virtual de Android (ART VM extendida para incluir características de tiempo real). La última solución planteada en la Figura 2.1d utiliza un hipervisor de tiempo real capaz de ejecutar el sistema operativo Android en concurrencia con un sistema operativo de tiempo real.

Todas las propuestas mostradas en la Figura 2.1 requieren adaptarse al hardware específico que se vaya a utilizar y también deben ser portadas a la versión exacta de Android que se necesite en cada situación. Las propuestas de la Figura 2.1a y la Figura 2.1d son las que requieren menos proceso de adaptación, no obstante el kernel de Linux que utiliza Android no sigue la línea principal de desarrollo de Linux, ya que se realizan cambios sustanciales para adaptarse a los requisitos de energía y rendimiento que exigen los dispositivos móviles. Por lo tanto no es posible aplicar de forma directa las soluciones actuales compatibles con el kernel de Linux para otorgarle más

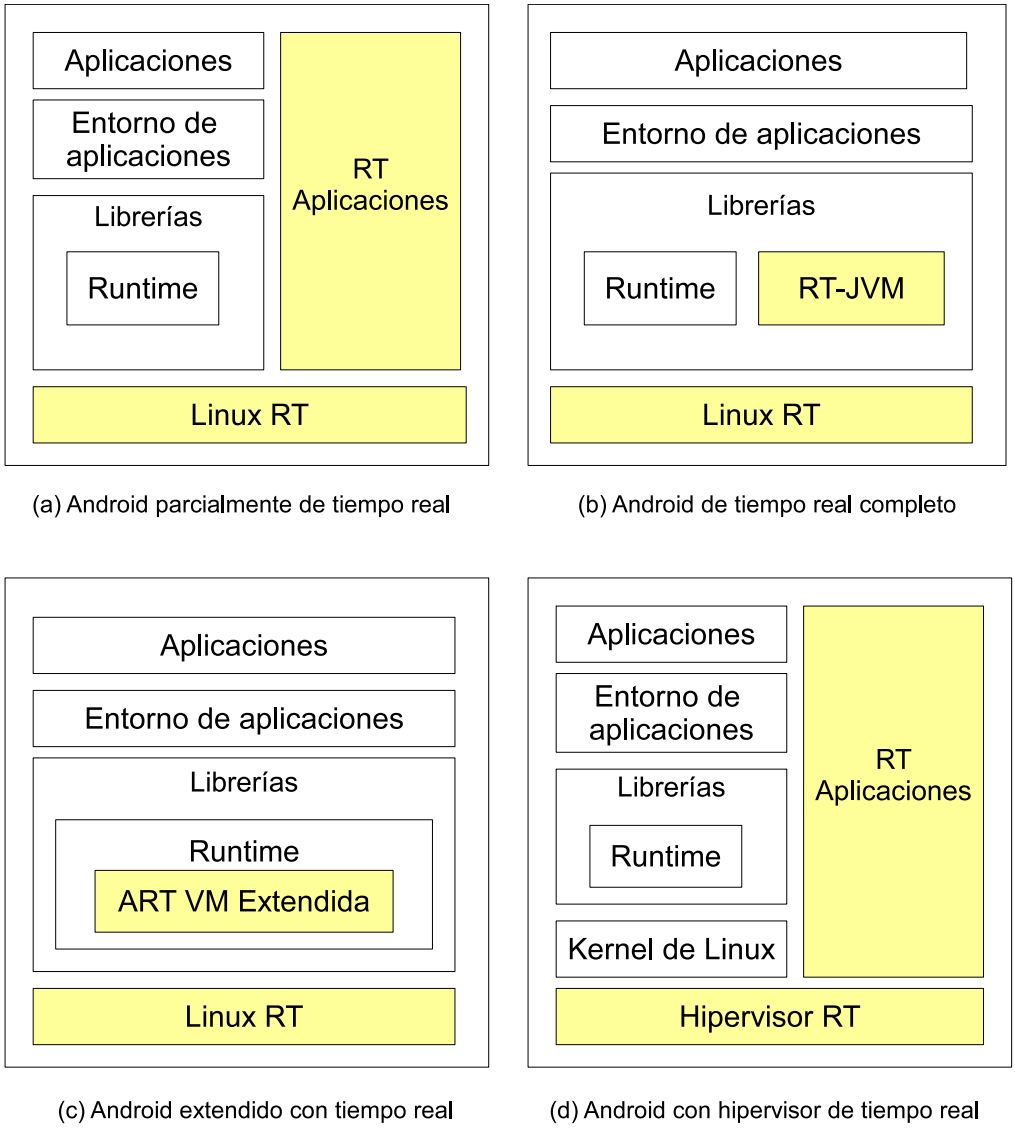

Fig. 2.1: Soluciones teóricas para la integración de tiempo real en Android [36]. Las partes de color amarillo indican cambios en la arquitectura de Android.

propiedades de tiempo real, ni tampoco es trivial adaptar la interfaz de conexión de los hipervisores existentes con el kernel de Android/Linux.

En lo que resta de este capítulo vamos a ir describiendo las soluciones disponibles en la actualidad que podrían ser aplicadas en Android para proporcionar a este sistema un comportamiento temporal más predecible. Comenzaremos con la Sección 2.1 describiendo aquellas que son utilizadas en los sistemas Linux tradicionales y, por consiguiente, aplicables a Android siempre y cuando se hagan las modificaciones necesarias. A continuación en la Sección 2.2 se especificarán aquellas soluciones diseñadas explícitamente para ser aplicadas sobre la plataforma Android. Finalmente, en la Sección 4.3

se resumirán las características de cada una de las soluciones y adaptaciones descritas en este capítulo.

2.1 Linux de tiempo real

Los sistemas de tiempo real necesitan de una plataforma donde se ejecute un sistema operativo con características de tiempo real (*RTOS, Real-Time Operating System*, en inglés). Hay algunos aspectos realmente importantes en estos sistemas de tiempo real como son garantizar el uso exclusivo de ciertos recursos para tareas de tiempo real, distintas políticas de planificación de tareas, contar con mecanismos de sincronización entre tareas para evitar problemas como el de la inversión de prioridad, y la utilización de un kernel en el sistema operativo que proporcione tiempos de respuesta acotados, de forma que así sea capaz de garantizar los requerimientos temporales de las tareas que están bajo su gestión.

Los sistemas operativos basados en el kernel de Linux, a pesar de tener un claro propósito de uso general, han ido evolucionando a lo largo del tiempo en términos de características y comportamiento temporal, de tal modo que han ampliado su capacidad para ser utilizados en entornos más específicos. Prueba de ello es el esfuerzo realizado por diversos desarrolladores para mejorar su rendimiento en entornos donde el comportamiento temporal de las aplicaciones es un requisito. A través de la modificación de aspectos claves del comportamiento interno del kernel del sistema con el parche denominado PREEMPT_RT [42] se ha conseguido mejorar sustancialmente sus características para utilizarse en entornos con demandas temporales. De este modo, utilizando el parche PREEMPT_RT se consigue un mayor grado de predictibilidad en los tiempos de respuesta de los servicios proporcionados por el sistema operativo. Además, cabe destacar que gran parte de las características de tiempo real ofrecidas por el parche PREEMPT_RT se han ido añadiendo a la línea principal del kernel de Linux. En la siguiente subsección se va a describir con más detalle qué ventajas nos ofrece la aplicación del parche PREEMPT_RT sobre el kernel de Linux.

2.1.1 Parche PREEMPT_RT para Linux

Las tareas de tiempo real normalmente se activan en respuesta a un evento externo o a la expiración de un temporizador. Ante estos casos el kernel de Linux actúa llevando a cabo una serie de acciones:

1. Un manejador reacciona ante la interrupción provocada por un dispositivo o por la expiración de un temporizador.

2. El manejador de interrupciones identifica la tarea que estaba esperando al evento para ser añadida a la cola de tareas listas para ser ejecutadas.

3. El planificador es invocado para que escoja la tarea que debe ser reanudada. Si en la cola de tareas listas se ha insertado una de tiempo real que tiene una prioridad más alta que la tarea que se encontraba en ejecución en el sistema, el planificador seleccionará la nueva tarea.

4. Despacho. El sistema operativo realizará un cambio de contexto para ejecutar la tarea seleccionada.

Tal y como se ha descrito previamente, si la tarea de tiempo real tiene una alta prioridad será seleccionada por el planificador para su ejecución inmediata, aunque esto no siempre ocurre de forma instantánea. En el paso número 1 podríamos encontrarnos con que las interrupciones hayan sido enmascaradas temporalmente por la ejecución de una sección crítica del kernel (*local_irq_disable()*). En los pasos 2-4 se pueden producir retrasos debido a fallos de caché, a contenciones en la memoria o al uso de recursos compartidos. Incluso el paso 3 puede ser también retrasado debido a que una sección crítica del kernel haya deshabilitado la expulsión de tareas (*preempt_disable()*). En el paso 4 se puede producir un vaciado del TLB (*Translation Lookaside Buffer*) que ocasionaría retrasos adicionales. Estas latencias afectan a los tiempos de respuesta de las tareas y pueden ser cruciales para determinar la planificabilidad de las aplicaciones de tiempo real que vayan a ser ejecutadas en el sistema.

Para tratar de mejorar los tiempos de respuesta de Linux y conseguir otorgarle un mayor grado de predictibilidad temporal, en el año 2005 Ingo Molnár inició el desarrollo del parche denominado PREEMPT_RT. El objetivo de este parche es disminuir las latencias y obtener un sistema más predecible, al mismo tiempo que se mantiene el tipo de kernel monólitico de Linux permitiendo a los desarrolladores escribir aplicaciones de tiempo real en el espacio de usuario. Desde que se inicializó el desarrollo del parche PREEMPT_RT numerosas mejoras aportadas en él se han añadido a la línea principal del kernel de Linux. Las mejoras más relevantes adoptadas se ilustran en la Figura 2.2 tal y como han descrito Federico Rehenzani et. al [43].

Desde la versión 2.6 del kernel de Linux tenemos la opción denominada CONFIG_PREEMPT que si es activada antes de la compilación del kernel hace que gran parte del código sea expulsable. Aunque el kernel seguirá siendo no expulsable cuando las interrupciones estén deshabilitadas, algo que ocurre por ejemplo durante la ejecución de los *spinlocks*. Aún seguían quedando

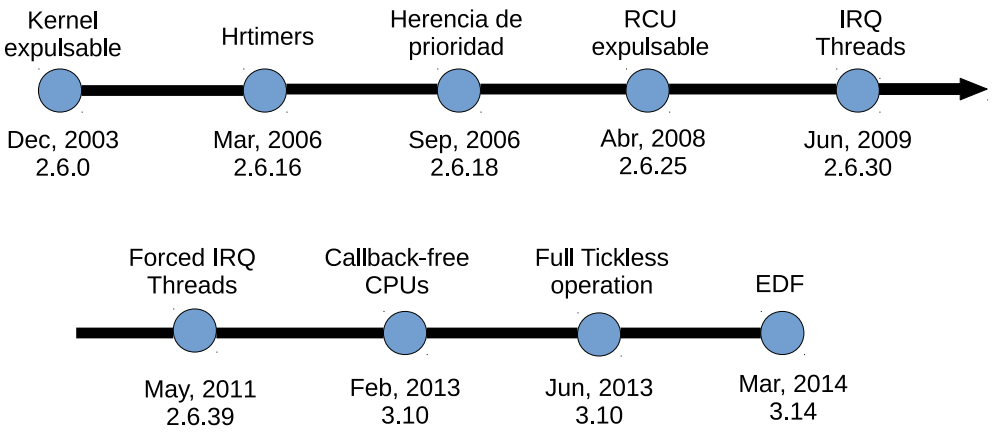

Fig. 2.2: Características añadidas a la línea principal del desarrollo del kernel de Linux provinientes del parche PREEMPT_RT [43].

partes del código no expulsables en la versión principal del kernel de Linux (*vanilla*), lo que provoca que en determinadas situaciones no tengamos tiempos de respuesta acotados para las tareas de tiempo real. Por este motivo uno de los principales objetivos del parche PREEMPT_RT es intentar reducir al máximo las secciones del código que no son expulsables.

Cuando el parche PREEMPT_RT está aplicado y activo con el parámetro de configuración previo a la compilación del kernel denominado CONFIG_PREEMPT_RT_FULL los mecanismos de sincronización *spinlock_t* y *rwlock_t* se convierten en expulsables, mientras que el mecanismo *raw_spinlock_t* mantiene el comportamiento del *spinlock* que se encuentra en la versión *vanilla* del kernel. Además con el parche se utilizan mutexes y semáforos que implementan la herencia de prioridad. Con estos cambios se consigue aumentar de manera significativa el número de zonas de código expulsables del kernel de Linux, dejando solo no expulsables las regiones críticas protegidas con los *spinlocks* del tipo *raw_spinlock_t* y parte de los manejadores de interrupciones.

A pesar de las mejoras que se introducen en términos de predictibilidad temporal al aplicar el parche PREEMPT_RT, es prácticamente imposible conocer con exactitud la respuesta temporal del sistema en todas las situaciones. Los sistemas de propósito general como Linux tienen una complejidad muy alta que imposibilita el análisis estático de todas las piezas del código que lo componen, algo que sí es posible llevar a cabo en la mayoría de sistemas operativos puramente de tiempo real [44]. Prueba de esto es que existe un estudio [45] donde se aplica el parche PREEMPT_RT sobre Linux en dos plataformas hardware distintas (Beaglebone Black y Raspberry Pi 2), y se de-

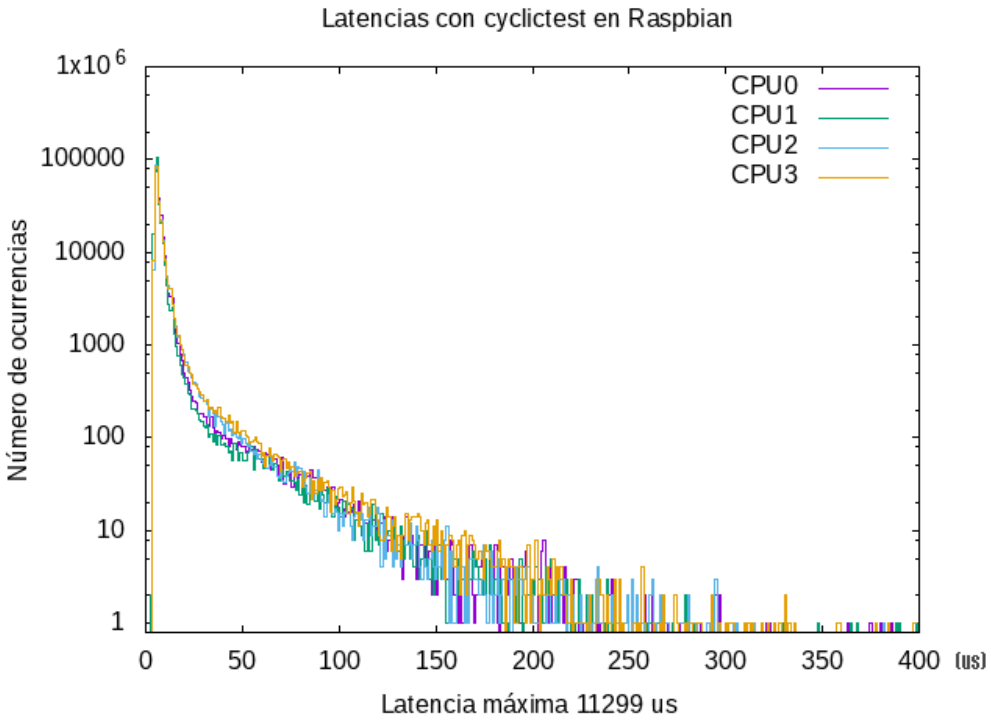

Fig. 2.3: Histograma de latencias obtenidas con la herramienta *Cyclictest* en el sistema operativo Raspberry Pi OS sobre una Raspberry Pi 3 B+.

muestra que a pesar de haber aplicado el parche no se puede tener la certeza de que los plazos temporales se satisfacen en el 100 % de las ocasiones.

Una forma rápida y efectiva de comprobar cómo responde un sistema operativo ante requisitos temporales consiste en medir la latencia que se produce desde que un temporizador expira y el hilo activado por ese temporizador se ejecuta. Con ese objeto hemos utilizado la herramienta denominada *cyclictest*[1] diseñada para medir las latencias en los sistemas operativos Linux de tiempo real a través del uso de temporizadores. Mediante esta herramienta hemos evaluado el comportamiento temporal de Raspberry Pi OS[2] (una distribución de Linux basada en Debian para placas Raspberry Pi) ejecutándose sobre la placa de desarrollo Raspberry Pi 3 B+ con una carga alta en el sistema (reproducción de contenido multimedia en alta definición y descarga de paquetes por la red). Debido a las numerosas interrupciones que se producen en un sistema como Raspberry Pi OS durante 1 minuto de ejecución hemos detectado una gran variabilidad en algunas de las latencias medidas. Incluso en nuestros tests hemos observado una latencia máxima que ha alcanzado los 11299 microsegundos (ver Figura 2.3). Además, a pesar de que la mayoría

[1] `https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/`

[2] `https://www.raspberrypi.org/software/`

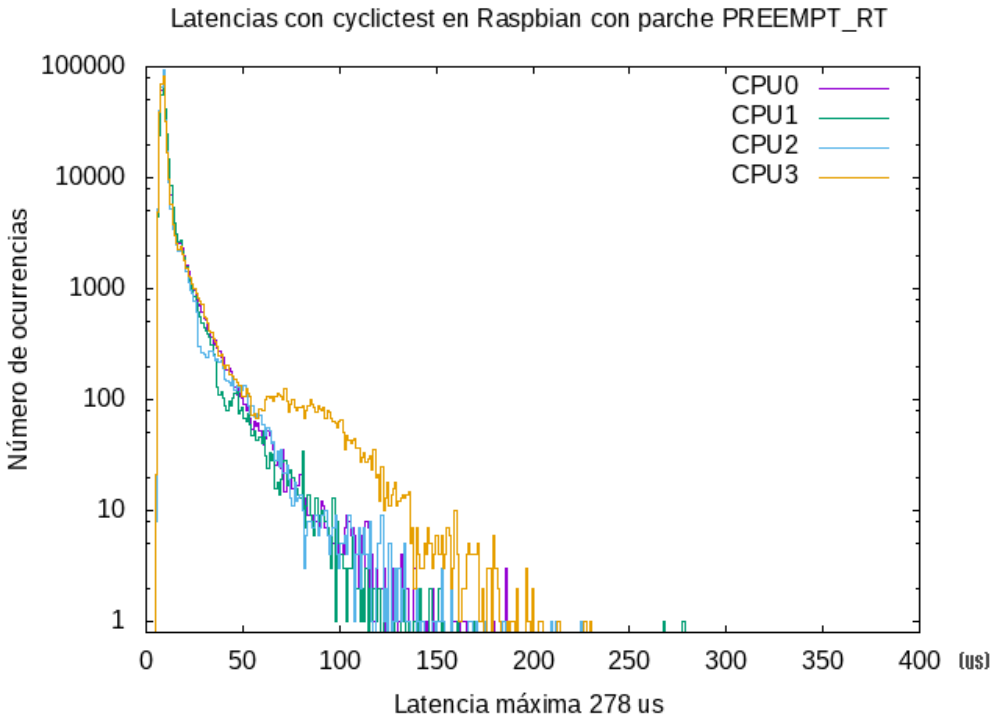

Fig. 2.4: Histograma de latencias obtenidas con la herramienta *cyclictest* en el sistema operativo Raspberry Pi OS con el parche PREEMPT_RT en una Raspberry Pi 3 B+.

de las ejecuciones medidas se encuentran situadas por debajo de los 50 microsegundos no podemos dar una cota temporal razonable para el tiempo de respuesta de peor caso.

Para comparar la mejora que supone aplicar el parche PREEMPT_RT sobre los tiempos de respuestas de peor caso hemos repetido el mismo test anterior después de haber parcheado el kernel de Linux utilizado en Raspberry Pi OS. En la Figura 2.4 se observa cómo todas las latencias se encuentran por debajo de los 278 microsegundos. Por lo tanto podemos afirmar que se produce una mejora notable con respecto al caso previo donde no se había utilizado el parche PREEMPT_RT.

El parche PREEMPT_RT no es la única alternativa existente para tratar de otorgar un comportamiento temporal más determinista a Linux, existen otras soluciones que se basan en la ejecución de un doble núcleo. A continuación procedemos a describir las más relevantes.

2.1.2 Linux con doble núcleo

Algunas soluciones que intentan conseguir un mejor grado de predictibilidad para las tareas de tiempo real basan su arquitectura en un doble núcleo.

En primer lugar se utiliza un pequeño kernel (microkernel/nanokernel) para ejecutar directamente sobre él las tareas con requisitos temporales, y al mismo tiempo ejecutan el kernel estándar de Linux como una tarea de baja prioridad. Esta arquitectura se ilustra en la figura 2.5.

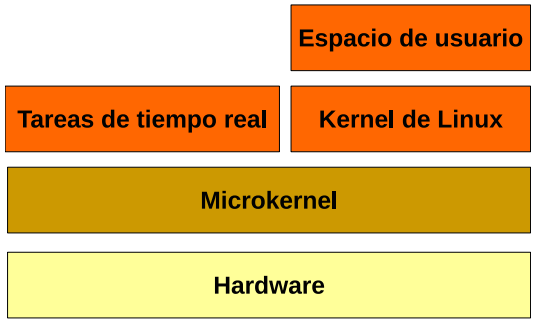

Fig. 2.5: Arquitectura de doble núcleo con microkernel para Linux de tiempo real.

Los proyectos más conocidos que utilizan esta solución son RTLinux, RTAI y Xenomai:

- **RTLinux** [46] está desarrollado por la empresa denominada Finite State Machine Labs, Inc (FSMLabs) [47]. Este sistema operativo utiliza un microkernel para manejar el hardware mientras que Linux se ejecuta como un hilo de baja prioridad. De este modo las tareas de tiempo real y los manejadores de interrupción nunca se ven afectados por operaciones que no tienen caráter de tiempo real. Cuando el microkernel recibe una interrupción comprueba si está relacionada con alguna tarea de tiempo real o no, en caso de que no sea de tiempo real dicha interrupción será marcada y se tramitará posteriormente cuando Linux tenga permiso para ejecutarse. En las primeras versiones de este software, la interacción entre las tareas de tiempo real y el microkernel se realizaba a través de una API reducida que no seguía ninguno de los estándares utilizados en tiempo real. En posteriores versiones se incluyó la compatibilidad con el estándar *POSIX Threads*. Sin embargo, en este sistema se necesitan reescribir los drivers de los dispositivos que vayan a ser usados en las tareas de tiempo real, y esto se puede convertir en una tarea extremadamente ardua debido a la gran complejidad del hardware actual.

- **Real-Time Application Interface (RTAI)** [48] es una implementación basada inicialmente en RTLinux con licencia GNU/GPL. Los desarrolladores de RTAI utilizan un nanokernel denominado en inglés *Adaptive Domain Environment for Operating Systems (ADEOS)*. Con él es posible compartir recursos hardware entre diferentes sistemas operativos o

instancias de un mismo sistema operativo. RTAI, de forma similar a RTLinux, trata al núcleo estándar de Linux como un hilo con la menor prioridad, lo que provoca que su activación se realice cuando no exista ninguna tarea con mayor prioridad en ejecución. Además RTAI posee un módulo denominado LXRT que permite el desarrollo de aplicaciones de tiempo real en el espacio de usuario. Al igual que ocurre con *RTLinux* existe el inconveniente de que se necesitan reescribir los drivers de los dispositivos que se vayan a utilizar.

- **Xenomai** [49] sigue un planteamiento arquitectural similar al usado en RTAI. Xenomai añade un subsistema debajo de Linux, de tal modo que en la capa más inferior del software nos topamos con ADEOS (I-pipe), el cual está encargado de recibir las interrupciones hardware. ADEOS/I-Pipe pasa los eventos a sus clientes software siguiendo un orden de prioridad (el software de Xenomai tiene más prioridad que el kernel de Linux estándar). Linux recibe solo los eventos virtuales de interrupción y responderá cuando el software de más alta prioridad (Xenomai) haya tenido la oportunidad de atender las interrupciones. Con la capa ADEOS/I-Pipe se consigue prevenir que el kernel de Linux expulse a las tareas de tiempo real durante su ejecución.

Aunque conceptualmente RTAI y Xenomai son prácticamente iguales, hay alguna diferencia sustancial en el tratamiento de las interrupciones, RTAI puede atender las interrupciones no solo usando ADEOS sino interceptándo-las directamente tal y como se muestra en la Figura 2.6a. En cambio Xenomai maneja todas las interrupciones utilizando ADEOS (ver Figura2.6b). Por otro lado RTLinux y RTAI tienen el inconveniente de que las tareas de tiempo real se ejecutan al mismo nivel que el kernel estándar de Linux y por lo tanto no hay protección de memoria entre ellos.

Este tipo de soluciones presentan una importante desventaja frente a la utilización directa de un kernel de Linux estándar o modificado con el parche PREEMPT_RT, ya que se pierde la posibilidad de utilizar la gran cantidad de drivers y librerías ya desarrollados para Linux. Además, el uso de un doble núcleo requiere de una profunda modificación a nivel de código del kernel, lo que dificulta considerablemente la mantenibilidad y la portabilidad a través de las distintas versiones. Incluso algunos trabajos [50] afirman que este tipo de implementaciones son menos estables y seguras debido a la complejidad en las interacciones entre los dos kernels, lo que se termina traduciendo en un esfuerzo extra para los desarrolladores.

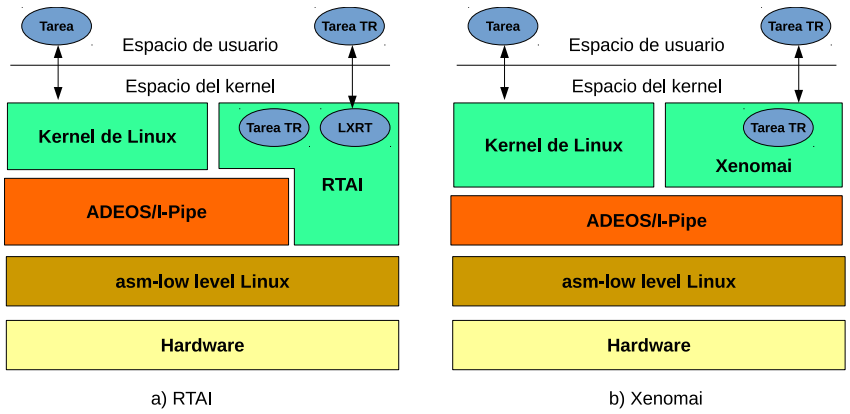

Fig. 2.6: Arquitecturas de RTAI y Xenomai.

2.2 Soluciones de tiempo real para Android

Desde que en 2010 se publicara el trabajo donde se proponen cuatro posibles modificaciones en la plataforma Android para conseguir tener un comportamiento temporal más predecible [36] (ver Figura 2.1), han surgido diferentes desarrollos basados en cierta medida en dicho trabajo.

Maurer *et al.* [39] realizaron un desarrollo en el que proponían tener una solución basada en la Figura 2.1a donde las aplicaciones con requisitos temporales se ejecutan directamente sobre el kernel de Linux/Android con el parche PREEMPT_RT aplicado. Las aplicaciones de tiempo real se escribirán normalmente en lenguaje C para ejecutarse directamente sobre el kernel del sistema. Además estas aplicaciones podrían intercambiar datos con el resto de aplicaciones del sistema gracias a un mecanismo de comunicación no bloqueante. Desafortunadamente este trabajo en la actualidad no sigue en desarrollo y su última publicación expone sus experimentos sobre Android 3.1 con un kernel de Linux basado en la versión 2.6.36 [3].

Existen otras soluciones desarrolladas durante los últimos años en el ámbito universitario que se han constituido como una línea de investigación con cierta relevancia: RTAndroid y RTDroid, las cuales serán descritas a continuación. Además en el ámbito empresarial existen algunas compañías que han tratado de ofrecer algunos productos compatibles con Android y aplicaciones de tiempo real, en la Subsección 2.3 hablaremos de ellas.

[3] La última versión estable de Android en octubre de 2020 es Android 11

2.2.1 RTAndroid

La solución denominada RTAndroid [51] trata de incorporar características de tiempo real a la plataforma Android a través de la extensión y modificación de algunos de sus componentes (ver Figura 2.7).

En primer lugar se aplica el parche de tiempo real PREEMPT_RT [42] sobre el kernel de Linux que los dispositivos Android incorporan. Tal y como afirman los propios autores de esta solución, la aplicación del parche no se puede realizar de manera directa. Requiere de un proceso de adaptación en función de la versión del kernel utilizada y del hardware usado en el dispositivo. Para conseguir un mayor grado de predictibilidad se controla el cambio de frecuencia en el procesador para así evitar los cambios dinámicos que puedan ocasionar tiempos de respuestas no acotados. Por otro lado el recolector de basura de la máquina virtual Java de Android (Dalvik/ART) se ha modificado para que pueda ser ejecutado concurrentemente y no provoque la suspensión de las aplicaciones de tiempo real. A nivel de la capa denominada framework de aplicaciones se ha realizado una modificación sobre el componente *Binder*, el cual es clave para la intercomunicación de las aplicaciones que se ejecutan en el sistema. La nueva arquitectura desarrollada en esta solución para el *Binder* proporciona herencia de prioridad en las llamadas a procedimientos remotos. Por último se ha llevado a cabo una adaptación del mecanismo más importante en Android (*Intent*) para la comunicación entre aplicaciones. En él se utiliza una estructura de datos que ordena por prioridades los hilos de las aplicaciones de tiempo real que se comunican a través de los *Intents*. De este modo el sistema puede priorizar y manejar los *Intents* concurrentes en función de su prioridad.

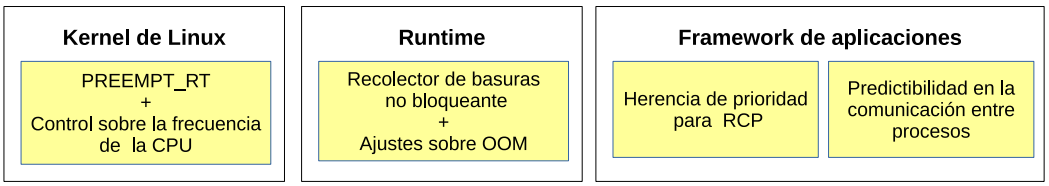

Fig. 2.7: Extensiones realizadas en la plataforma Android para añadir soporte de tiempo real en la solución denominada RTAndroid.

Con cada nueva versión de Android que se libere y que realice cambios que afecten a las modificacciones llevadas a cabo en RTAndroid se necesitará realizar una verificación y adaptación de la solución. Por lo tanto, podemos afirmar que no es fácilmente mantenible a lo largo del tiempo.

2.2.2 RTDroid

RTDroid [52] se presenta como una solución desarrollada en la Universidad de Búfalo para obtener un sistema con garantías temporales compatible con el modelo de programación presente en Android. En esta solución, se utiliza un kernel de tiempo real, la máquina virtual Java de Android (Dalvik/ART) se reemplaza por una con características de tiempo real, y además se han rediseñado algunos componentes claves del framework de Android. Para llevar a cabo todo lo citado anteriormente se realiza un rediseño de Android como el mostrado en la Figura 2.8 donde nos topamos en la capa más inferior con un kernel que posee características de tiempo real (el kernel de Linux con el parche PREEMPT_RT [42] o RTEMS [53]). Inmediatamente encima se situa una máquina virtual Java de tiempo real (Fiji VM) y la librería Bionic extendida con las funciones necesarias para el desarrollo de programas de tiempo real. Además, con la capa denominada framework de aplicaciones se trata de mantener la compatibilidad con el modelo de programación utilizado en Android. En ella se han recreado las construcciones para el paso de mensajes entre distintos hilos de ejecución (*RT Looper* y *RT Handler*) con características de tiempo real. También se ha rediseñado el comportamiento del servicio del sistema encargado de programar alarmas en el sistema para que se ejecute una acción específica (*RT AlarmManager*) y el servicio que gestiona los datos de los sensores físicos y virtuales disponibles en los dispositivos Android (*RT SensorManager*).

Los autores de esta solución la clasifican en tres perfiles distintos en función de las garantías temporales que ofrezca la plataforma donde se vaya a ejecutar:

- **Tiempo real laxo en dispositivo móvil:** Este perfil es el que menos garantías temporales proporciona debido a que no es posible aplicar de manera inmediata los parches de Linux de tiempo real para el kernel que se utiliza en los dispositivos en Android. Este sistema operativo introduce un número significativo de cambios en el kernel de Linux que impiden la aplicación directa del parche sin realizar una modificación del mismo de manera manual. A pesar de esto los autores de esta solución afirman que sin la utilización de un kernel de tiempo real es posible ejecutar aplicaciones con garantías de tiempo real laxo.

- **Tiempo real laxo en ordenador de escritorio:** Con este perfil se consiguen unas garantías mayores que en el caso de los dispositivos móviles puesto que es posible aplicar directamente los parches de tiempo real

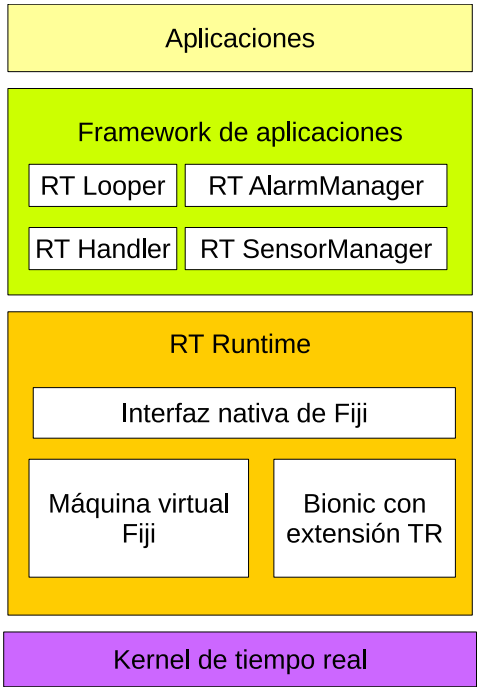

Fig. 2.8: Arquitectura adoptada por la solución RTDroid.

sobre el kernel de Linux. Aun así Linux con el parche de tiempo real no es utilizado habitualmente para entornos de tiempo real estricto ya que la ejecución de código por parte de algunas secciones del kernel pueden provocar en ocasiones puntuales pérdidas de plazos en las aplicaciones con requisitos temporales.

- **Tiempo real estricto en dispositivo embebido:** En este caso se utiliza un sistema operativo de tiempo real como es RTEMS [53]. Además se utiliza una placa de desarrollo con drivers específicamente diseñada para tiempo real. De este modo los autores de esta solución proporcionan garantías temporales mucho más estrictas.

Esta solución no está disponible de forma directa para cualquier dispositivo compatible con Android. Los desarrolladores deben ofrecernos el código compatible con el hardware que se vaya a utilizar. Además, mantener la compatibilidad con las nuevas versiones de Android requiere de un constante proceso de adaptación y verificación.

2.3 Soluciones empresariales

En el ámbito empresarial podemos encontrar algunas soluciones que permiten tener un entorno donde conviven simultáneamente aplicaciones con requisitos temporales y la plataforma Android.

La empresa llamada eSOL basa su solución [54] en la sustitución del kernel de Linux que posee Android por uno específicamente desarrollado por ellos mismos para sistemas de tiempo real. Para que todos los drivers desarrollados para Android puedan seguir siendo utilizados sobre el nuevo kernel incorporan un adaptador que hace la función de middleware entre la plataforma Android y el nuevo kernel de tiempo real (ver Figura 2.9). Esta solución, aparte de su carácter privativo y no abierto, implica la modificación del adaptador cada vez que los drivers originales de Android tengan un cambio significativo.

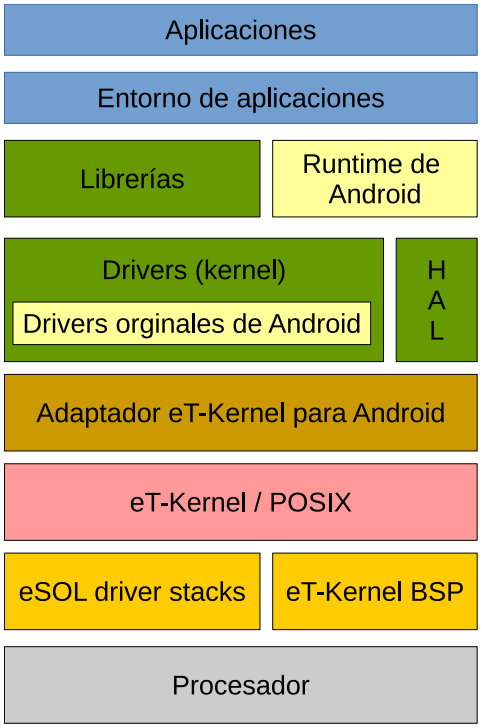

Fig. 2.9: Arquitectura utilizada por la empresa eSOL para obtener una mejor predictibilidad temporal en la plataforma Android.

Por otro lado, XtratuM [29] es un hipervisor desarrollado inicialmente por el Instituto de Automática e Informática Industrial de la Universidad Politécnica de Valencia aunque en la actualidad es gestionado por la empresa tecnológica denominada fentISS[4]. Este hipervisor permite tener en un mismo hardware diferentes sistemas operativos ejecutándose concurrentemente. El diseño adoptado para XtratuM hace que él sea el único encargado de gestionar todo el hardware de la plataforma utilizada. Para que los distintos sistemas operativos que estén ejecutándose sobre XtratuM puedan tener acceso al hardware se proporciona una interfaz de acceso. Esto implica que

[4] `https://fentiss.com/`

dichos sistemas operativos tienen que ser adaptados y modificados para utilizar la interfaz de acceso.

Debido a las posibilidades que ofrece XtratuM, un trabajo previo [55] buscaba obtener una plataforma domótica donde coexistiese el sistema Android con otros sistemas operativos de tiempo real donde se puedan ejecutar las aplicaciones con requisitos temporales. De este modo se aprovechan las características gráficas de la interfaz de usuario de Android para que se puedan manejar los componentes integrados en el sistema domótico. Para conseguir tener Android funcionando sobre XtratuM se utilizó un porting de Android para la arquitectura x86 [56], y éste a su vez se adaptó y modificó para que funcionase con la interfaz de acceso que proporciona XtratuM. A pesar de que existe un parche para que Linux soporte el módulo de XtratuM su aplicación no es directa, ya que el kernel utilizado en Android presenta algunas diferencias respecto al kernel estándar de Linux. Esto demuestra que si se optase por una solución que utilice un hipervisor, para cada nueva versión de Android que modifique el kernel se debería realizar un proceso de adaptación, lo que se traduce en una difícil mantenibilidad a lo largo del tiempo.

2.4 Caracterización de las soluciones disponibles

Durante este capítulo se han descrito las soluciones que permiten, en mayor o menor medida, mejorar la predictibilidad temporal del sistema operativo Android. Cada una de ellas posee unas características que la hacen diferir de las demás. Por lo tanto, en esta sección se va a proceder a describir cuáles son las características más relevantes que pueden tener estas soluciones y cuáles son satisfechas en cada una de ellas.

Hemos identificado cinco características que pueden ser satisfechas o no por las distintas soluciones y que procedemos a describir a continuación:

- **Grado de predictibilidad temporal:** Cuando desarrollamos entornos de tiempo real estos nos pueden ofrecer diferentes grados de predictibiliad temporal en función de su capacidad para cumplir con los plazos temporales requeridos. Si el sistema es totalmente predecible, tanto por el software utilizado como por el hardware, y se puede certificar que todos los plazos temporales se satisfacen diremos que se trata de un entorno de *tiempo real estricto*. Si los plazos temporales no se cumplen

en la totalidad de los casos pero sí en la mayoría diremos que tenemos un entorno de *tiempo real laxo*.

- **Modificación del kernel de Android/Linux:** Algunas soluciones pueden necesitar modificar y adaptar parte del código del kernel basado en Linux utilizado por Android, veáse el caso del parche PREEMPT_RT que no es aplicable de forma directa.

- **Compatibilidad con el framework de Android:** Las aplicaciones que hacen uso del entorno ofrecido por Android normalmente están escritas en Java y ejecutadas sobre la máquina virtual Java propia de este sistema (*Android RunTime, ART*). Para obtener una solución integral donde las aplicaciones con requisitos temporales pueden ser desplegadas directamente como aplicaciones Java será necesario realizar modificaciones que suplan todas las carencias de tiempo real de la máquina virtual Java y del entorno donde se encuentran las funciones y librerías que nos ofrece Android.

- **Complejidad de mantenimiento:** Android es un sistema en constante evolución. Cada nueva versión implica numerosos cambios en cada una de las capas software que conforman el sistema. Cuando las soluciones existentes para conseguir tiempo real realizan adaptaciones o modificaciones sobre las capas software de Android, en cada nueva versión se debe verificar si estas adaptaciones siguen siendo compatibles o no y en caso de no serlo realizar las modificaciones pertinentes. Por lo tanto, vamos a clasificar en tres tipos esta característica: (1) *alta*: cuando varias capas software deben ser modificadas, (2) *media:* cuando solo la capa perteneciente al kernel debe ser mantenida, y (3) *baja:* si ninguna capa software requiere cambios importantes en cada nueva versión.

- **Portabilidad entre versiones:** Cuando las soluciones que consiguen mayor predictibilidad temporal realizan modificaciones profundas en componentes claves como la máquina virtual Java o el kernel se produce una fuerte dependencia con la versión de Android utilizada. Esto provoca que se necesiten realizar adaptaciones para cada versión de Android en la que se quiere utilizar la solución.

En la Tabla 2.1 se ilustran qué características de las descritas anteriormente satisfacen las soluciones que se han presentado durante el presente capítulo. En todas ellas queda reflejado que no son fácilmente portables a otras versiones y, además, requieren de un gran mantenimiento para adaptarse

	Maurer et al.[39]	RTAndroid[51]	RTDroid[52]	eSOL[54]	XtratuM + Android[29]
Grado de predictibilidad temporal	Estricto[5]	Estricto[5]	Estricto	Estricto	Estricto[6]
Modificación del kernel	Sí	Sí	Sí	Sí	Sí
Compatibilidad con el framework de Android	No	Sí	Sí	Sí	Sí
Complejidad de mantenimiento	Media	Alta	Alta	Alta	Media
Portabilidad entre versiones	No	No	No	No	No

Tabla 2.1: Tabla comparativa con las distintas soluciones disponibles para añadir características de tiempo real a la plataforma Android.

a las nuevas versiones de Android que van apareciendo. Por estos motivos en el siguiente capítulo del presente trabajo presentamos una solución que permite superar estas limitaciones en el sistema operativo Android.

[5]La solución utiliza el kernel de Linux/Android con el parche PREEMPT_RT. Esto nos proporciona un mayor grado de predictibilidad temporal aunque no podríamos afirmar que en el 100 % de los casos se cumplan los plazos temporales. La complejidad tan alta que posee un sistema de propósito general como Android imposibilita en la práctica el análisis estático temporal [44].

[6]Cuando se utiliza el hipervisor XtratuM con Android, este sistema operativo no es el destinado a ejecutar aplicaciones de tiempo real.

3 Aislamiento de procesadores multinúcleo en Linux/Android

> *Todo es muy difícil antes de ser sencillo.*
>
> **— Thomas Fuller**
> (Eclesiástico e historiador)

Añadir características de tiempo real al sistema operativo Android requiere de un estudio profundo del kernel de Linux. Este kernel es un software con una extrema complejidad que en los últimos años ha crecido de una forma muy significativa, lo que se puede cuantificar a través de la evolución del número de líneas de código fuente que lo conforman. En la Figura 3.1 se ilustra cómo desde la versión 2.6.20 con algo más de 8 millones de líneas de código se ha incrementado este número hasta superar los 25 millones para la versión 4.14 [1].

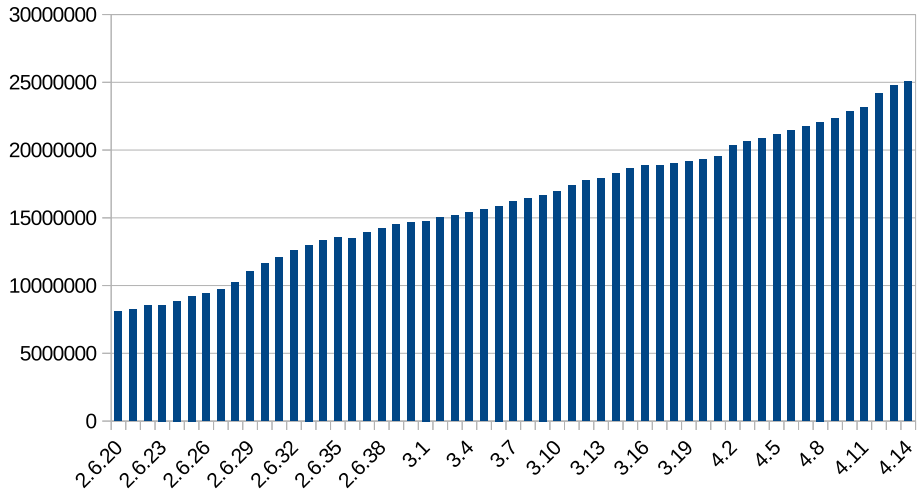

Fig. 3.1: Evolución del número de líneas de código del kernel de Linux desde la versión 2.6.20 hasta la 4.14

Como se ha descrito en el Capítulo 2 existen algunas implementaciones para mejorar la predictibilidad temporal del kernel de Linux y de Android. Todas estas soluciones tienen en común la necesidad de modificar o adaptar parte del código del kernel, e incluso algunas de ellas necesitan modificar la máquina virtual de Java o las librerías específicas utilizadas en Android. Todo

[1] `https://commons.wikimedia.org/wiki/File:Lines_of_Code_Linux_Kernel.svg`

esto provoca que sea muy complejo mantener las soluciones actualizadas y disponibles para las últimas versiones del sistema operativo Android. Llevar a cabo modificaciones en un kernel de Linux como el que utiliza Android no es una tarea trivial. Además, este sistema operativo realiza una adaptación del kernel de Linux para dispositivos móviles, lo que impide que se puedan aplicar de forma directa soluciones como el parche PREEMPT_RT (ver Subsección 2.1.1). Por ejemplo, si se quiere utilizar este parche sobre el kernel basado en la versión 4.9 para el dispositivo Pixel 3 de Google aparecen diferencias en 35 archivos del código fuente que hacen imposible la aplicación directa de dicho parche.

El presente capítulo de esta tesis tiene como objetivo desarrollar y mostrar una solución que permita mejorar la predictibilidad temporal de los sistemas basados en Linux (como es el caso de Android) sin necesidad de realizar cambios profundos. De este modo se consigue una solución portable y aplicable a cualquier dispositivo Android actual. Para poder alcanzar dicha solución sacaremos partido de las arquitecturas multinúcleo que tan comunes son hoy en día. La solución que se describe en este capítulo se llevará a cabo siguiendo las siguientes propuestas:

- Aislamiento de los núcleos del procesador en los que ejecutar las aplicaciones de tiempo real libres de las interferencias de las demás aplicaciones y de la mayor parte de las actividades del sistema operativo (ver Sección 3.5).
- Aplicación en el sistema de una serie de ajustes para mejorar la predictibilidad temporal (ver Sección 3.6).
- Las aplicaciones de tiempo real serán aplicaciones nativas que se ejecutarán sobre el kernel y las librerías nativas (ver Subsección 3.7.1).

Para el desarrollo de este capítulo comenzaremos con la Sección 3.1 en la que describiremos las arquitecturas multinúcleo utilizadas en la actualidad en los dispositivos Android. A continuación, en la Sección 3.2 se resumirán cuáles son las políticas de planificabilidad que pueden ser adoptadas en los procesadores multinúcleo, las cuales son utilizadas para distribuir los diferentes hilos o procesos en los núcleos disponibles en el sistema. Posteriormente, en la Sección 3.3 analizaremos cómo funciona el planificador utilizado en el kernel de los sistemas Linux/Android. En la Sección 3.4 se expone nuestra solución para ejecutar aplicaciones de tiempo real en Android. Después, en la Sección 3.5 se estudian cuáles son los mecanismos de aislamiento más idóneos disponibles en Android/Linux para aislar núcleos del procesador.

Durante el transcurso de la Sección 3.6 se describen los ajustes generales que deben ser aplicados en el sistema para mejorar la respuesta temporal de las aplicaciones de tiempo real. La Sección 3.7 muestra todas las pruebas llevadas a cabo para evaluar la viabilidad de la solución presentada en este capítulo. Finalmente, en la Sección 3.8 se desarrollan las conclusiones.

3.1 Arquitecturas multinúcleo en Android

Las arquitecturas multinúcleo son la tendencia actual para continuar con la mejora de rendimiento en los procesadores de propósito general, sistemas embebidos o computadores de alto rendimiento [57]. Un procesador multinúcleo es la combinación de dos o más núcleos en un solo chip. Estos núcleos están conectados a una memoria común a todos ellos a través de un bus compartido. Además, los núcleos normalmente tendrán una caché L1 independiente y una caché L2 compartida entre todos ellos dentro del chip.

Los procesadores multinúcleo que se comercializan en el ámbito de la electrónica de consumo están optimizados para mejorar el rendimiento medio y no los tiempos de peor caso. La utilización de la memoria caché y de buses compartidos en dichos procesadores, introduce cierta incertidumbre que aumenta la complejidad del análisis temporal de los sistemas [58] [59]. A pesar de esto, el uso de arquitecturas multinúcleo en entornos de tiempo real proporciona algunas ventajas:

- **La reducción del número de procesadores en los sistemas:** Los procesadores multinúcleo tienen un consumo energético menor en comparación con aquellos que solo tienen un único núcleo funcionando a frecuencias más altas y, por lo tanto, generan menos calor que será más fácil de disipar. Por otro lado, tener menos procesadores implica sistemas más pequeños y menos pesados.

- **El aumento de la capacidad de computo:** Las aplicaciones actuales cada vez son más complejas y requieren de más recursos para cumplir con sus requisitos. Al tener varias unidades computacionales disponibles las aplicaciones pueden dividirse en diferentes tareas para ejecutarse en paralelo y mejorar el rendimiento.

Los dispositivos móviles Android buscan mejorar su rendimiento manteniendo un consumo energético bajo. A estos efectos los procesadores multinúcleo se han convertido en algo habitual en teléfonos móviles o tabletas. Los fabricantes de procesadores más relevantes han optado por presentar en los últimos años modelos con arquitecturas multinúcleo heterogéneas. Algunos

de ellos son los chips Snapdragon[2] de Qualcomm o los Exynos[3] de Samsung. Este tipo de chips basan su arquitectura en una denominada *ARM big.LITTLE* [60]. En ella se emplazan grupos de núcleo de procesadores lentos (*LITTLE*) y núcleos más potentes (*big*). Esta arquitectura desarrollada por la empresa ARM busca poder atender dinámicamente las necesidades computacionales de las aplicaciones, al mismo tiempo que reduce el consumo energético. Para ello, las aplicaciones pueden migrar entre distintos grupos de procesadores en función de sus requisitos de computo. Este tipo de arquitecturas ya son utilizadas en la actualidad en entornos como el de la automoción; un ejemplo de ello es la plataforma Renesas R-Car H3 [61].

El grueso de los dispositivos Android utilizan procesadores ARM. Existen numerosos procesadores basados en ARM orientados a distintos ámbitos, y por eso se ha diversificado el porfolio de productos disponibles en el catálogo de ARM con la microarquitectura de los núcleos denominada *Cortex*. Dentro de ella nos topamos con tres perfiles diferentes:

- **Cortex-A:** Diseñada para ser utilizada con aplicaciones complejas sobre sistemas operativos. Posee opciones de coma flotante, extensiones multimedia y unidad de gestión de memoria (*MMU*, *Memory Management Unit* en inglés).

- **Cortex-R:** Orientada principalmente para aplicaciones críticas de tiempo real. Es muy similar al perfil Cortex-A aunque añade algunas características para tiempo real y seguridad crítica que hacen que la ejecución sea más determinista. Algunas de estas características son la unidad de protección de memoria (*MPU*, *Memory Protection Unit* en inglés) o el uso de memorias RAM con paridad.

- **Cortex-M:** Diseñada para su uso en microcontroladores. Su objetivo principal es el bajo consumo y la reducción de costes.

Los procesadores ARM utilizados en la actualidad en los dispositivos Android basan sus núcleos en la familia Cortex-A. A pesar de que estos núcleos no fueron diseñados inicialmente para cumplir con requisitos de tiempo real, algunos trabajos [62] [63] han demostrado que procesadores basados en la arquitectura *ARM big.LITTLE* con núcleos de este tipo pueden ser utilizados en entornos con requisitos temporales.

Los procesadores multinúcleo suponen un reto para los sistemas de tiempo real, ya que el software existente necesita realizar ajustes para adaptarse y

[2] `https://www.qualcomm.com/snapdragon`
[3] https://`www.samsung.com/semiconductor/minisite/exynos/`

aprovechar todos los núcleos disponibles. Estos sistemas deben decidir en qué núcleos ejecutará cada una de las tareas en las que se dividen los programas. Además, a medida que las arquitecturas multinúcleo están siendo adoptadas por la industria, se intenta integrar en un único sistema la máxima funcionalidad posible. Por lo tanto, en sistemas operativos basados en Linux utilizados en entornos de tiempo real resulta habitual encontrarse con aplicaciones que tengan distintos requisitos no funcionales relacionados con la respuesta temporal. Por este motivo, existen diferentes técnicas de planificación para asignar la ejecución de las tareas en los distintos núcleos disponibles. En la siguiente sección procedemos a describirlas.

3.2 Planificación en procesadores multinúcleo

Cuando tenemos un sistema con diversas tareas (procesos y/o hilos) y distintos núcleos en un procesador donde dichas tareas pueden ser ejecutadas, nos encontraremos principalmente con tres tipos de algoritmos de planificación para la asignación de tareas a los núcleos [64]:

- **Planificación particionada:** Las tareas son asignadas estáticamente a los procesadores (núcleos). Además, las tareas serán planificadas únicamente en el procesador en el que han sido asignadas, lo que implica que no habrá migraciones a otros núcleos.

- **Planificación global:** Las tareas son planificadas utilizando un único planificador global permitiendo que se puedan ejecutar en cualquier núcleo. En esta clase de algoritmos existe una única cola global para almacenar las tareas pendientes de ejecución. En cualquier momento dado las tareas con mayor prioridad almacenadas en la cola pueden ser seleccionadas para ejecutarse en alguno de los núcleos del procesador y, por ende, las tareas en ejecución pueden ser expulsadas o migradas de un núcleo a otro.

- **Planificación híbrida:** Algunos sistemas no pueden ser planificados utilizando únicamente algoritmos de planificación particionada o global. Dependiendo de la arquitectura hardware usada, una planificación global podría incurrir en un sobrecoste muy alto debido a la migración de las tareas entre los procesadores. Por otro lado, una planificación particionada puede ocasionar un uso muy fragmentado de los núcleos reduciendo la utilización del sistema considerablemente. Por estos

motivos existen soluciones que se basan en una mezcla de los métodos usados en los algoritmos de planificación global y particionada.

Los sistemas operativos basados en Linux utilizan un planificador global que puede asignar la ejecución de las tareas a cualquier núcleo del procesador. En la siguiente subsección vamos a describir cómo funciona el planificador implementado en el kernel de Linux.

3.3 Planificador del kernel de Linux/Android

Con el surgimiento de los sistemas operativos multitarea apareció la necesidad de tener algoritmos de planificación. La función de estos algoritmos es repartir el tiempo disponible del procesador entre todas las tareas (procesos y/o hilos) que estén listas para ser ejecutadas. Con esto se consigue que los sistemas operativos den al usuario la ilusión de que todas las aplicaciones en uso en un momento dado se están ejecutando de manera simultánea. En sistemas con un solo procesador realmente solo un hilo se estará ejecutando en un instante temporal dado. Sin embargo, cuando en el hardware tenemos un procesador multinúcleo los hilos se podrán ejecutar en paralelo.

El planificador es la pieza software encargada de escoger cuáles serán las siguientes tareas que serán ejecutadas por el procesador. Como los sistemas operativos basados en Linux son de propósito general buscan que todas las tareas puedan ir progresando mediante su ejecución. Para que esto pueda ocurrir Linux implementa un modelo multitarea expulsable. Esto significa que el planificador expulsará la tarea que se encuentra en ejecución cuando otra de mayor prioridad se active. En Linux, el planificador tomará estas decisiones en función de un temporizador periódico que expira y genera una interrupción entre 100 y 1000 veces cada segundo. Cuando la interrupción se produce, el planificador decide si otorga el uso de los núcleos del procesador a alguna otra tarea.

Las tareas se pueden clasificar en dos tipos: interactivas (*I/O bound*) y no interactivas (*CPU-bound*). Las tareas interactivas son dependientes de operaciones de entrada/salida (por ejemplo, escribir en disco). En cambio aquellas definidas como no interactivas necesitan utilizar de forma intensiva el procesador para realizar cálculos. El planificador tiene que balancear el uso del procesador entre estos dos tipos de tareas y asegurarse de que todas ellas tienen los suficientes recursos de ejecución. Para maximizar la utilización del procesador y garantizar unos tiempos de respuesta rápidos, Linux intenta

otorgar a las tareas no interactivas intervalos de ejecución largos, aunque se ejecutarán de manera menos frecuente. En contraposición, las tareas interactivas se ejecutarán durante menos tiempo en el procesador pero de manera más frecuente.

Aparte de la clasificación previa existe otro criterio que utiliza el planificador de Linux para categorizar las tareas. En este caso las tareas pueden ser de tiempo real o no y, por consiguiente, el planificador utiliza dos algoritmos distintos:

- **Planificador completamente justo** (*Completely Fair Scheduler*, en inglés): Este algoritmo de planificación tiene como objetivo maximizar el uso del procesador; para ello emplea el concepto de *tiempo de ejecución virtual* (*virtual runtime*, en inglés). Cuando las tareas se crean, se registra el momento exacto y se empieza a contabilizar la cantidad de tiempo de espera y de ejecución que ha transcurrido desde ese momento inicial. El tiempo correspondiente a la espera de una tarea es aquel que debería haber estado ejecutando si no existiesen otras tareas en el sistema, mientras que el tiempo de la tarea en ejecución en el procesador se decrementa del tiempo de espera anterior. El tiempo resultante se divide entre el número de tareas que hay en el sistema y se hace una pequeña ponderación a través de la prioridad (entre -20 y 19) que tenga cada una de las tareas. Dicha prioridad puede ser modificada en los sistemas Linux a través del programa *nice*, y precisamente por este motivo a estas prioridades se las conoce como valores *nice*.

  Cuando el planificador necesite seleccionar la siguiente tarea a ejecutar realizará la búsqueda a través de un árbol binario rojo-negro [65] [66] ordenado con las tareas en función de su tiempo de ejecución virtual. La razón de utilizar una estructura de datos como la del árbol binario rojo-negro reside en que buscar, insertar o borrar nodos tiene tiempos de complejidad $O(\log n)$, donde n es el número de tareas. El planificador completamente justo asigna a cada tarea una ponderación basada en su prioridad, de tal modo que durante su ejecución la tarea con menor prioridad tendrá su tiempo de ejecución más limitado que una tarea con mayor prioridad. Esto se traduce en que las tareas de menor prioridad tendrán menos tiempo de ejecución en el procesador comparadas con las tareas de mayor prioridad.

  Con este algoritmo de planificación se consigue que todas las tareas tengan fracciones temporales dinámicas para su ejecución. Así se intenta ser lo más justo posible con todas ellas.

- **Planificador de tiempo real:** Los sistemas operativos basados en Linux proporcionan al menos dos políticas de planificación de tiempo real: SCHED_FIFO y SCHED_RR. A partir de la versión 3.14 del kernel de Linux también está disponible SCHED_DEADLINE. La política de planificación denominada SCHED_FIFO implementa un algoritmo donde las tareas son seleccionadas en función de su prioridad. Dentro de un mismo nivel de prioridad, las tareas se elegirán en función de su orden de llegada, es decir que la primera tarea en entrar será la primera en ser planificada. Una tarea planificada con la política SCHED_FIFO comienza su ejecución y continuará hasta que voluntariamente deje el procesador, se bloquee o sea expulsada por otra tarea de tiempo real con mayor prioridad. La política SCHED_RR posee un funcionamiento similar al descrito anteriormente para SCHED_FIFO, excepto que cada tarea puede ejecutar durante porciones temporales. Cuando una tarea exceda su porción temporal se colocará al final de la lista de tareas con su misma prioridad pendientes de ejecutar, con lo que se consigue una rotación cíclica entre tareas de la misma prioridad.

  La política SCHED_DEADLINE se basa en prioridades dinámicas y en la utilización del plazo de respuesta absoluto de las tareas como la base para tomar las decisiones de planificación. El plazo de respuesta absoluto de cada tarea se calcula como la hora de activación más el plazo relativo fijo asignado a cada una de estas tareas. Aquellas tareas con el plazo más cercano serán seleccionadas antes para la ejecución que otra tarea con plazo más lejano. Además, esta política incorpora el algoritmo de reserva de ancho de banda denominado *Constant Bandwidth Server* (CBS) [67], el cual permite asegurar que las tareas serán planificadas por un tiempo máximo igual a su capacidad en cada ventana temporal de duración igual a su periodo.

  Como las tareas que utilizan políticas de tiempo real podrían consumir todo el tiempo de ejecución del procesador y, por ende, bloquear el sistema operativo, por defecto los sistemas Linux tienen un mecanismo con el cual podemos indicar un porcentaje de utilización máximo del procesador para este tipo de tareas. En Linux lo habitual es encontrarnos con una configuración inicial donde las tareas con planificación de tiempo real utilizan como máximo un 95 % de tiempo de uso del procesador; sin embargo el sistema operativo permite configurar este valor[4].

[4]Se puede modificar a través de estos dos ficheros /proc/sys/kernel/sched_rt_period_us y /proc/sys/kernel/sched_rt_runtime_us

Cuando el hardware posee un procesador multinúcleo, el planificador tiene a su disposición varias unidades computacionales para lanzar en ellas las tareas pendientes de ejecución. Debido a esto, por cada núcleo existe una estructura de datos, denominada en inglés *runqueue*, donde se mantienen las tareas que están listas para ejecutar. Las tareas son asignadas al menos a una *runqueue* y el planificador utiliza un algoritmo de balanceo de carga para tratar de conseguir que las tareas se repartan de la manera más eficiente entre los distintos núcleos.

En la siguiente sección se describirá cuál es la solución buscada para aprovecharnos de las arquitecturas multinúcleos actuales, y así conseguir mejorar la respuesta temporal de las aplicaciones de tiempo real ejecutadas en el sistema operativo Android basado en el kernel de Linux.

3.4 Solución con aislamiento para mejorar respuesta temporal

Este capítulo presenta la solución desarrollada para mejorar la respuesta temporal de las aplicaciones de tiempo real sin necesidad de realizar cambios profundos en el sistema operativo. De este modo se conseguirá obtener una solución fácilmente mantenible y portable para diferentes tipos de hardware y distintas versiones del sistema.

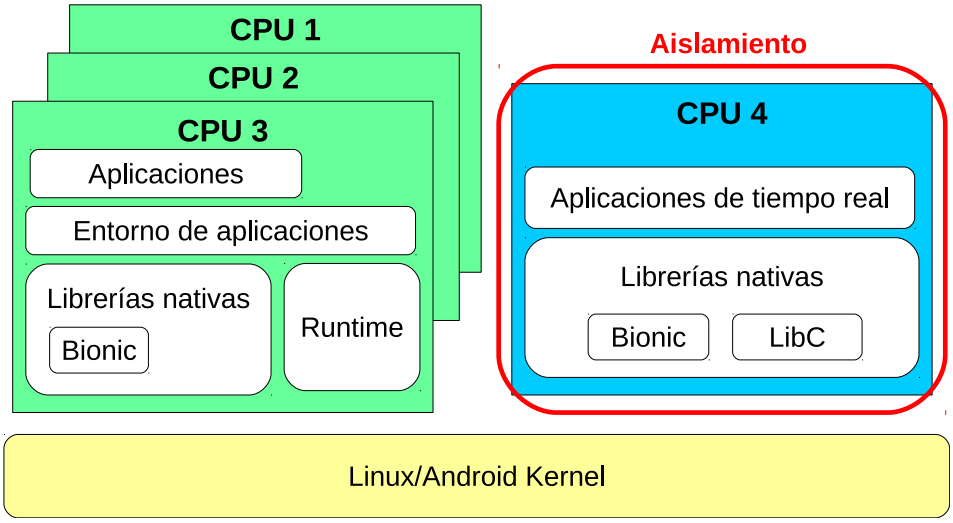

Fig. 3.2: Solución propuesta para ejecutar aplicaciones de tiempo real en Android.

La solución propuesta e ilustrada en la Figura 3.2 plantea la ejecución de aplicaciones de tiempo real en un núcleo aislado del procesador. Además, estas aplicaciones se ejecutarán directamente sobre las librerías nativas y el kernel del sistema operativo configurado debidamente para tiempo real,

y de este modo se conseguirá evitar la incertidumbre temporal causada por la máquina virtual de Java (Dalvik/ART) utilizada por la mayoría de aplicaciones desarrolladas para Android. Específicamente en la Figura 3.2 se muestra a modo de ejemplo un procesador con cuatro núcleos donde uno de ellos (CPU 4) es utilizado en exclusiva por las aplicaciones de tiempo real.

En la siguiente sección se describirán los mecanismos que están disponibles en Linux/Android para evitar que el planificador asigne tareas a determinados núcleos.

3.5 Mecanismos de aislamiento en Linux/Android

Android se ha ido basando en diferentes versiones del kernel de Linux desde que empezara a utilizar la versión 2.6.25 en Android 1.0. Las últimas versiones de este sistema operativo se basan en versiones del kernel 4.x. Por lo tanto, es posible hacer uso de las funcionalidades avanzadas de Linux para aislar núcleos específicos de la actividad del planificador. Estas funcionalidades incluidas en el kernel se denominan *isolcpus* y *cpusets*; ambas son evaluadas a continuación.

3.5.1 Isolcpus

Desde el lanzamiento de la versión 2.6.9 en octubre de 2004, el kernel de Linux incluye un parámetro de arranque denominado *isolcpus* que permite establecer una lista de núcleos (o procesadores) aislados. Estos nunca serán tenidos en cuenta por el planificador de Linux ni por el algoritmo de balanceo de carga. Por lo tanto, el planificador no ejecutará procesos en un procesador o núcleo aislado, excepto los hilos del kernel, los manejadores de interrupciones o aquellos hilos o procesos asignados explícitamente mediante su afinidad. Para configurar la afinidad de una tarea se puede utilizar una llamada al sistema denominada *sched_setaffinity* o el comando *taskset* de Linux.

Isolcpus es un parámetro de configuración que se ajusta al iniciar el sistema, así que no puede ser modificado mientras el sistema operativo está en funcionamiento. Solo puede ser configurado durante el arranque del sistema operativo, es decir, se tendrá que realizar desde el gestor de arranque (*bootloader*, en inglés). En el caso de una distribución Linux de escritorio se realizaría desde un gestor como el *GRUB*. En el caso de un dispositivo Android el proceso es algo más complejo. En Android es posible usar un gestor

de arranque como *U-Boot*[5], sin embargo, la mayoría de fabricantes tienen su propia versión del gestor de arranque diseñada específicamente para el hardware utilizado. Por lo tanto, en Android se debe crear una imagen de arranque con la herramienta denominada *mkbootimg*, donde estableceremos el parámetro isolcpus como una opción por línea de comando para el kernel (*cmdline*). Una vez hecho esto ya se puede utilizar la imagen de arranque creada en el dispositivo Android utilizado.

Existe una herramienta denominada *Tuned*[6] que, entre otras cosas, facilita el uso del parámetro de arranque *isolcpus* para aislar procesadores o núcleos de un procesador, pero por el momento solo está disponible para sistemas Linux Red Hat y Ubuntu.

A partir de la versión 4.15 del kernel de Linux esta opción ha quedado obsoleta, ya que en la propia documentación oficial relativa a los parámetros del kernel se nos desaconseja su uso en favor de los *cpusets* que describiremos en la siguiente subsección.

3.5.2 CPUSET

Mientras Android está ejecutando una aplicación, los servicios en segundo plano u otras aplicaciones pueden consumir gran parte de los recursos del sistema y esto podría provocar una ejecución lenta e impredecible. Para solucionar este problema, Android utiliza una característica del kernel llamada *cgroups* (abreviación de *control groups* en inglés) que limita y aísla el uso de recursos (CPU, memoria, red, etc) sobre un conjunto de procesos.

Hay dos mecanismos que utiliza Android para impactar en la planificación: la prioridad de procesos o hilos (*nice*) y los *cgroups* [68]. El nivel de prioridad determinado por el valor *nice* tiene un impacto sobre la política de planificación justa de Linux/Android; los hilos o procesos con un valor *nice* mayor tendrán una ejecución menos frecuente que aquellos que tengan un valor menor. Aparentemente esto podría ser suficiente para asegurar que los procesos en primer plano no se vean afectados negativamente por la ejecución de los servicios en segundo plano, pero, en la práctica, esto es insuficiente porque en cualquier momento podríamos tener diversos procesos en segundo plano ejecutándose, provocando de este modo una bajada en el rendimiento de los procesos en primer plano. Por este motivo Android utiliza los *cgroups* a través de los cuales mueve los procesos en segundo plano a un grupo denominado *bg_non_interactive* y los procesos en primer plano al grupo llamado *default*. Los procesos que se encuentran en el grupo *bg_non_interactive* tienen un

[5] `https://www.denx.de/wiki/U-Boot/`
[6] `https://tuned-project.org/`

límite de uso del 5 % del procesador, mientras que los que se hallan en el grupo *default* pueden alcanzar el 95 % de ocupación del procesador. Por esta razón, los *cgroups* están activos por defecto en el kernel de Android.

Además, el kernel de Linux tiene una característica llamada *cpuset* que asigna procesadores (o núcleos) y nodos de memoria a los *cgroups*. Su objetivo es restringir el número de procesadores y recursos de memoria que un proceso o conjunto de procesos pueden utilizar. Sin embargo, Android no siempre tiene esta característica activada por defecto, y por lo tanto debemos activar la opción denominada CONFIG_CPUSETS disponible en los parámetros del kernel antes de su compilación.

Cuando utilizamos una arquitectura multinúcleo a través de los *cpusets* podemos controlar la asignación de procesos a los distintos núcleos del procesador. Los nodos de memoria también pueden ser asignados y controlados para su utilización por los distintos procesos. En el ámbito de los *cpusets* se consideran como los nodos de memoria todos los distintos bancos de memoria principal que existan en el sistema. Sin embargo, en los dispositivos Android se utilizan procesadores con multiprocesamiento simétrico (*SMP, Symmetric Multi-Processing*), y por lo tanto, únicamente cuentan con un nodo de memoria.

De acuerdo con la documentación oficial [69] los *cpusets* están representados como una jerarquía de directorios en un sistema de pseudo-ficheros, donde el directorio raíz en dicha jerarquía (comunmente ubicado en */dev/cpuset/*) representa el sistema al completo (todos los núcleos o procesadores y los nodos de memoria). Si la configuración por defecto de los *cpusets* no se modifica todos los núcleos o procesadores del sistema tendrán el comportamiento establecido de serie.

Un *cpuset* se considera descendiente de otro si en la jerarquía está ubicado como un directorio hijo. Si esto ocurre el *cpuset* descendiente heredará el subconjunto de núcleos o procesadores y nodos de memoria del *cpuset* padre.

Los procesos en el sistema solo pertenecen a un único *cpuset*. Un proceso solo se ejecutará en los procesadores o núcleos del *cpuset* al que pertenece. Si tenemos los suficientes privilegios podemos mover los procesos de un *cpuset* a otro. Por lo tanto, se podrían mover todos los procesos del sistema a un *cpuset* con un subconjunto de núcleos de un procesador multinúcleo, y así dejar un *cpuset* exclusivo con un subconjunto de núcleos para ejecutar en él los procesos con características de tiempo real.

Tomando la suposición de que contemos con un procesador de cuatro núcleos podríamos crear dos *cpusets*, uno de ellos contendrá todos los procesos del sistema, y en el otro se ejecutarán en exclusiva todas las aplicaciones

con requisitos temporales. En el Código 3.1 se ilustra un script donde se utilizan tres núcleos del procesador para alojar todos los procesos del sistema, mientras que el cuarto núcleo se deja en exclusividad para los procesos de tiempo real.

```
1 mkdir /dev/cpuset
2 mount -t cpuset none /dev/cpuset/
3 cd /dev/cpuset
4 mkdir sys
5 echo 0-2 > sys/cpus
6 echo 1 > sys/cpu_exclusive
7 echo 0 > sys/mems
8 mkdir rt
9 echo 3 > rt/cpus
10 echo 1 > rt/cpu_exclusive
11 echo 0 > rt/mems
12 echo 0 > rt/sched_load_balance
13 echo 1 > rt/mem_hardwall
14 for i in $(cat tasks); do echo $i > sys/tasks; done
```

Código 3.1: Script para crear dos *cpusets*, uno para los procesos del sistema y otro para los procesos con requisitos temporales

En el Código 3.1, en primer lugar creamos un directorio para seguidamente montar en él la jerarquía donde se albergarán los distintos *cpusets* que vayamos estableciendo (ver líneas 1 y 2). A continuación se crea un directorio denominado *sys* (línea 4), que será el *cpuset* para todos los procesos del sistema. En la línea 8 se crea otro directorio que define al segundo *cpuset* destinado a contener en exclusiva los procesos de tiempo real. Cuando creamos un nuevo directorio dentro de la jerarquía, automáticamente se generan una serie de ficheros que determinarán el comportamiento de ese *cpuset*. Para conseguir nuestro objetivo modificaremos los siguientes ficheros:

- **cpus**: Incluye una lista con el número de los procesadores o núcleos del sistema donde los procesos pertenecientes al *cpuset* pueden ejecutarse.

- **cpu_exclusive**: Este fichero puede contener únicamente dos valores, 0 ó 1. Si tiene el valor 1 establece que ningún otro *cpuset* que no sea hijo o padre pueda utilizar los mismos procesadores o núcleos. Cuando el valor es 0 la anterior restricción no es satisfecha.

- **mems**: Lista con los nodos de memoria donde los procesos del presente *cpuset* tienen permiso para acceder y manipular datos.

- **sched_load_balance**: El fichero puede tener el valor 0 ó 1. Cuando es 1 el kernel puede realizar un balanceo de los procesos entre los distintos procesadores o núcleos del *cpuset*. Si este fichero toma el valor 0 el kernel evitará el balanceo de procesos.

- **mem_hardwall**: Especifica si se restringe al kernel el alojamiento de las páginas de memoria y búferes de datos en los nodos de memoria usados por el *cpuset*. Si tiene el valor 0 (por defecto) indica que no existe dicha restricción, si el valor es 1 la restricción se hace efectiva.

- **tasks**: Contiene los identificadores de proceso (PIDs, *Process Identifier* en inglés) de los procesos que se ejecutan en el *cpuset*.

Teniendo en cuenta todo lo comentando anteriormente, con el Código 3.1 se generan dos *cpusets*. En el *cpuset* denominado *sys* se emplazan todos los procesos del sistema que no sean de tiempo real. Para conseguir esto en la línea 14 se mueven todos los procesos actuales del sistema a dicho *cpuset*. Además, hay que tener en cuenta que todos los futuros hijos de los procesos desplazados serán ubicados en el mismo *cpuset*. Por otro lado, a través del fichero *tasks* añadiremos los procesos de tiempo real al *cpuset* denominado *rt*. Para clarificar mejor esta situación en la Figura 3.3 se ilustra lo que se consigue al ejecutar el Código 3.1. Hemos tomado como ejemplo un sistema multinúcleo con 4 núcleos donde 3 de ellos son utilizados para los procesos del sistema y el cuarto en exclusiva para los procesos de tiempo real. El número de núcleos asignado a cada *cpuset* podría ser distinto, en este caso el *cpuset* destinado a los procesos de tiempo real podría tener dos núcleos. En caso de tener más de un núcleo para la parte de tiempo real podríamos utilizar una planificación particionada o dejar al planificador del sistema que haga el balanceo de los procesos entre los distintos núcleos (*sched_load_balance* con valor a 1).

Una limitación que presenta este mecanismo es que, a pesar de que los procesos del sistema se mueven a un *cpuset*, los procesos que son hijos de *kthreadd* no pueden ser desplazados. Este demonio se ejecuta en el espacio del kernel y es usado para crear nuevos procesos del kernel. Por lo tanto, no podemos asegurar que esta clase de procesos no se ejecuten en un núcleo aislado.

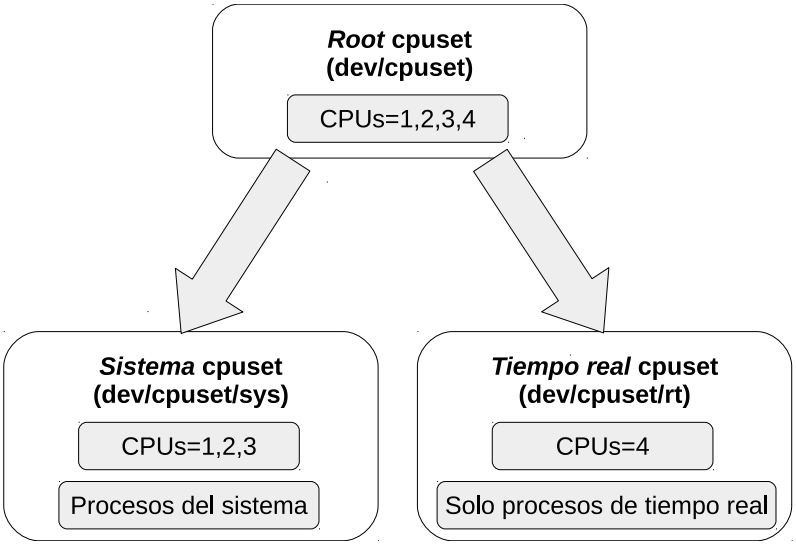

Fig. 3.3: Ejemplo de dos *cpusets* diferentes para aislar los procesos de tiempo real en un sistema multinúcleo con 4 núcleos.

3.5.3 CPUSET o Isolcpus

Estos dos mecanismos pueden ser utilizados para aislar núcleos en un sistema multinúcleo y utilizarse para ejecutar aplicaciones de tiempo real sobre ellos. Sin embargo, existen algunas desventajas si decidimos utilizar el mecanismo *isolcpus*.

En primer lugar, *isolcpus* es un mecanismo que solo puede ser cambiado durante el arranque del sistema. Además, en la mayoría de casos, es necesario modificar la imagen de arranque de Android cada vez que queramos realizar una modificación en la distribución de los núcleos aislados. En el caso de los *cpusets* las modificaciones se pueden realizar durante la ejecución del sistema.

En segundo lugar, el uso de *isolcpus* no impide a las aplicaciones asignar procesos o hilos a uno de los núcleos aislados. Los procesos podrían ser asignados utilizando la llamada al sistema *sched_setaffinity* o el comando *taskset*. Por lo tanto, no se puede asegurar que ninguna aplicación del sistema utilice la función *sched_affinity* para ejecutarse en alguno de los núcleos aislados. Este inconveniente no existe con los *cpusets* porque la asignación de procesos a los distintos *cpusets* definidos en el sistema se realiza mediante la escritura del identificador de procesos (PID) en el fichero denominado *tasks*. Esto imposibilita que las aplicaciones del sistema asignen procesos o hilos a núcleos pertenecientes a la parte de tiempo real. Sin embargo, ninguno de los dos mecanismos permite controlar dónde se emplazan los procesos pertenecientes al kernel, porque estos son hijos del proceso *kthreadd* y la

implementación del kernel de Linux no permite al usuario asignar a los procesos de kernel a un núcleo o procesador específico.

Y en tercer lugar, como ya se ha comentado en la Subsección 3.5.1 a partir de la versión 4.15 del kernel de Linux el mecanismo *isolcpus* ha quedado obsoleto y se desaconseja su uso.

3.5.4 Afinidad de las interrupciones

Cuando algún elemento hardware como por ejemplo, una tarjeta de red o dispositivo de almacenamiento, envía una interrupción, esta necesitará ser atendida por el procesador. Estas interrupciones podrían interferir en la ejecución de las aplicaciones de tiempo real cuando utilizamos alguno de los mecanismos descritos anteriormente para aislar núcleos.

La afinidad de una interrupción se define como el conjunto de núcleos o procesadores que pueden atender dicha interrupción cuando se produce. Para balancear la carga que genera la atención de las interrupciones, los sistemas operativos basados en Linux poseen un demonio llamado *irqbalance* que ayuda a balancear dicha carga entre los distintos procesadores o núcleos y así aumentar el rendimiento. No obstante, a partir de la versión 2.4 del kernel de Linux tenemos la posibilidad de asignar ciertas interrupciones a procesadores o núcleos específicos. Esta facilidad ofrecida por Linux se denomina *SMP IRQ affinity* y permite controlar qué núcleos o procesadores atenderán las distintas interrupciones que se producen en el sistema.

Por cada interrupción existe un directorio en */proc/irq*, que contiene un fichero llamado *smp_affinity*, donde es posible cambiar la afinidad de la interrupción modificando una máscara de bits. Esta máscara representa qué núcleo o procesador atenderá la interrupción en cuestión; cada bit de ella corresponde en nuestro caso a un núcleo del procesador. Si por ejemplo tenemos un procesador con cuatro núcleos y queremos que una interrupción sea atendida solamente por el primer núcleo tendríamos que escribir en el fichero el valor hexadecimal 1. En el caso de querer que la interrupción sea atendida por tres núcleos deberíamos escribir el valor hexadecimal 7. Para clarificar estos dos ejemplos, se ilustra en la Figura 3.4 qué valores binarios y hexadecimales debe tener la máscara en función de los núcleos que queramos utilizar.

Con el Código 3.2 se consigue modificar la afinidad de todas las interrupciones enmascarables, para que así sean atendidas solo por los tres primeros núcleos de un procesador de cuatro núcleos. En este caso concreto el cuarto núcleo verá reducido el tiempo dedicado a ejecutar manejadores de interrup-

	Binario	Hexadecimal
CPU 0	0001	1
Máscara	0001	1

	Binario	Hexadecimal
CPU 0	0001	1
CPU 1	0010	2
CPU 2	0100	4
Máscara	0111	7

Fig. 3.4: Ejemplo de utilización de la máscara de bits para determinar qué núcleos del procesador atenderán una interrupción.

ción mejorando de este modo los tiempos de respuesta de las aplicaciones que se ejecuten en él.

```
1 for i in $(find /proc/irq -name "smp_affinity");
2 do echo 7 > $i;
3 done
```

Código 3.2: Script para dejar un núcleo sin atender interrupciones enmascarables.

Cabe destacar que no todas las interrupciones permiten cambiar su afinidad, por ejemplo las interrupciones entre procesadores (*IPIs*, *inter-processor interrupt* en inglés). Se deben cumplir dos requisitos para que el sistema operativo nos deje modificar la afinidad de una interrupción:

1. El controlador de la interrupción debe tener soporte para una tabla de redirecciones (como por ejemplo, el controlador *I/O-APIC* en el caso de procesadores con arquitectura IA-64).

2. La afinidad de la interrupción no puede estar gestionada por el kernel.

Con el mecanismo descrito en esta sección gran parte de las interrupciones que se producen en el sistema pueden ser asignadas a los núcleos destinados a ejecutar aplicaciones sin requisitos de tiempo real. Aunque para algunas interrupciones no tenemos la garantía de que no vayan a ser atendidas por núcleos aislados.

3.6 Ajustes y estrategias para mejorar la respuesta temporal con aislamiento

El aislamiento supone una importante mejora en los tiempos de respuesta (tal y como se verá en la Sección 3.7) pero es necesario aplicar otros ajustes y estrategias sobre el sistema para ejecutar las aplicaciones de tiempo real con una mayor predictibilidad temporal.

3.6.1 Planificación de tiempo real con mecanismos de aislamiento

Los mecanismos de aislamiento descritos previamente permiten ejecutar procesos en un núcleo aislado de un procesador. Por lo tanto, estos procesos tendrán menos interrupciones o interferencias en comparación con procesos que se están ejecutando en núcleos no aislados. Para conseguir tener unos tiempos de respuesta acotados, las aplicaciones con requisitos temporales necesitan hacer uso de un planificador de tiempo real. Afortunadamente, como ya hemos descrito en la Sección 3.3 los sistemas operativos basados en Linux al menos proporcionan dos políticas de planificación de tiempo real (*SCHED_FIFO* y *SCHED_RR*).

En los sistemas Linux, los hilos que utilizan una política de planificabilidad de tiempo real tienen prioridad en su ejecución sobre cualquier otro que utilice el planificador general completamente justo. De este modo, simplemente utilizando una política de tiempo real es posible mejorar los tiempos de respuesta medios de manera significativa. Es importante destacar que si no se utiliza algún mecanismo de aislamiento, el uso de una política de planificación no proporciona garantías de que no existan otros procesos o hilos en el sistema utilizando una política de planificabilidad de tiempo real con mayor prioridad. Esto podría provocar que nuestras aplicaciones de tiempo real se vieran expulsadas en mitad de una ejecución, y por lo tanto, afectaría a sus tiempos de respuesta.

3.6.2 Fijar frecuencia del procesador

La modificación dinámica de la frecuencia de un procesador es utilizada por el sistema operativo para disminuir de forma significativa el consumo energético [70]. El uso de esta funcionalidad en dispositivos móviles permite un incremento de la autonomía de sus baterías. Este tipo de estrategias no son exclusivas de los dispositivos móviles, ya que en la mayoría de sistemas operativos actuales y procesadores comerciales se utiliza debido a que no se requiere una frecuencia operacional alta durante todo el tiempo.

El cambio de frecuencia en un sistema operativo como Android se puede producir en cualquier momento durante la ejecución de un programa. Este cambio de frecuencia determinará cuántas instrucciones pueden ser ejecutadas durante un periodo de tiempo, es decir, afectará al tiempo de ejecución del programa que se esté ejecutando.

Esta estrategia de cambio de frecuencia está implementada directamente en el kernel de Linux y se denomina *cpufreq*. Desde la versión 3.4 del kernel todos los módulos necesarios son cargados automáticamente. Para que sea posible el cambio de frecuencia existen tres componentes principales:

- **Módulo *cpufreq***: Proporciona una interfaz común entre las diferentes tecnologías de bajo nivel para el manejo de la frecuencia y el software de más alto nivel encargado de las políticas para modificar el cambio de frecuencia.
- **Drivers**: Estos drivers implementan las tecnologías usadas por los procesadores para la modificación de la frecuencia.
- **Gobernador**: La política que define cómo se modifica la frecuencia del procesador está determinada por un software denominado gobernador. Este gobernador es un algoritmo que calcula de la frecuencia en función de la carga de trabajo del procesador y del número de procesos activos.

En los dispositivos actuales con Android se incorporan distintos gobernadores que muchos fabricantes crean o adaptan, aunque en el kernel de Linux por defecto existen como mínimo 5 gobernadores:

- **Performance**: Se establece la frecuencia a la más alta para conseguir el mayor rendimiento.
- **Powersave**: Se utiliza la frecuencia más baja para minimizar el consumo energético.
- **Userspace**: Permite al usuario o a cualquier programa de usuario con los suficientes privilegios modificar y establecer la frecuencia de forma manual.
- **Ondemand**: La frecuencia es modificada dinámicamente en función de la carga de trabajo en el procesador.
- **Conservative**: Muy similar al gobernador *ondemand*, selecciona la frecuencia en función del nivel de uso del procesador. La diferencia principal reside en que los incrementos y decrementos en la frecuencia son más escalonados.

Para fijar la frecuencia se debe utilizar el gobernador denominado *userspace*. Esto es posible hacerlo en cualquier sistema Android modificando el fichero */sys/devices/system/cpu/cpuX/cpufreq/scaling_governor*. Para el cambio de frecuencia se pueden utilizar los ficheros que determinan la frecuencia

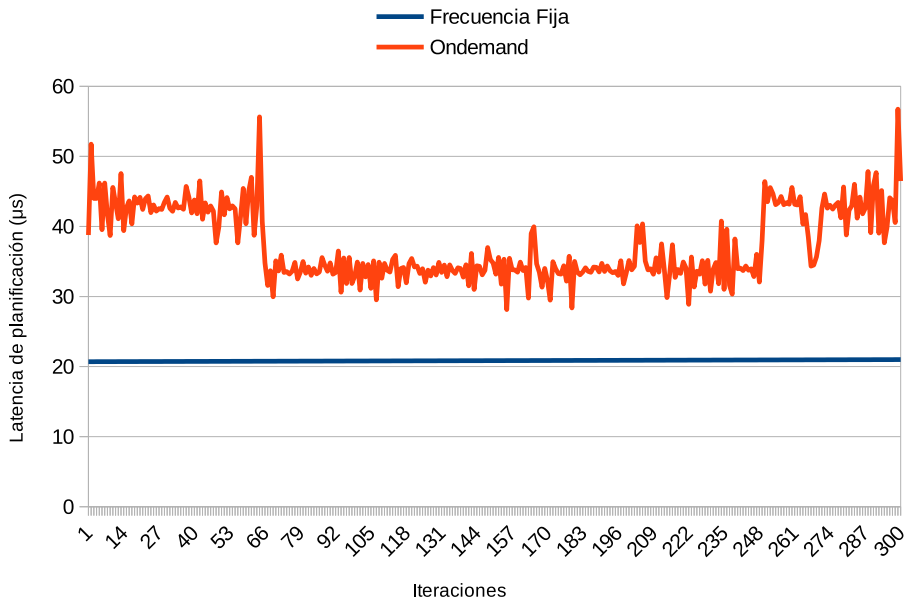

Fig. 3.5: Latencia de planificación observada en Android con la frecuencia fijada y con la frecuencia determinada por el gobernador *Ondemand*.

máxima y mínima (*/sys/devices/system/cpu/cpuX/cpufreq/scaling_max_freq* y */sys/devices/system/cpu/cpuX/cpufreq/scaling_min_freq*). Además en los sistemas multinúcleo es posible determinar de forma individualizada qué gobernador y frecuencias se utilizan en cada uno de los núcleos que conforman el procesador. Por lo tanto, la estrategia que vamos a seguir para los núcleos aislados consistirá en fijar las frecuencias máxima y mínima a un mismo valor (habitualmente al valor máximo).

En la Figura 3.5 se ilustra cómo un cambio dinámico de la frecuencia puede afectar a los tiempos de respuestas de los programas. Se ha realizado un test donde se ha medido la latencia de planificación en un dispositivo Android (Nexus 5). El test ha sido escrito en lenguaje C para que sea ejecutado directamente sobre el kernel y las librerías nativas. En dicho test un hilo con prioridad de tiempo real es ejecutado en un núcleo aislado utilizando los mecanismos de aislamiento expuestos en este capítulo (*cpusets* y afinidad de interrupciones). Este hilo se suspende durante 100 milisegundos y se mide en qué instante se reanuda, para así, calcular el retraso que ha ocurrido con respecto al instante teórico en el que debería haber comenzado su ejecución. Esta prueba se ha realizado con el gobernador *ondemand* usado por defecto en el sistema operativo. A continuación se ha ejecutado el mismo test fijando la frecuencia del núcleo aislado a su valor máximo. Comparando los resultados en la gráfica de la Figura 3.5 queda patente que cuando se utiliza una frecuencia fija las latencias son significativamente más homogéneas.

3.6.3 Desactivar mecanismos de ahorro de energía (*CPU-hotplug*)

Los sistemas operativos basados en Linux como Android poseen algunos mecanismos para disminuir el consumo energético y, por ende, aumentar la autonomía de las baterías. Algunos de estos mecanismos se aprovechan de los distintos estados de bajo consumo que ofrecen los procesadores actuales.

El kernel de Linux posee una facilidad para añadir o eliminar núcleos o procesadores en el sistema. Dicha facilidad se denomina *CPU-hotplug* y su función es la de permitir deshabilitar núcleos o procesadores sin tener que apagar por completo el sistema. Cuando la carga de trabajo es baja, algunos de los núcleos no utilizados pueden deshabilitarse. De este modo, estos núcleos consumirán menos energía. Cuando se produce el apagado de uno de los núcleos se realiza una migración de todos los procesos de ese núcleo a los núcleos disponibles. A continuación el controlador de interrupciones es notificado de que ese núcleo ya no está disponible para atender ninguna futura interrupción.

Desde una perspectiva de tiempo real utilizar un mecanismo como el de *CPU-hotplug* supone una desventaja, ya que el kernel necesita invertir un tiempo significativo para activar un núcleo que esté deshabilitado. Esto es algo que ya ha sido estudiado y demostrado en algunos trabajos previos [71] [72].

Para ejecutar aplicaciones de tiempo real en un núcleo del procesador y evitar que se produzcan incrementos en las latencias, dicho núcleo debe estar necesariamente habilitado (*online*). Para conseguir que un núcleo no se deshabilite se puede utilizar el script expuesto en el Código 3.3.

```
1 stop mpdecision
2 chmod 664 /sys/devices/system/cpu/cpuX/online
3 echo 1 > /sys/devices/system/cpu/cpuX/online
4 chmod 444 /sys/devices/system/cpu/cpuX/online
5 start mpdecision
```

Código 3.3: Script para dejar un núcleo habilitado permanentemente.

En algunos procesadores de la serie Snapdragon de Qualcomm utilizados en dispositivos Android se usa un demonio desarrollado específicamente para controlar el estado de los núcleos. Este demonio llamado *mpdecision* puede ser detenido (ver línea 1 en el Código 3.3). Por otro lado, en algunos procesadores Mediatek también es posible desactivar el mecanismo usado

en el sistema para apagar o encender los núcleos. En este caso se realiza modificando el fichero */proc/hps/enabled* con un valor igual a 0.

En el Código 3.3 se modifican los permisos de escritura y lectura del fichero del sistema que controla el estado de cada uno de los núcleos del procesador (líneas 2 y 4). Al hacer esto es posible forzar que un núcleo quede habilitado de forma permanente, ya que ningún otro elemento del sistema podrá modificar el fichero denominado *online* que determina si un núcleo está encendido o apagado.

3.6.4 Parámetros del kernel de Linux/Android

Con el objetivo de poder aplicar los mecanismos de aislamiento descritos en el presente capítulo y de mejorar los tiempos de respuestas de las aplicaciones de tiempo real, algunos parámetros de configuración del kernel de Linux/Android deben ser tenidos en cuenta.

Para utilizar el mecanismo de los *cpusets* es necesario que la opción llamada CONFIG_CPUSETS esté activada en los ajustes del kernel. En caso de que no esté activa es necesario activarla y recompilar el kernel.

Asimismo, el parámetro de configuración del kernel denominado CONFIG_PREEMPT debe estar activo. Dicho parámetro hace que el código del kernel sea expulsable, a excepción de los manejadores de interrupciones y las regiones protegidas por *spinlocks*. De acuerdo con [73], cuando tenemos un kernel configurado con el parámetro CONFIG_PREEMPT, la latencia de peor caso observada puede ser de hasta cientos de milisegundos, aunque algunos drivers pueden tener manejadores de interrupciones que introduzcan latencias mucho mayores.

Existe una opción de configuración del kernel que puede reducirnos las interrupciones ocasionadas por el planificador. Esta opción se llama CONFIG_NO_HZ_FULL, y en caso de estar activa para alguno de los núcleos del procesador, las interrupciones de planificación ocurrirían aproximadamente cada segundo, por lo tanto, puede reducir la latencia. Sin embargo, debe ser activada si en el sistema existe algún núcleo del procesador donde se vaya a ejecutar exclusivamente una sola tarea de tiempo real, o una única tarea que necesite mucho tiempo de cómputo.

En las estructuras internas del kernel se utiliza un mecanismo de sincronización llamado RCU (*read-copy-update*) [74]. Este mecanismo permite actualizar las estructuras de datos sin la necesidad de bloquear a los posibles lectores por la actualización de los datos de un escritor. RCU asegura que los lectores obtendrán datos coherentes manteniendo múltiples versiones de ellos. Además, dichos datos no serán modificados hasta que transcurra un

periodo de gracia para que todos los lectores finalicen sus operaciones. Para la implementación de este mecanismo se utilizan regiones de código no expulsables, lo que provoca irremediablemente una degradación en los tiempos de respuesta [75]. Por este motivo el kernel de Linux/Android ofrece la opción CONFIG_RCU_BOOST que aumenta la prioridad de aquellos lectores que no están en ejecución y provocan que el *periodo de gracia* sea demasiado largo.

Existe otra opción relacionada con el mecanismo RCU que puede tener un efecto sobre las latencias máximas. Este parámetro se denomina CONFIG_RCU_BOOST_PRIO y es usado para incrementar la prioridad de tiempo real de aquellos lectores que han sido expulsados de su ejecución. Cuando se va a trabajar con aplicaciones de tiempo real que tengan uno o más hilos que realicen cálculos que saturen el procesador; se debe especificar en este parámetro una prioridad superior a la del proceso de tiempo real más prioritario que haga uso intenso del procesador. En caso de que no exista ningún proceso de tiempo real que sature al procesador puede dejarse el valor por defecto (nivel de prioridad 1). También es posible que existan aplicaciones de tiempo real que estén compuestas de hilos que no saturen de forma individual al procesador, pero que su ejecución conjunta sí lo haga. En este caso, la prioridad del parámetro debería ser superior a la del hilo de tiempo real con menor prioridad.

Como hemos comentado algunas de estas opciones pueden no estar activas por defecto, y por lo tanto, con mucha probabilidad deberemos recompilar el kernel de Linux/Android del dispositivo que vayamos a utilizar. Este proceso conlleva realizar los siguientes pasos; descargar las fuentes del kernel, seleccionar la arquitectura y hardware específicos, activar las opciones necesarias, realizar la compilación, y finalmente instalar el kernel generado en el dispositivo.

3.6.5 Ajuste de la memoria

Uno de los recursos que debe gestionar un sistema operativo es la memoria RAM. Por este motivo, los kernels basados en Linux utilizan un componente software para tomar las decisiones necesarias cuando el sistema se queda sin suficiente memoria. El kernel oficial de Linux posee un componente denominado *OOM, out-of-memory killer* que es utilizado para liberar memoria cuando no hay suficiente espacio para alojar nuevos procesos, es decir, cuando la memoria virtual de todos los procesos es superior a la suma de la memoria física y de la memoria virtual de intercambio alojada en disco (*swap*).

Android tiene su propia implementación para el componente *OOM*, ya que en los sistemas Linux clásicos no se tiene en cuenta cómo de frecuente ha

sido el acceso a un proceso o aplicación para ser eliminado de la memoria. Por lo tanto, Android clasifica y prioriza los procesos a través de los siguientes grupos:

1. **Procesos activos:** Es el proceso que está ejecutando la aplicación que el usuario está utilizando en primer plano. Este tipo de procesos tienen una preferencia máxima, de tal modo que nunca deberían ser eliminados de la memoria principal.

2. **Procesos visibles:** Procesos que son utilizados por la aplicación en primer plano.

3. **Servicios:** Procesos que están siendo ejecutados en segundo plano.

4. **Procesos ocultos:** Son los procesos de las aplicaciones que no están visibles.

5. **Proveedor de contenido:** Aquellos procesos que ayudan a las aplicaciones a gestionar el acceso a los datos que ellas u otras aplicaciones almacenan. Además proporcionan una forma para compartir datos entre aplicaciones.

6. **Procesos vacíos:** Todos aquellos procesos que ya han finalizado, pero todavía siguen presentes en la memoria principal. Estos procesos tienen la prioridad más baja para mantenerse en memoria.

Cada proceso en el sistema tiene un valor numérico de ajuste llamado *oom_adj* que establecerá a qué grupo de los anteriores pertenece, y se podrá modificar dinámicamente. Cuanto más alto sea el valor *oom_adj* más probable será que el proceso sea sacado de la memoria principal. Además, por cada grupo se utiliza un valor (*minfree*) para establecer el umbral que dispara la necesidad de eliminar alguno de los procesos en memoria principal.

Por lo tanto, los procesos con requisitos de tiempo real que vayan a ser ejecutados en un núcleo aislado deben tener el valor más bajo disponible para que su expulsión de la memoria sea improbable. Es posible modificar el valor *oom_adj* de un proceso a través del fichero */proc/PID/oom_adj*. Así que, hay que cerciorarse que los procesos de tiempo real tengan el valor más bajo posible, que en Android es -17.

3.7 Evaluación de los mecanismos de aislamiento

En esta sección se evaluará la respuesta temporal de aquellas aplicaciones de tiempo real que se ejecutan sobre un núcleo aislado en el sistema operativo Android. Para conseguir aislar un núcleo del procesador se utilizará el mecanismo *cpusets* y la afinidad de interrupciones. Además el sistema estará configurado según lo descrito en la Subsección 3.6.4.

3.7.1 Entorno de pruebas

Todos los tests han sido ejecutados en un teléfono Nexus 5 con Android 6.0 Marshmallow con permisos de superusuario y un kernel 3.4.0. El Nexus 5 posee un procesador Qualcomm MSM8974 Snapdragon 800 de cuatro núcleos a una frecuencia máxima de 2.3 GHz y 2GB de memoria RAM. En este caso concreto el fabricante del procesador utiliza el demonio *mpdecision* para controlar el encendido y apagado de los núcleos. Sobre este dispositivo se ha ejecutado el script del Código 3.4 que fija la frecuencia de uno de sus cuatro núcleos y deshabilita su apagado. Además, crea dos *cpusets*, uno de ellos incluye tres núcleos del procesador y contiene todos los procesos del sistema y otro incluye el núcleo restante y se destina exclusivamente a la ejecución de los procesos de tiempo real. Por último se modifica la afinidad de la interrupciones para que todas aquellas que permitan realizar dicho cambio no sean procesadas en el núcleo aislado.

```
1  stop  mpdecision
2  chmod 664 /sys/devices/system/cpu/cpu3/online
3  echo 1 > /sys/devices/system/cpu/cpu3/online
4  chmod 444 /sys/devices/system/cpu/cpu3/online
5   start  mpdecision
6
7  echo userspace > /sys/.../cpu3/.../scaling_governor
8  echo 1958400 > /sys/.../cpu3/.../scaling_max_freq
9  echo 1958400 > /sys/.../cpu3/.../scaling_min_freq
10
11 mkdir /dev/cpuset
12 mount -t cpuset none /dev/cpuset/
13 cd /dev/cpuset
14 mkdir sys
15 echo 0-2 > sys/cpus
```

```
16 echo 1 > sys/cpu_exclusive
17 echo 0 > sys/mems
18 mkdir rt
19 echo 3 > rt/cpus
20 echo 1 > rt/cpu_exclusive
21 echo 0 > rt/mems
22 echo 0 > rt/sched_load_balance
23 echo 1 > rt/mem_hardwall
24
25 for i in $(cat tasks);
26 do echo $i > sys/tasks;
27 done
28
29 for i in $(find /proc/irq -name "smp_affinity");
30 do echo 7 > $i;
31 done
```

Código 3.4: Script utilizado para aislar uno de los núcleos del procesador de un Nexus 5.

Para evaluar la respuesta temporal de los mecanismos aplicados se ha ejecutado un test para realizar la medida de las interferencias sufridas por una aplicación de tiempo real bajo diversas condiciones. El test consiste en medir el tiempo entre dos instrucciones consecutivas que realizan una lectura del reloj y, por lo tanto, se puede observar si existen interferencias entre las dos lecturas consecutivas (ver Código 3.5).

```
1 while(i<ITERATIONS){
2   clock_gettime(CLOCK_MONOTONIC, &time1);
3   clock_gettime(CLOCK_MONOTONIC, &time2);
4   calc_diff(&time1, $time2)
5   ...
6 }
```

Código 3.5: Código C utilizado para medir el tiempo transcurrido entre dos instrucciones consecutivas.

Estos tests se ejecutan siempre en un núcleo específico del procesador utilizando la llamada al sistema *sched_affinity* o el mecanismo *cpusets*. Asimismo, se ha utilizado el lenguaje de programación C con un compilador cruzado para la arquitectura ARM; de este modo, los tests se ejecutan directamente sobre el kernel y la librería Bionic. Por lo tanto, se evita la utilización de

la máquina virtual de Java para Android (ART), evitando los problemas de incertidumbre temporal que genera su uso.

Los tests se han ejecutado en un sistema con una carga de trabajo muy alta. Para conseguir esto, se ha utilizado una conocida aplicación para medir el rendimiento de teléfonos y tabletas llamada AnTuTu [7]. Esta aplicación mide el rendimiento general del sistema estresando el procesador con diversos algoritmos.

Además, para asegurarnos que el sistema está estresado y genera numerosas interrupciones durante todo el tiempo de ejecución de los tests se reproduce un vídeo en alta definición utilizando la aplicación de Youtube.

El núcleo aislado solo ejecutará nuestro test que consumirá prácticamente la totalidad del tiempo del procesador. Solo se verá interrumpido por hilos del kernel y alguna de las interrupciones no enmascarables. El resto de los núcleos ejecutarán la aplicación AnTuTu y el vídeo en alta definición.

3.7.2 Evaluación de cpusets, afinidad de interrupciones y SCHED_FIFO

En esta subsección se muestran los resultados de dos tests (test A y test B), ambos ejecutados en el entorno descrito previamente. El test A es ejecutado con prioridad 99 de tiempo real (SCHED_FIFO) en un sistema donde no hay ningún núcleo aislado. Además, con el objetivo de observar la situación de peor caso, todas las interrupciones que permiten el cambio de afinidad son asignadas al núcleo donde está corriendo este test.

Con el test B se utiliza el mecanismo *cpuset* para aislar un núcleo y lanzar en él dicho test. Asimismo, a través de la afinidad de interrupciones se eliminan del núcleo asilado todas aquellas interrupciones que permiten modificar su afinidad. A diferencia del test A, se utiliza la política de planificabidad SCHED_FIFO con una prioridad media (50).

Se ha llevado a cabo un experimento con 10^8 iteraciones, donde se ha medido el tiempo transcurrido entre dos lecturas consecutivas de reloj (ver Código 3.5). En la Tabla 3.1 se muestra el porcentaje de ocurrencias de ejecuciones sin y con interferencias para el test A y el test B. Como hemos observado que más del 99.8 % de las ejecuciones tienen un tiempo de ejecución que está entre 0.5 y 1 microsegundos, hemos considerado ejecuciones con interferencias significativas todas aquellas que se encuentran por encima de 1 microsegundo. A tenor de los resultados, queda patente cómo la aplicación de

[7] `http://www.antutu.com/en/index.htm`

	Test A: SCHED_FIFO (máxima prioridad)	**Test B: cpuset, afinidad de interrupciones, y SCHED_FIFO (prioridad media)**
Ejecuciones sin interferencias	99.802 %	99.993 %
Ejecuciones con interferencias	0.198 %	0.007 %

Tabla 3.1: Tabla comparativa con el porcentaje de interferencias observadas utilizando la política SCHED_FIFO con respecto al uso de un núcleo aislado utilizando *cpuset* y la afinidad de interrupciones.

	Test A: SCHED_FIFO (máxima prioridad)	**Test B: cpuset, afinidad de interrupciones, y SCHED_FIFO (prioridad media)**
35 μs-40 μs	26774	64
40 μs-200 μs	4123	49
200 μs-1000 μs	5879	0

Tabla 3.2: Número de ocurrencias observadas en diferentes intervalos para los tiempos de respuesta de dos instrucciones consecutivas.

los mecanismos de aislamiento descritos en este capítulo permiten alcanzar un número de ejecuciones sin interferencias significativamente mayor.

En la Tabla 3.2 se muestra el número de ocurrencias observadas en diferentes intervalos para los tiempos de respuestas de las iteraciones de los test A y B. Se puede ver cómo las ejecuciones de larga duración son significativamente menos cuando aislamos un núcleo y ejecutamos sobre él nuestro programa de test (test B). Concretamente hemos observado que en el test B el peor tiempo de respuesta ha sido de 179 microsegundos, mientras que en el test A, donde no se ha utilizado ningún mecanismo de aislamiento, se ha alcanzado un tiempo de respuesta de peor caso de 456 microsegundos. También se ha calculado el valor medio y la desviación estándar de ambos tests (ver Figura 3.6). Nuevamente, el test A donde no se ha aplicado ningún mecanismo de aislamiento muestra unos valores notablemente mayores que en el test B, en especial en la desviación estándar. Este hecho demuestra que existen tiempos de respuesta muy dispersos, por lo tanto, la predictibilidad temporal es baja cuando no se utilizan los mecanismos de aislamiento descritos en el presente capítulo.

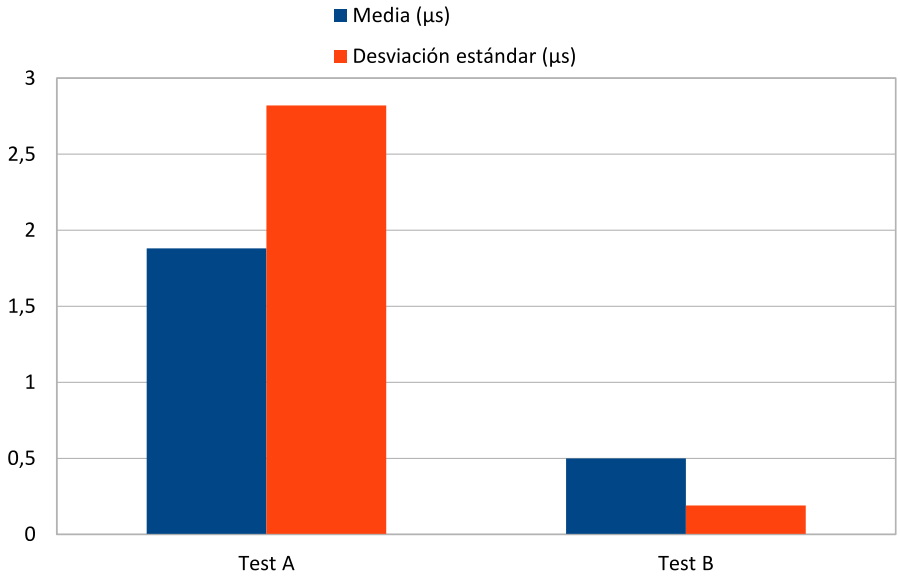

Fig. 3.6: Media y desviación estándar obtenida en la ejecución de los test A (solo prioridad de tiempo real máxima) y test B (con aislamiento)

3.7.3 Efectos de no usar *cpuset*

	Test C: SCHED_FIFO (prioridad media)	**Test D: cpuset, afinidad de interrupciones, SCHED_FIFO (prioridad media)**
Tiempo de respuesta de peor caso	1534129 μs	198 μs

Tabla 3.3: Tiempos de respuesta de peor caso observados durante la ejecución de dos instrucciones consecutivas en tests realizados con prioridad de tiempo real intermedia.

Se ha realizado otro experimento para determinar cuál es la mayor interferencia observada en la ejecución de dos instrucciones consecutivas de medida del reloj (Código 3.5) cuando utilizamos los mecanismos de aislamiento descritos en este capítulo (test C), y cuando solo usamos una política de planificación de tiempo real (test D). En esta ocasión se ha seleccionado una prioridad intermedia en el test D, para de este modo, demostrar cómo cualquier aplicación del sistema que utilice políticas de tiempo real con una prioridad mayor puede tomar el procesador y expulsarnos durante periodos de tiempo no acotados. En el experimento la aplicación AnTuTu genera procesos de tiempo real con alta prioridad que consumen recursos computacionales y hacen que existan tiempos de respuesta en el test C superiores al segundo

(ver Tabla 3.3). La diferencia con el test realizado en la Subsección 3.7.2 donde se medía el efecto producido por las actividades del sistema reside en que ahora lo que se pone de manifiesto son los efectos de otros hilos de tiempo real que se pueden crear por otras aplicaciones.

3.8 Conclusiones

En este capítulo hemos presentado una solución para poder ejecutar aplicaciones de tiempo real laxo en dispositivos multinúcleo con el sistema operativo Android. Gracias a la tendencia actual de utilizar procesadores multinúcleo podemos usar mecanismos disponibles en el kernel de Linux/Android para aislar algunos de los núcleos del procesador, y así ejecutar en ellos las aplicaciones con requerimientos temporales. La solución presentada es portable y por lo tanto puede ser usada en cualquier dispositivo Android actual. A diferencia de algunas otras soluciones propuestas en los trabajos que hemos descrito en el Capítulo 2 nuestra solución no requiere de una modificación a nivel de código del kernel de Linux/Android.

Aplicando nuestra solución en los experimentos llevados a cabo hemos podido observar que se han reducido considerablemente tanto el número de interferencias como la longitud de estas, mientras que si no se utilizan los mecanismos de aislamiento descritos en este capítulo se han observado interferencias de duración superior a los 1.5 segundos. Por lo tanto, podemos confirmar que hay una mejora muy sustancial en los tiempos de respuesta que nos permite ejecutar aplicaciones de tiempo real laxo en Android.

4 Lenguajes de programación con características de tiempo real en Android

> *El tiempo es el mejor autor; siempre encuentra un final perfecto.*
>
> **— Charles Chaplin**
> Actor, humorista y director inglés

Para ejecutar aplicaciones con restricciones temporales no basta solo con tener un sistema operativo de tiempo real, sino que es necesario disponer de lenguajes y librerías apropiados que nos ofrezcan características de tiempo real, para que de este modo las aplicaciones con requisitos temporales puedan funcionar correctamente. Por lo tanto, a lo largo de este capítulo se describirán las limitaciones para tiempo real de la librería *Bionic* (la versión de la librería estándar C utilizada en Android), se propondrá una alternativa que permita la utilización de la librería tradicional *GNU C Library* (*glibc*) en este sistema operativo, y se describirán los pasos necesarios para utilizar el lenguaje de programación Ada en el desarrollo de aplicaciones con requisitos temporales sobre Android.

El presente capítulo de esta tesis se organiza del siguiente modo; en la Sección 4.1 comenzaremos hablando sobre la librería *Bionic* utilizada en Android, además, se exponen sus limitaciones para la ejecución de aplicaciones con requisitos temporales. Posteriormente, en la Sección 4.2 se describe cómo es posible utilizar la librería tradicional *glibc* en Android para suplir las limitaciones de la librería *Bionic*. Durante la Sección 4.3 se realiza una caracterización temporal de algunas de las funciones más relevantes para las aplicaciones de tiempo real pertenecientes a la librería *glibc*. A continuación, en la Sección 4.4 se expone cómo es posible utilizar el lenguaje de programación Ada en Android para el desarrollo de aplicaciones de tiempo real. Finalmente, en la Sección 4.5 se presentan las conclusiones correspondientes a este capítulo.

4.1 Librería *Bionic* y sus limitaciones para tiempo real

Aunque las aplicaciones Android se escriben principalmente en el lenguaje de programación Java, es posible ejecutar código escrito en otros lenguajes para solventar algunas de sus limitaciones, como por ejemplo la penalización del rendimiento que supone el uso de la máquina virtual cuando gestiona la memoria dinámica [76]. Además, muchas de las librerías utilizadas para el desarrollo de aplicaciones en Java están escritas en lenguaje C, para mejorar su rendimiento. Por estos motivos Android proporciona un kit de desarrollo nativo (*Native Development Kit*, NDK) [33] que soporta los lenguajes C y C++.

La implementación de la librería estándar C habitualmente utilizada en Linux es la *GNU C Library* (comúnmente conocida como *glibc*). Esta librería proporciona soporte para los lenguajes C y C++, definiendo las llamadas al sistema y otras funciones básicas que utilizan casi todos los programas ejecutados en el sistema. Como Android está basado en el kernel de Linux parecería lógico asumir que se utilizase esta librería, sin embargo, debido a las especificaciones y requisitos de Android se decidió utilizar una nueva librería bautizada con el nombre de *Bionic*. Existen tres razones principales por las que se optó por no utilizar una librería estándar como la *glibc* e implementar una nueva:

- Un requisito de Android es que en él se puedan usar aplicaciones de terceros que no solo permitan ser monetizadas, sino que en ellas se pueda utilizar código propietario. Como la librería estándar C (*glibc*) usada en los sistemas Linux se distribuye bajo una licencia GPL que no permite enlazar con aplicaciones propietarias, se decidió crear una librería específica para Android llamada *Bionic* que usase una licencia BSD. Esta librería sigue siendo de código abierto, pero no obliga a que todo el código que se enlace con ella herede la condición de código abierto y, por consiguiente, los programadores de aplicaciones Android podrán desarrollar código propietario para este sistema operativo.

- Durante las primeras versiones de Android los dispositivos móviles por norma general tenían un espacio de memoria más limitado que otros dispositivos más “tradicionales”, como por ejemplo un ordenador de escritorio.

- Los procesadores que utilizan los dispositivos con Android poseen frecuencias inferiores si los comparamos con los que incorporan ordenadores de sobremesa, portátiles o servidores.

La libería *Bionic* tiene optimizado su tamaño gracias a que todos los comentarios de los ficheros de cabecera han sido eliminados, y han suprimido todo el código que han considerado inútil para Android. Además, ofrece una ventaja que reside en su optimización para procesadores lentos, para lo que se han realizado cambios en la implementación de la librería *Pthread* [77]. La librería *Pthread* proporciona soporte para la gestión de hilos y está incluida en la implementación tradicional de *glibc*. Además, se basa en el estándar POSIX definido por IEEE para especificar una API portable entre los diferentes sistemas operativos. Sin embargo, en la librería *Bionic* usada en Android se han realizado cambios en la librería *Pthread* que no son apropiados para las aplicaciones de tiempo real.

Para determinar si podemos usar *Bionic* en aplicaciones de tiempo real escritas en C hemos ido probando si las funciones más relevantes que se utilizan de manera habitual con requisitos temporales están disponibles. Se han detectado que algunas tan imprescindibles como las relacionadas con los protocolos de los mutexes [78] no se encuentran implementadas. A continuación se listan las funciones y constantes más relevantes no disponibles:

- Mutexes:
    - *pthread_mutexattr_setprotocol*: Función para establecer el protocolo de planificación de un mutex.
    - *pthread_mutexattr_setprioceiling* y *pthread_mutexattr_getprioceiling*: Establecen u obtienen el techo de prioridad de un mutex.
    - Constantes *PTHREAD_PRIO_PROTECT* y *PTHREAD_PRIO_INHERIT*.
    - *pthread_mutexattr_getrobust* y *pthread_mutexattr_setrobust*: Funciones para establecer y obtener la robustez de un mutex, lo que significa que especifican el comportamiento de un mutex cuyo hilo finaliza sin haberlo liberado.

La ausencia de las funciones anteriores demuestra que no están disponibles los protocolos de herencia de prioridad ni de techo de prioridad en la librería *Bionic*. No obstante, recientemente en una de las versiones más modernas de Android (9.0 API 28) se ha incorporado la herencia de prioridad para los mutexes, aunque hay que remarcar que su ausencia sigue existiendo en cualquier versión previa a Android 9.0.

Versión de Android	API	Funciones
4.2	16	842
4.2.2	17	870
4.3	18	878
4.4	19	893
5.0	21	1118
6.0	23	1183
7.0	24	1228
8.0	26	1280
9.0	28	1378
10	29	1394

Tabla 4.1: Número de funciones disponibles de la librería *Bionic* en las distintas versiones de Android [79].

- Prioridades:
  - *pthread_setschedprio* y *pthread_getschedprio*: Función que establece la prioridad de un hilo.
  - El símbolo *PTHREAD_EXPLICIT_SCHED*: Especifica que tanto la política de planificabilidad como los atributos asociados son establecidos a través del objeto de atributos.
- Hilos cancelables: Ninguna versión de Android soporta la terminación de hilos a través de las función POSIX *pthread_cancel*.

También existen otras funciones relevantes para aplicaciones de tiempo real que no están implementadas en la librería *Bionic* en versiones previas a Android 6.0 (API 23):

- Señales:
  - *sigwaitinfo*: Función para esperar una señal.
  - *sigtimedwait*: Función para esperar una señal durante un plazo de tiempo acotado.
  - *sigqueue*: Función para enviar señales acompañadas de información.

Con la evolución y mejora en el hardware de los dispositvos que ejecutan Android también se ha ido extendiendo la librería *Bionic*, para así ir incorporando en cada nueva versión del sistema operativo más funciones. Esto

se puede observar en la Tabla 4.1 donde se ilustran el número de funciones disponibles en la librería *Bionic* a lo largo de las distintas versiones de Android.

Lo primero que se podría considerar para poder solucionar las limitaciones de tiempo real descritas en esta sección sería modificar el código de la librería *Bionic* para dar soporte a las funciones no implementadas en las diferentes versiones de Android. Esto supondría un gran esfuerzo y una constante adaptación a las nuevas versiones de la librería que van apareciendo. Por ello, hemos decidido optar por una solución más portable y fácil de aplicar. Esta consiste en utilizar la librería tradicional *glibc* en Android. En la siguiente sección describimos cuál es el proceso a seguir para llevar esto a cabo.

4.2 *Glibc* en Android

Para poder utilizar la librería *glibc* en Android, en nuestro caso [1] es necesario usar una librería compilada y adaptada para arquitecturas ARM que se ejecute en sistemas operativos Linux. El modo más directo e inmediato es hacerlo mediante una compilación estática (en la orden de compilación se añade la opción *static*). Esto se realiza de forma cruzada en un dispositivo Linux que tenga la librería *glibc* para la arquitectura ARM/Linux. Así se consigue enlazar con todas las funciones necesarias para que el ejecutable final esté completo.

Utilizar la compilación estática puede ser un tanto limitante, ya que por ejemplo hace que los ejecutables ocupen más. Por ello también hemos resuelto el uso de librerías dinámicas. Para lograrlo se debe realizar lo siguiente:

- Copiar todas las librerías dinámicas al dispositivo Android donde queremos ejecutar de manera nativa las aplicaciones. Las librerías dinámicas pueden ser obtenidas de las proporcionadas por el compilador cruzado. Normalmente en distribuciones de Linux se ubican en la ruta */usr/compilador-cruzado/lib*.

- Durante la compilación es necesario indicar el enlazador dinámico que se utilizará y la ruta de las librerías dinámicas en el dispositivo Android. Esto se hace mediante las opciones de compilación *–dynamic-linker* y *–rpath*. A modo de ejemplo se expone en el Código 4.1 la orden de compilación para un programa sencillo.

[1] Todas las pruebas se realizan en un Nexus 5 con arquitectura ARM y la versión de Android 6.0

```
arm-linux-gnueabi-gcc hello_world.c \
-o hello_world -fPIE -pie \
-Wl,--dynamic-linker=/data/local/libs/ld-linux.so.3 \
-Wl,-rpath=/data/local/libs
```

Código 4.1: Instrucción para realizar una compilación cruzada donde se indica la ruta del enlazador dinámico y las librerías dinámicas.

Teniendo en cuenta que el kernel que incorporan los dispositivos con Android no sigue la rama principal de desarrollo del kernel de Linux, sería arriesgado afirmar que la librería *glibc* se puede utilizar sin inconveniente alguno en este sistema operativo sin antes realizar algunos tests. Por ello hemos adaptado los test funcionales que están disponibles en el conjunto denominado "Open POSIX Test Suite" [80], donde se realizan pruebas funcionales para:

- Hilos.
- Semáforos.
- Temporizadores.
- Variables condicionales.
- Colas de mensajes.
- Protocolos de herencia de prioridad con mutexes.

Todos estos tests han sido pasados satisfactoriamente cuando se ejecutaban en Android haciendo uso de la librería *glibc* y, por ende, podemos afirmar que es posible hacer uso de esta librería sobre el kernel Linux/Android.

4.3 Caracterización de funciones para tiempo real de la librería *glibc*

Para poder caracterizar algunas funciones POSIX utilizadas habitualmente con aplicaciones de tiempo real se ha utilizado un teléfono Nexus 5. Este teléfono cuenta con un procesador de cuatro núcleos a una frecuencia de 2.3 GHz y ejecuta la versión de Android 6.0. Además, se han realizado las medidas para dos casos distintos. En el primero de ellos (test A) se miden las funciones en un procesador sin ninguno de sus núcleos aislados; todos los hilos de los tests se ejecutan con prioridades de tiempo real. En el otro

	Test A: sin aislamiento	Test B: con un núcleo aislado
Bloqueo de un mutex libre (lock)	Mínimo: 1,874μs Máximo: 1646,301 μs Media: 2,143 μs	Mínimo: 0,261 μs Máximo: 34,064 μs Media: 0,318 μs
Bloqueo de un mutex libre (trylock)	Mínimo: 1,874 μs Máximo: 91,303 μs Media: 2,208 μs	Mínimo: 0,261 μs Máximo: 36,304 μs Media: 0,323 μs
Desbloqueo de un mutex (unlock)	Mínimo: 1,770 μs Máximo: 111,980 μs Media: 2,121 μs	Mínimo: 0,261 μs Máximo: 33,387 μs Media: 0,317 μs
Delay de 1000 μs: clock_nanosleep (absoluto)	Mínimo: 1030,924 μs Máximo: 5891,718 μs Media: 1112,932 μs	Mínimo: 1027,969 μs Máximo: 1108,230 μs Media: 1067,169 μs
Cambio de prioridad (pthread_setschedparam)	Mínimo: 7,970 μs Máximo: 180,053 μs Media: 15,793 μs	Mínimo: 7,454 μs Máximo: 64,147 μs Media: 8,114 μs
Cambio de prioridad (pthread_setschedprio)	Mínimo: 8,229 μs Máximo: 130,345 μs Media: 16,162 μs	Mínimo: 7,811 μs Máximo: 68,396 μs Media: 8,446 μs

Tabla 4.2: Caracterización temporal de algunas funciones POSIX de la librería *glibc* utilizadas habitualmente en aplicaciones de tiempo real.

caso (test B) se ha aislado uno de los núcleos del procesador aplicando los mecanismos descritos en el Capítulo 3 de esta tesis. Por lo tanto, se ejecuta el test B en un núcleo aislado con las mínimas interferencias posibles. En ambos casos se utiliza la librería tradicional *glibc*, y además en el sistema se ha establecido una carga de trabajo alta (ejecución de benchmarks y descarga de paquetes por red) para tratar de simular el peor de los escenarios posibles.

Los resultados que se muestran en la Tabla 4.2 se han obtenido midiendo la duración de las distintas funciones durante 150 mil ejecuciones para cada caso. La Tabla 4.2 pone de manifiesto que existe una mejora sustancial para los tiempos correspondientes al test B (núcleo aislado), donde los tiempos de peor caso son en algunos casos unas 7 veces inferiores y la media de los resultados es alrededor de solo un 15 % superior al valor mínimo obtenido. Los tiempos máximos observados en el test sin aislamiento pueden ser provocados por cualquier interrupción del sistema o por otras tareas que utilicen prioridades de tiempo real mayores. Incluso en el caso aislado, en un sistema operativo como Android donde no tenemos pleno control sobre

todas las actividades del sistema no se pueden determinar con total exactitud tiempos máximos de respuesta, aunque en los numerosos tests realizados hemos observado que se obtienen valores similares a los mostrados en la Tabla 4.2.

4.4 Ada en Android

Ada proporciona soporte nativo en el propio lenguaje para las características requeridas en los sistemas de tiempo real. Además, este lenguaje incluye soporte para características adicionales que son útiles para sistemas embebidos, tales como programación por contrato, datos fuertemente tipados, cláusulas de representación, comprobaciones estáticas en la compilación y primitivas avanzadas para la sincronización. Todo ello contribuye a la capacidad de desarrollar software más fiable. Por esos motivos es muy interesante poder desarrollar aplicaciones Ada para Android.

Pudiendo compilar programas C/C++ utilizando la librería tradicional *glibc* en Android es sensato pensar que se podría utilizar el lenguaje de programación Ada en este sistema teniendo un compilador adecuado para el hardware que vayamos a utilizar. Para la consecución de este objetivo ha sido necesario generar una versión cruzada del compilador GNAT para la arquitectura ARM/LInux.

GNAT es el compilador libre GNU Ada y es el único que soporta todos los anexos opcionales del estándar del lenguaje. Algunos autores del compilador GNAT crearon la compañía AdaCore para proporcionar soporte continuo en el desarrollo del compilador y también ofrecer un soporte profesional.

Actualmente hay tres distribuciones principales de GNAT que se describen a continuación:

- **Edición GNAT GPL**: Está disponible de manera libre, está mantenida por la compañía AdaCore y no se distribuye para desarrollo profesional. Si se quiere distribuir un programa en código binario enlazado con la librería de ejecución de Ada (run-time system) se debe licenciar dicho programa con una licencia compatible con GPL, lo que implica distribuir el código fuente.

- **GNAT Pro**: Es una versión profesional de GNAT desarrollada por AdaCore, de pago, que utiliza una licencia de software comercial que no obliga a los creadores de aplicaciones a distribuir su código fuente.

- **GNAT para Android**: Previamente a esta tesis la empresa AdaCore propuso el uso del lenguaje Ada en Android a través de su compilador

cruzado de pago para la arquitectura ARM/Android [81]. En esta propuesta se utilizaba el lenguaje Ada en combinación con Java para desarrollar las partes más críticas del software (como por ejemplo secciones de uso intensivo de la CPU).

- **GNAT FSF**: Esta versión está incorporada en el popular sistema de compilación GCC, perteneciente a la "*Free Software Foundation*". Se distribuyen solamente las fuentes bajo una licencia GPL modificada que permite el desarrollo de software propietario.

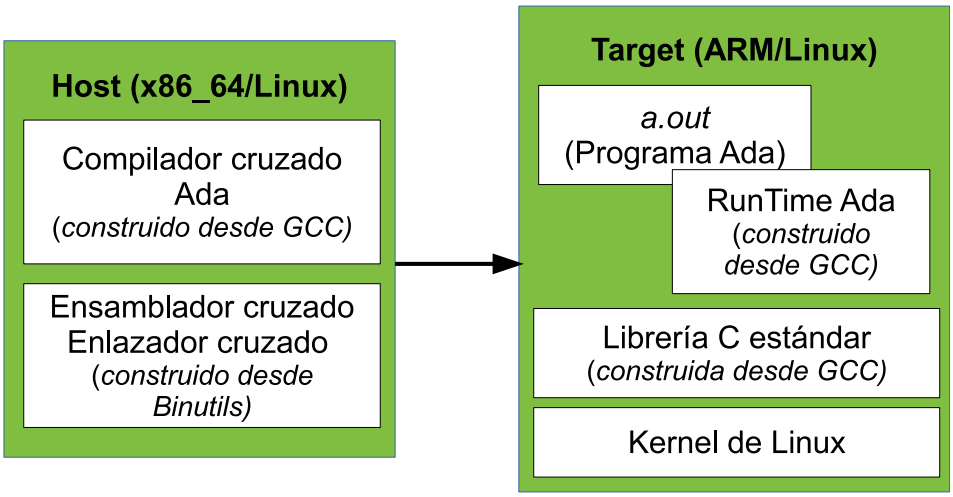

Fig. 4.1: Proceso de compilación cruzada y ejecución de un programa Ada con host x86_64 Linux en un target ARM/Linux.

Tomando en cuenta que no existe una versión gratuita de GNAT válida para Android, hemos decidido descargar las fuentes de GNAT FSF disponibles en GCC, para así generar un compilador cruzado GNAT específico para nuestro entorno de desarrollo. Hemos seleccionado la versión GNAT FSF debido a su licencia y porque permite construir diferentes versiones de GNAT adaptables a diferentes tipos de dispositivos.

Nuestro objetivo ha sido generar un compilador cruzado GNAT FSF para un entorno x86_64/Linux (*host*) que nos permita compilar programas para un dispositivo ARM/Linux (*target*). Para crear un compilador cruzado como el ilustrado en la Figura 4.1 es necesario generar previamente algunas herramientas y librerías. El proceso para crear todas las partes mostradas en la Figura 4.1 puede ser automatizado mediante la creación de un script [2] que permita construir cada una de las partes necesarias. En dicho script se deben construir u obtener de forma secuencial las siguientes herramientas y paquetes:

[2]El script para la creación de un compilador cruzado ha sido obtenido desde la siguiente dirección: `https://preshing.com/20141119/how-to-build-a-gcc-cross-compiler/`

1. **Binutils:** Conjunto de herramientas de programación para realizar manipulaciones sobre código objeto. Entre estas herramientas se incluyen el ensamblador o el enlazador.

2. **Cabeceras del kernel de Linux:** A través de estas cabeceras se definen las funciones exportadas por la versión del kernel de Linux donde se van a ejecutar los programas compilados.

3. **Compilador para C/C++ y Ada y librería de tiempo de ejecución:** Utilizando las herramientas obtenidas con *Binutils* se construyen los compiladores cruzados escogidos. Para realizar la generación de estos compiladores se utilizan los compiladores nativos de C/C++(gcc/g++) y de Ada (GNAT) de la máquina usada como *host* durante el proceso de construcción. Además, en este paso se compila la librería de tiempo de ejecución para Ada.

4. **Cabeceras de la librería estándar C y ficheros de inicialización:** En este paso se utilizan las cabeceras que definen las funciones de la librería *glibc*. Además utilizando el compilador cruzado del paso anterior se generan los ficheros de inicialización para la arquitectura hardware especificada.

5. **Librería de apoyo al compilador:** Se utiliza el compilador generado en el paso 3 y los ficheros de inicialización del paso 4 para crear esta librería, que será necesaria para la construcción de las librerías estándar C/C++ del siguiente paso.

6. **Librerías estándar C/C++:** En el último paso se generan las librerías estándar C/C++ para el hardware y sistema operativo seleccionados.

El compilador cruzado se ha construido para la arquitectura ARM/Linux porque no podía ser generado directamente para Android debido a las limitaciones que posee la librería *Bionic*. Sin embargo, como Android está basado en el kernel de Linux, tal y como hemos explicado en la Subsección 4.2, es posible ejecutar binarios generados con el anterior compilador cruzado si usamos la librería *glibc* en vez de la librería *Bionic*. Para utilizar la librería *glibc* en Android con un compilador GNAT debemos copiarla en nuestro dispositivo Android (*target*) e indicar el enlazador dinámico correcto a utilizar. También se debe determinar la ruta de las librerías dinámicas dentro del dispositivo. A modo de ejemplo mostramos una instrucción de compilación para un programa Ada en Android:

```
arm-unknown-linux-gnueabi-gnatmake \
hello_world.adb -fPIE -pie \
-Wl,--dynamic-linker=/data/local/libs/ld-linux.so.3 \
-Wl,-rpath=/data/local/libs
```

Código 4.2: Instrucción para realizar una compilación cruzada de un programa Ada en Android.

Para verificar que el proceso previo se puede llevar a cabo con todos los programas Ada sin que exista ningún problema, en la siguiente subsección vamos a describir los tests que se han ejecutado.

4.4.1 Probando Ada en Android

El conjunto de tests denominado *The Ada Conformity Assesment Test Suite (ACATS)* [82] se utiliza para verificar que los compiladores son conformes con el estándar del lenguaje Ada, y además permiten probar toda la funcionalidad de este lenguaje. Tal y como ya se ha descrito en un trabajo previo [83] los ACATS no están integrados de forma nativa en ningún entorno de test moderno y están diseñados para que los usuarios de estos tests creen scripts personalizados para su compilación, ejecución y análisis de resultados.

Existe un script que forma parte del conjunto de tests del código fuente del compilador GCC (localizado en *gcc/testsuite/ada/acats*). Dicho script por defecto está diseñado para usarse con un compilador nativo. Sin embargo, lo hemos adaptado para que se puede utilizar con el compilador cruzado de GNAT para ARM/Linux en un dispositivo Android [3]. El script original se ha dividido en dos partes: la primera de ellas (ejecutada en el dispositivo anfitrión) realiza la compilación de todos los tests mediante el compilador cruzado GNAT, y la segunda parte se lanza directamente en el dispositivo Android para llevar a cabo la ejecución de cada uno de los tests compilados previamente.

Este conjunto de tests está compuesto por más de 2300 tests individuales. Se han ejecutado en un teléfono Nexus 5 con Android 6.0 donde todos los tests han pasado satisfactoriamente. Por lo tanto, podemos confirmar que es posible utilizar programas escritos en el lenguaje de programación Ada sobre el sistema operativo Android siguiendo las directrices descritas en el presente capítulo.

[3] https://www.istr.unican.es/androidrt/

4.5 Conclusiones

Para ejecutar aplicaciones con requisitos temporales en Android no basta únicamente con tener un sistema operativo con características de tiempo real, sino que también se necesita que el sistema soporte lenguajes de programación de tiempo real. En el caso del sistema operativo Android, tal y como hemos visto a lo largo de este capítulo, por defecto, presenta limitaciones para utilizar programas que utilicen funciones necesarias en las aplicaciones con requerimientos temporales. Por lo tanto, se ha propuesto una solución portable y fácilmente aplicable para utilizar el lenguaje de programación C con la librería tradicional *glibc* en Android.

A pesar de que el lenguaje C es uno de los más extendidos para el desarrollo de sistemas de tiempo real existen alternativas como Ada que ofrecen ventajas sobre C en el desarrollo de programas concurrentes con requisitos de tiempo real. Ada es un lenguaje con una serie de características (programación por contrato, datos fuertemente tipados, primitivas avanzadas para la sincronización...) que facilitan la creación de software fiable. Por lo tanto, tener la posibilidad de utilizar este lenguaje de programación en un sistema operativo como Android ofrece ventajas evidentes para el desarrollo de aplicaciones con requisitos temporales.

A través de la propuesta que hemos mostrado en el presente capítulo es posible utilizar Ada en Android con software libre y de código abierto sin necesidad de realizar ningún cambio en el sistema operativo. Por lo tanto, es una solución válida para cualquier dispositivo y versión de Android.

5 Sincronización no bloqueante entre aplicaciones de tiempo real y no tiempo real en dispositivos multinúcleo

> *Se cometen muchos menos errores usando datos incorrectos que no empleando dato alguno.*
>
> **— Charles Babbage**
> Matemático y científico de la computación británico.

Como ya hemos mencionado en el Capítulo 3 de la presente tesis, en los últimos años la mejora de los procesadores se ha llevado a cabo a través de un incremento en el número de núcleos. Por este motivo la mayoría de fabricantes actuales para dispositivos móviles ofrecen procesadores con varios núcleos. Con esta clase de procesadores varios procesos o hilos se pueden ejecutar simultáneamente en diferentes núcleos para reducir de forma significativa los tiempos de computación en el sistema. Esto ayuda a maximizar el rendimiento de los dispositivos móviles.

Con la ejecución simultánea de diversos procesos o hilos, nace la necesidad de tener mecanismos de comunicación entre los procesos (*inter-process communication (IPC)*, en inglés). Existen diferentes mecanismos que son utilizados para realizar dicha comunicación [84] [85], tales como el paso de mensajes, la memoria compartida o las llamadas a procedimientos remotos. El objetivo principal de cualquiera de estos mecanismos es transferir y compartir datos entre procesos o hilos. No obstante, se necesitan mecanismos de sincronización para asegurar que el acceso concurrente a los datos se realiza de forma correcta sin generar errores.

Puesto que muchas de las aplicaciones de tiempo real están compuestas por procesos e hilos que se ejecutan concurrentemente, los desarrolladores de estas aplicaciones tienen que utilizar mecanismos de sincronización para poder intercambiar información correcta entre los distintos procesos e hilos. Además, cada vez es más habitual encontrarnos con sistemas que permiten

la ejecución de aplicaciones heterogéneas en el mismo dispositivo. Esto significa que en el mismo sistema pueden coexistir aplicaciones de tiempo real y de no tiempo real que pueden tener la necesidad de compartir datos entre ellas. Precisamente, en el Capítulo 3 presentamos una solución que permite tener aplicaciones con distintos requisitos temporales ejecutándose en dispositivos multinúcleo bajo el sistema operativo Android/Linux. Por lo tanto, el objetivo de este capítulo es el desarrollo de mecanismos de sincronización que permitan compartir datos entre aplicaciones de no tiempo real y tiempo real. Además, con el objetivo de que las aplicaciones con requisitos temporales no se vean afectadas por el acceso o modificación de los datos desde la parte de no tiempo real pretendemos que estos mecanismos de sincronización sean no bloqueantes.

El presente capítulo de esta tesis se organiza del siguiente modo; en la Sección 5.1 se comienza describiendo cuáles son los mecanismos que pueden ser utilizados para compartir datos entre tareas en un sistema operativo como Android. Posteriormente, en la Sección 5.2 se discute sobre los diferentes mecanismos de sincronización existentes cuando se accede de forma concurrente a datos compartidos, y cuáles se usan habitualmente en entornos de tiempo real. En la Sección 5.3 exponemos cuáles son las necesidades y requisitos que queremos cubrir cuando se comparten datos de forma concurrente en un sistema con mezcla de tareas con y sin requisitos de tiempo real. En la Sección 5.4 se detallan los cinco protocolos de sincronización no bloqueantes desarrollados para compartir datos. A continuación, en la Sección 5.5 se muestran los resultados obtenidos tras evaluar los protocolos de sincronización no bloqueante sobre un dispositivo con el sistema operativo Android. Y, finalmente, en la Sección 5.6 se desarrollan las conclusiones obtenidas en este capítulo.

5.1 Compartir datos entre tareas

Cuando se necesita compartir datos entre distintas tareas que se están ejecutando simultáneamente en un mismo sistema existen distintos mecanismos disponibles. Algunos estudios [86] han evaluado mecanismos como los *sockets*, las *pipes*, y la memoria compartida para determinar cuál es el mecanismo más rápido compartiendo datos de forma concurrente, llegando a la conclusión de que la memoria compartida es el que mejores tiempos de respuesta arroja, ya que con ella los procesos comparten el mismo segmento de memoria y, por lo tanto, se evita la copia de datos entre los procesos y el kernel. Además, el acceso a este tipo de memoria se realiza a través

de referencias o punteros, lo que facilita bastante su uso desde cualquier programa.

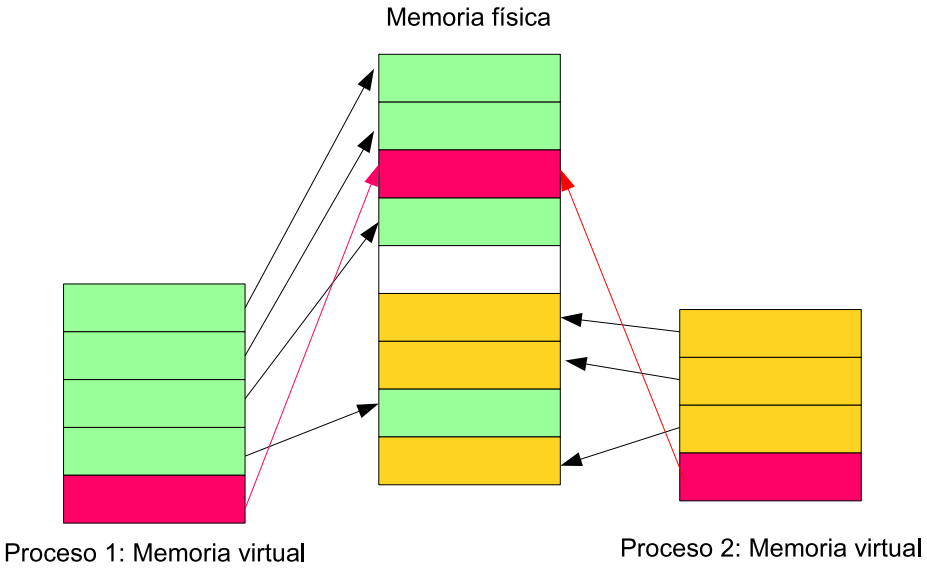

Fig. 5.1: Memoria compartida entre dos procesos. Los procesos en su memoria virtual tendrán un área mapeada en la memoría física que será accesible por ambos procesos.

La memoria compartida técnicamente es un espacio de memoria asociado al espacio de direcciones de los procesos que necesitan intercambiar datos entre ellos. Tal y como se ilustra en la Fig. 5.1, si tenemos dos procesos que necesitan compartir datos, en sus zonas de memoria virtual existirá un área que se habrá mapeado sobre el mismo área de memoria física y que es accesible por ambos procesos.

En los sistemas operativos basados en Linux la memoria compartida es una característica soportada de forma nativa. Sin embargo, como veremos a continuación existen diferentes alternativas y no todas ellas están soportadas en Android de forma nativa.

5.1.1 Anonymous Shared Memory (ASHMEM)

Android no soporta de forma nativa la interfaz POSIX para asignar y gestionar la memoria compartida y, por consiguiente, los desarrolladores de Google crearon un mecanismo para este sistema operativo similar a la memoria compartida de POSIX pero con el objetivo de ser muy eficiente en dispositivos con poca memoria. Este mecanismo se denomina *Anonymous Shared Memory (ASHMEM)* y permite crear regiones de memoria compartida bajo una entrada en un fichero en */dev/ashmem*, aunque no están disponibles para usar directamente desde las aplicaciones Android escritas en Java o Kotlin. A

diferencia de otros mecanismos para compartir memoria, con *ASHMEM* la región de memoria compartida se borra automáticamente cuando se cierra el último descriptor de fichero que apunta hacia ella. Además, el sistema puede modificar el tamaño del área de memoria utilizado por *ASHMEM* en función de las necesidades específicas en un momento dado. Sin embargo, algunos estudios [87] no aconsejan su uso directo por la documentación tan escasa que existe, y por considerar que podría haber algún cambio de diseño o implementación en un futuro que afectase a su comportamiento.

5.1.2 Ficheros mapeados en memoria

Un mecanismo para compartir memoria entre procesos consiste en utilizar un fichero. Para mejorar el rendimiento, el fichero se puede mapear en la memoria virtual. Además, dicho fichero se puede mapear en el espacio de direcciones de diferentes procesos para que sea accesible a través de un puntero aritmético.

Hemos realizado un test para medir el rendimiento de este mecanismo. En él se ha compartido un fichero que contiene un entero de 32 bits que ha sido mapeado en un segmento de memoria compartida. Además, se han utilizado dos procesos: uno de ellos modificaba el número entero y el otro leía el valor de dicho entero. Estos procesos se han ejecutado en un núcleo aislado de un procesador de un teléfono Nexus 5 que ejecuta Android. Después de analizar los tiempos de respuesta obtenidos hemos detectado que cada 4096 bytes escritos en el segmento de memoria compartido se produce un volcado de los datos hacia la memoria secundaria (flash) del dispositivo, este hecho ha provocado que el peor dato observado haya ocurrido tras la modificación de 1024 enteros (4096 bytes). En la Tabla 5.1 hemos ilustrado los tiempos de respuesta observados en la lectura del entero de 32 bits, donde el peor tiempo es provocado por el volcado de los datos a la memoria principal.

Fichero mapeado en memoria	
Primera lectura	2879 ns
Tiempo medio	501 ns
Peor tiempo de respuesta	14115 ns

Tabla 5.1: Tiempos de respuesta obtenidos en 10000 lecturas de un entero utilizando un fichero compartido mapeado en memoria desde un núcleo aislado del procesador.

5.1.3 Memoria compartida POSIX

La interfaz POSIX proporciona una serie de funciones que nos permiten tener segmentos de memoria compartida entre procesos. Los objetos compartidos se pueden mapear concurrentemente en el espacio de direcciones de diversos procesos. Ninguna de estas funciones se encuentra en la librería *Bionic* usada en Android, pero sí que se encuentran dentro de la librería tradicional *glibc*, la cual puede utilizarse en Android siguiendo las directrices descritas en el Capítulo 4 de esta tesis. Para utilizar la función que abre un segmento de memoria compartida (*shm_open*) se debe deshabilitar la capa de seguridad llamada SELinux (*Security-Enhanced Linux*) que Android tiene activada por defecto en sus últimas versiones. SELinux proporciona una protección en el acceso de las aplicaciones Java a los ficheros alojados en el directorio */dev*. Desde Android 4.2, SELinux se usa para definir los límites de las aplicaciones de Android y, por lo tanto, si lo desactivamos temporalmente podemos comprometer la seguridad del sistema en su conjunto. Por este último motivo descartamos su uso para compartir memoria en Android.

5.1.4 tmpfs

El sistema de ficheros temporales (tmpfs) permite almacenar ficheros en memoria virtual. Los ficheros creados utilizando este mecanismo tienen la apariencia de un sistema de ficheros tradicional montado en disco, pero realmente se almacena en la memoria principal del sistema. Debido a que los ficheros montados con tmpfs se mantienen en la caché del kernel no pueden ser retirados de la memoria principal. Gracias a esto, el acceso a ellos es significativamente más rápido que el acceso a la memoria secundaria.

tmpfs	
Primera lectura	573 ns
Tiempo medio	364 ns
Peor tiempo de respuesta	573 ns

Tabla 5.2: Tiempos de respuesta obtenidos en 10000 lecturas de un entero utilizando el mecanismo tmpfs desde un núcleo aislado del procesador.

Hemos realizado diversos tests para medir los tiempos de respuesta para acceder a un número entero de 32 bits utilizando este mecanismo, y hemos obtenido unos tiempos de respuesta que oscilan entre los 340 y 570 nanosegundos en nuestro entorno de pruebas (Nexus 5 con procesador de cuatro núcleos y Android 6.0). En la Tabla 5.2 se ilustran los tiempos de

respuesta observados al compartir un valor entero entre dos procesos que se ejecutan en un núcleo aislado del procesador. Por lo tanto, podemos concluir que este mecanismo nos proporciona mejores tiempos de respuesta y es más predecible para ser utilizado en aplicaciones con requisitos de tiempo real. Además, es el más adecuado para utilizar entre aplicaciones nativas de tiempo real (por ejemplo, Ada o C) y las aplicaciones Java, ya que está soportado nativamente en Android.

5.2 Sincronización

A pesar de que tengamos mecanismos para intercambiar datos entre distintos procesos o hilos en un sistema operativo, esto no implica que el acceso o modificación concurrente de dichos datos se haga de forma correcta. Para asegurar la consistencia de los datos es necesario que exista una sincronización.

5.2.1 Sincronización bloqueante

La manera más habitual de realizar la sincronización cuando se accede de forma concurrente a recursos compartidos es a través de la exclusión mutua. En ella se utilizan primitivas de sincronización como los mutexes, semáforos o secciones críticas, las cuales aseguran que algunas secciones del código de los programas serán ejecutadas con exclusividad. Si un proceso intenta acceder a un recurso compartido protegido mediante exclusión mutua que está siendo consultado o modificado por otro proceso se bloqueará hasta que el recurso se libere. Sin embargo, la exclusión mutua tiene algunos inconvenientes:

- Se pueden producir inversiones de prioridad, que consisten en que una tarea con alta prioridad puede verse bloqueada durante un tiempo no acotado esperando a que una tarea de menor prioridad libere el recurso compartido.

- Una tarea que necesita un recuso compartido puede tener que esperar hasta que otra que esté haciendo uso del recurso finalice. Esto provocará que la complejidad para calcular el tiempo de respuesta de peor caso aumente. Además, también se puede producir un retraso no deseado en otras tareas que dependan de la ejecución de la tarea bloqueada (*efecto convoy*).

- Cabe la posibilidad de que ocurran interbloqueos (*deadlocks*), es decir, que algunas tareas se mantengan en una espera infinita. Esto ocurre

porque dichas tareas necesitan que algunos recursos se liberen por otras tareas, que a su vez requieren acceder a recursos bloqueados por las tareas mencionadas inicialmente.

- Los numerosos cambios de contexto que pueden producirse con el uso de protocolos bloqueantes pueden reducir el rendimiento del sistema.

Los bloqueos infinitos o las inversiones de prioridad pueden ser evitados o mitigados con protocolos como el de herencia de prioridad o de techo de prioridad. Sin embargo, en los sistemas con procesadores multinúcleo el rendimiento cuando utilizamos este tipo de soluciones puede ser bajo debido a un nuevo tipo de inversión de prioridad denominado bloqueo remoto [88] [89]. Por este motivo han surgido a lo largo de los años algunas propuestas para planificar tareas y compartir recursos en sistemas de tiempo real con varios procesadores o núcleos. Algunas de las propuestas más relevantes son:

- *Multiprocessor Priority Ceiling Protocol (MCPC)* [90] cuyo objetivo principal es adaptar el protocolo de techo de prioridad a los multiprocesadores. En este protocolo se utiliza una planificación de prioridades fijas y las secciones críticas están protegidas con semáforos binarios modificados para operar según el nuevo protocolo.

- *Distributed Priority Ceiling Protocol (DPCP)* [91] [88] busca utilizar el protocolo de techo de prioridad en los multiprocesadores. Al igual que el protocolo MCPC utiliza prioridades fijas, pero los recursos compartidos globales están protegidos con semáforos que utilizan la operación atómica *read-modify-write*.

- *Multiprocessor Stack Resource Policy (MSRP)* [92] [93] considera, al igual que los protocolos *MCPC* y *DCPP*, que los recursos compartidos pueden ser locales (usados en un solo núcleo) o globales (usados desde diversos núcleos). El protocolo añade un nuevo comportamiento, ya que cuando una tarea no puede satisfacer un acceso a un recurso compartido no se suspende dicha tarea, sino que se lleva a cabo una espera activa hasta que se concede el acceso.

A pesar de que existen protocolos aplicables a sistemas con varias unidades de computo, cabe destacar que son soluciones que provocan un bloqueo en el avance de las tareas.

5.2.2 Sincronización no bloqueante

La sincronización no bloqueante es aquella en la que ninguno de los hilos implicados en la sincronización libera su procesador. Este tipo de sincronización se presenta como una alternativa a la exclusión mutua (especialmente en multiprocesadores), ya que facilita la ejecución concurrente de las partes del código que interactúan con los recursos compartidos. Por lo tanto, la sincronización no bloqueante se puede ver como una solución a los potenciales problemas asociados a los protocolos bloqueantes descritos en la subsección previa. Además, algunos estudios [94] [95] han demostrado que los protocolos no bloqueantes pueden mejorar el rendimiento de los sistemas. Aun así, el hecho de que las operaciones puedan realizarse concurrentemente implica una mayor complejidad para garantizar la consistencia y corrección de los datos compartidos y, por consiguiente, el diseño e implementación de protocolos no bloqueantes supone una dificultad añadida.

Se puede decir que tradicionalmente existen tres tipos de protocolos no bloqueantes: los conocidos como *obstruction-free*, *lock-free* y *wait-free*. Esta clase de protocolos normalmente hacen uso de operaciones atómicas como *CAS (Compare And Swap)*, *FAA (Fetch And Add)* o *TAS (Test And Set)*, es decir, operaciones cuya ejecución es indivisible e irreductible, ya que se realizan al completo sin ser interrumpidas.

Obstruction-free

Los protocolos denominados *obstruction-free* son los que ofrecen una menor garantía de progreso. Se definen como aquellos en los que en un momento dado concreto un proceso ejecutándose de forma aislada (todos los demás procesos se han detenido o suspendido) completará la operación en un número acotado de pasos.

Este tipo de protocolos no son la mejor opción para ser utilizados en sistemas heterogéneos donde conviven aplicaciones con y sin requisitos de tiempo real, ya que no sería posible garantizar que los procesos que están compartiendo recursos se ejecuten de forma aislada en el sistema para que puedan completar su operación en un número finito de pasos.

Lock-free

Los protocolos *lock-free* son aquellos que ofrecen la garantía de que al menos un proceso progresará y finalizará la operación sobre los recursos compartidos en un número acotado de pasos. Esta clase de protocolos se diseñan con la premisa de que los conflictos ocasionados por el acceso

simultáneo a los recursos compartidos ocurrirán de forma muy poco frecuente. Aun así deben ser capaces de identificarlos cuando ocurran para reiniciar la operación de acceso o modificación del recurso.

Para ilustrar cómo puede funcionar un algoritmo *lock-free* se exponen a continuación los pasos que debería seguir un proceso que necesita acceder o modificar un recurso compartido:

1. Se realiza una copia local del recurso que se vaya a actualizar o modificar.

2. Se lleva a cabo la actualización del recurso local.

3. A través de una operación atómica el recurso local se intenta convertir en el recurso compartido. Al hacerlo de forma atómica se garantiza que no se corromperá la integridad de los datos. En el caso que se detecte algún conflicto, se volverá al paso número 1 para repetir la operación hasta que no exista conflicto alguno.

Este tipo de protocolos acaban con los problemas de la exclusión mutua, aunque esto no implica que no tengan algunos inconvenientes que se deban considerar. Pueden producirse situaciones en las que se repitan las operaciones de forma continua, y por lo tanto, no se concluya nunca. Esto se conoce como inanición (*starvation*, en inglés).

La posibilidad de que exista inanición en alguna de las tareas que haga uso de mecanismos de sincronización *lock-free* provoca que no se recomiende su uso para entornos de tiempo real estricto, ya que podríamos encontrarnos con tiempos de respuesta no acotados. No obstante, hay que matizar que si se tiene suficiente información del sistema se podría estimar el número máximo de posibles repeticiones en las operaciones.

Wait-free

Los protocolos *wait-free* evitan que existan operaciones con un número no acotado de repeticiones. Esto significa que todas las operaciones siempre finalizarán en un número finito de pasos. Justamente esto es idóneo para los sistemas de tiempo real donde se busca que los procesos que operan sobre los recursos compartidos tengan una ejecución acotada, y que además finalicen en un número de pasos concreto (preferiblemente pequeño).

El desarrollo de esta clase de algoritmos entraña una gran dificultad. Algunos de ellos realizan una copia temporal de los elementos compartidos en diferentes espacios de memoria (*buffers*), de tal modo que las operaciones se realizan sobre dichos *buffers* evitando los conflictos provocados por la

ejecución concurrente. Al hacer esto el *buffer* con el elemento más reciente se anota como el válido a través de algún protocolo de sincronización.

Otra de las formas más habituales para diseñar e implementar algoritmos *wait-free* es la utilización de mecanismos de ayuda para las operaciones de acceso o modificación de los recursos compartidos (en inglés, *helping mechanism*) [96] [97]. En esta clase de mecanismos los procesos implicados, aparte de tratar de completar sus propias operaciones, ayudan a otros a progresar para facilitar su finalización. Para llevar a cabo la ayuda se puede utilizar un array con un elemento por cada proceso que esté usando las operaciones *wait-free* del protocolo. De este modo los procesos antes de llevar a cabo una operación sobre el recurso compartido lo anuncian y detallan a través de su correspondiente casilla del array. Otros procesos que vean el anuncio en dicho array pueden ayudar llevando a cabo la ejecución de la operación pendiente. Este tipo de mecanismos generalmente aumentan la complejidad de los algoritmos, ya que una misma operación se puede estar realizando por varios procesos de forma simultánea, y se debe comprobar que finalmente la operación solo se ha llevado a cabo una vez. Para realizar esta comprobación se tienen que ejecutar un gran número de operaciones atómicas que pueden degradar el rendimiento del protocolo de sincronización.

5.2.3 Memoria transaccional

La memoria transaccional [98] se presenta como una alternativa a la sincronización bloqueante y no bloqueante descritas anteriormente. En ella se introduce el concepto de *transacción* como una secuencia de instrucciones que se realizan especulativamente de forma atómica y aislada en un procesador. Estas transacciones se ejecutan en paralelo permitiendo que se realicen modificaciones en la memoria compartida sin tener en cuenta lo que estén realizando otras transacciones. Sin embargo, se lleva un registro de cada lectura y escritura, y si después de completar una transacción no se detecta que se hayan realizado cambios de forma simultánea en la memoria compartida se da la transacción por finalizada y confirmada. En cambio, si se detectan cambios la transacción se considera en conflicto y se tiene que volver a ejecutar desde el principio hasta que finalice correctamente. Por lo tanto, con esto lo que se pretende conseguir es que diversos procesadores puedan acceder concurrentemente y con garantías a los recursos compartidos sin que se necesite llevar a cabo un análisis global del sistema.

A lo largo de los últimos años se han propuesto muchas implementaciones de memoria transaccional, tanto en software [99] [100] [101] como en

hardware [102] [103] [104], e incluso implementaciones híbridas [105] o aceleradas por hardware [106]. Los sistemas de memoria transaccional implementados con la ayuda del hardware poseen un rendimiento superior al ofrecido por los diseños hechos en software. Sin embargo los fabricantes de hardware, por norma general, incorporan soporte básico y limitado para la memoria transaccional.

Los sistemas con memoria transaccional generalmente sufren de un impacto en el rendimiento debido al mantenimiento del registro de las transacciones y al propio tiempo necesario para finalizar cada una de las transacciones. Además, cuando haya transacciones grandes y tareas muy frecuentes podría haber inanición, ya que se daría un constante conflicto que implicaría la reversión al estado inicial de algunas transacciones.

5.2.4 Sincronización no bloqueante en sistemas de tiempo real

En los sistemas de tiempo real existe la necesidad de tener mecanismos de sincronización cuya respuesta temporal para compartir datos entre distintos procesos o hilos sea predecible, para así poder analizar la predictibilidad del conjunto de tareas que se están ejecutando.

Comúnmente la forma más utilizada para realizar la sincronización en esta clase de sistemas es a través del uso de mecanismos bloqueantes, de modo que tengamos la garantía de que la ejecución de las partes del código que modifican o acceden a recursos compartidos se haga en exclusividad. Sin embargo, como ya se ha descrito en la Subsección 5.2.1 y en otros estudios [107], este tipo de mecanismos pueden ocasionar situaciones no deseadas que desencadenen un bajo rendimiento, inversiones de prioridad o interbloqueos.

En las últimas décadas han aparecido numerosos estudios e implementaciones de protocolos no bloqueantes que intentan solucionar los potenciales problemas que posee la sincronización bloqueante. De forma generalista existen propuestas no bloqueantes para tener estructuras de datos *lock-free*. De hecho el estudio llevado a cabo por Barnes [108] muestra que cualquier estructura de datos puede ser transformada para ser utilizada con protocolos de sincronización *lock-free*. Además, existen diversas implementaciones que proporcionan librerías de estructuras de datos *lock-free* [109] [110] [111]. Las implementaciones que hacen uso de protocolos *lock-free* no ofrecen totales garantías temporales sobre algunas ejecuciones, ya que pueden necesitar repetirse si detectan conflictos. Esto significa que un proceso de tiempo real que acceda a un recurso compartido utilizando un algoritmo *lock-free* podría

tener un tiempo de respuesta no acotado salvo que se impongan limitaciones sobre la frecuencia de las operaciones protegidas.

Existen diseños e incluso algunas implementaciones de técnicas y librerías para la creación de protocolos *wait-free* [112] que podrían proporcionar una respuesta temporal acotada. No obstante, este tipo de protocolos son complejos, necesitan de un espacio de memoria extra y pueden generar un incremento en el tiempo de respuesta final.

Ninguno de los ejemplos citados en los párrafos anteriores está diseñado explícitamente para sistemas con requisitos de tiempo real, y menos aún para entornos donde pueden coexistir aplicaciones de tiempo real con otras aplicaciones sin requerimientos temporales. Aun así hay algunos estudios que han propuesto soluciones de sincronización no bloqueante para entornos de tiempo real. Existe un trabajo [113] que describe una implementación *lock-free* para el acceso simultáneo a datos compartidos, donde el sistema es monoprocesador y se hace uso de una primitiva atómica multipalabra de comparación e intercambio (MWCAS). Otro trabajo [114] propone un protocolo no bloqueante para un *buffer* de lectura y escritura compatible con múltiples lectores. En este caso el escritor nunca tiene que repetir la operación, ya que son los lectores los que deben verificar al final de cada lectura que no exista ningún conflicto, para así, terminar o en caso de que exista conflicto reiniciar la operación. Otro estudio [115] describe una solución no bloqueante para un *buffer* de lectura y escritura con soporte para múltiples escritores y lectores en sistemas multiprocesadores de tiempo real.

Existe un estudio que ha propuesto el uso de la memoria transaccional en sistemas de tiempo real [116]. Sin embargo en él se ha utilizado un modelo donde todas las tareas tienen requisitos temporales. Por lo tanto, actualmente no es válido para sistemas donde coexisten aplicaciones de tiempo real y no tiempo real.

El trabajo presentado en [113] utiliza una primitiva atómica multipalabra (MWCAS, *Multi-Word Compare-And-Swap*) para realizar la implementación de los algoritmos *lock-free*. Por un lado la operación atómica MWCAS no está presente en todas las arquitecturas hardware de forma nativa, y por otro lado cuando se usan protocolos *lock-free* en sistemas donde coexisten tareas con y sin requisitos temporales no se conoce con certeza los tiempos de respuesta de algunas de las aplicaciones, ya que si se pretende compartir datos entre aplicaciones de tiempo real y no tiempo real este tipo de protocolos podrían ocasionar tiempos no acotados. Sin embargo, las soluciones expuestas en los trabajos [114] y [115] son válidas para sistemas donde coexisten aplicaciones con distintos requisitos temporales, y concretamente pueden ser utilizadas,

respectivamente, con un escritor o con varios escritores de tiempo real y varios lectores de no tiempo real. En el presente trabajo vamos a proponer otros protocolos que cubran más situaciones habituales.

Como se ha ido mostrando en esta sección, existen multitud de trabajos sobre sincronización no bloqueante, pero no hemos encontrado ningún estudio práctico que explore los diferentes escenarios de productor/consumidor en un sistema con procesador multinúcleo donde coexisten aplicaciones de tiempo real y no tiempo real y comparten datos. Un ejemplo práctico donde es interesante tener protocolos de sincronización no bloqueante entre aplicaciones con distintos requisitos temporales puede ser una aplicación de tiempo real ejecutándose sobre un núcleo aislado del procesador. Esta aplicación necesitaría mostrar datos en pantalla o recabar información interactuando con otras aplicaciones con acceso a la interfaz gráfica o librerías complejas que no presentan garantías de funcionamiento en tiempo real. Por lo tanto, el objetivo del presente capítulo es desarrollar protocolos de sincronización no bloqueante que nos permitan obtener tiempos de respuesta acotados para las aplicaciones de tiempo real; de tal modo que estas puedan compartir datos con aplicaciones que no tienen requisitos temporales.

5.3 Enfoque del problema

5.3.1 Entorno arquitectural

Como vimos en la sección previa no hemos encontrado ninguna propuesta que satisfaga los escenarios productor/consumidor más habituales en un sistema donde coexistan aplicaciones de tiempo real y no tiempo real. Por ende, la solución buscada consistirá en tener diferentes protocolos no bloqueantes de sincronización asociados a estructuras de datos válidas para esos escenarios. Además, la parte de tiempo real no deberá realizar repeticiones en las operaciones de escritura o lectura, lo que permitirá tener tiempos de respuesta acotados. La parte de no tiempo real tendrá un comportamiento no bloqueante, aunque si durante el acceso o modificación de los datos compartidos detecta algún conflicto con la parte de tiempo real deberá reintentar la operación.

Nuestra propuesta completa se ilustra en la Figura 5.2. En ella se muestra un sistema con un procesador de cuatro núcleos donde coexisten aplicaciones de tiempo real y no tiempo real. Las aplicaciones con requisitos temporales se ejecutan en uno de los núcleos aislado con los mecanismos descritos en el Capítulo 3 de esta tesis. Aquellos hilos de tiempo real que necesiten

compartir datos con aplicaciones sin requisitos temporales lo harán a través de protocolos de sincronización no bloqueantes (por ejemplo, ver *Hilo 3* e *Hilo-TR 3* en la Figura 5.2).

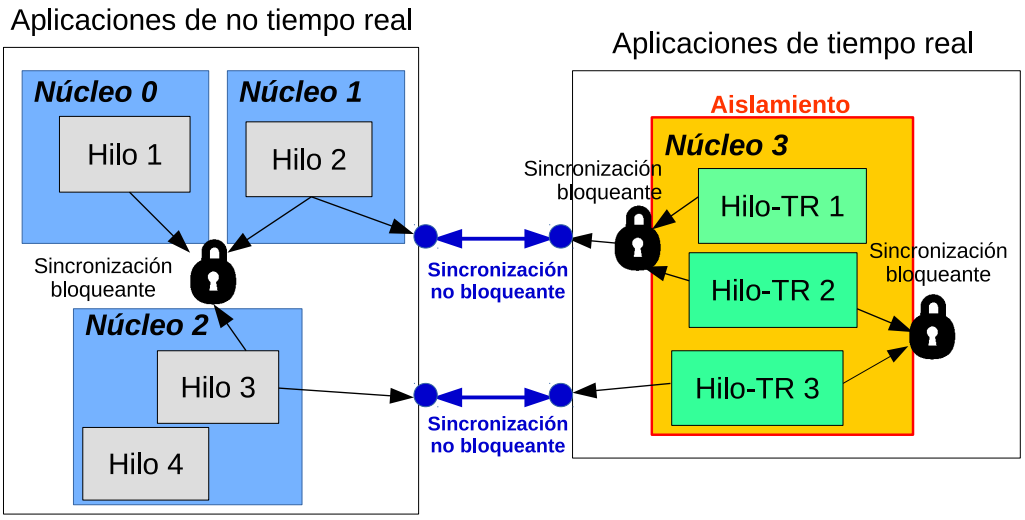

Fig. 5.2: Propuesta para la sincronización entre aplicaciones de tiempo real y no tiempo real.

Como el desarrollo y verificación de protocolos no bloqueantes es una tarea muy compleja, proponemos que la comunicación entre aplicaciones del mismo tipo (tiempo real y no tiempo real) se realice a través los mecanismos clásicos de exclusión mutua. Además, los protocolos que vamos a presentar en este capítulo solo permiten un hilo o proceso de tiempo real actuando como lector o escritor. Desarrollar protocolos no bloqueantes con múltiples lectores o escritores implicaría un aumento de la complejidad, una disminución del rendimiento y un mayor uso de memoria. Así que, optamos por una propuesta como la expuesta en la Figura 5.2, donde pueden existir varios hilos de tiempo real (*Hilo-TR 1* e *Hilo-TR 2*) que necesiten compartir datos con otro hilo de la parte de no tiempo real (*Hilo 2*). Para estos casos se serializa el acceso al algoritmo no bloqueante de sincronización mediante algún mecanismo de exclusión mutua (ver relación del *Hilo-TR 1* e *Hilo TR 2* en la Figura 5.2).

Un ejemplo real de un escenario donde nuestros protocolos podrían ser utilizados para compartir datos consiste en la adquisición de información a través de un sensor, procesado de dicha información y la muestra de los datos resultantes en una interfaz gráfica para el usuario. En este tipo de casos se necesita compartir uno o varios datos entre diferentes hilos usando algún patrón de productor/consumidor, donde la parte de tiempo real use un protocolo *wait-free*.

5.3.2 Identificación de protocolos de sincronización no bloqueante

Los protocolos de sincronización no bloqueantes se diseñan de tal modo que no precisen de secciones críticas, así que operan con copias locales de los recursos compartidos. Cuando por ejemplo un hilo que tiene el rol de escritor quiere actualizar algún dato compartido intentará colocar su copia local en el recurso compartido. Esta operación fallará si se detecta un conflicto con otro hilo que esté escribiendo o leyendo el mismo recurso compartido. En caso de que esto ocurra alguna de las operaciones tendrá que repetirse.

Las aplicaciones de tiempo real tienen que tener garantías temporales en el acceso a los recursos compartidos y, por ende, nunca deben repetir las operaciones de lectura o escritura que estén realizando. Sin embargo, las aplicaciones que no tienen requisitos temporales podrían repetir las operaciones tantas veces como fuese necesario hasta que los datos sean consistentes.

Teniendo en cuenta que las aplicaciones de tiempo real deben tener un comportamiento *wait-free* (siempre progresan sin repeticiones) hemos identificado los cuatro patrones productor/consumidor más comunes. Dos de estos patrones tienen la capacidad de compartir un dato, mientras que otros dos pueden compartir una cola de datos. Además, en estos patrones se ha limitado el número de lectores y escritores para simplificar la implementación y la verificación de los protocolos. A continuación se describen brevemente los cuatro patrones identificados:

- **SDrtR (shared data real-time reader)**: Dato compartido con un escritor sin requisitos temporales y un lector de tiempo real.

- **SDrtW (shared data real-time writer)**: Dato compartido con un escritor de tiempo real y múltiples lectores sin requisitos temporales. Se permite el uso de múltiples lectores debido a que en este caso concreto la complejidad de la implementación no aumenta.

- **QrtW (queue real-time writer)**: Cola circular con un escritor de tiempo real y un lector de no tiempo real. En este escenario hemos identificado dos posibles comportamientos cuando la cola está llena y un nuevo dato tiene que ser escrito.

    - **QrtWo (queue real-time writer with overwrite)**: Cuando la cola esté llena los nuevos datos sobrescriben los datos más antiguos.

- **QrtWc (queue real-time writer with clearing)**: Cuando la cola esté llena y un nuevo elemento necesite ser encolado todos los datos de la cola serán descartados.

- **QrtR (queue real-time reader)**: Cola circular con un escritor de no tiempo real y un lector de tiempo real.

Los escenarios identificados tienen una limitación en el número de lectores y escritores, sin embargo esta limitación proporciona garantías temporales para los tiempos de respuesta en la parte de tiempo real, e incluso es posible compartir datos entre varios lectores o escritores si usamos mecanismos de exclusión mutua para serializar el acceso a los algoritmos de sincronización no bloqueante (tal y como hemos descrito en la Subsección 5.3.1).

5.3.3 Propiedades de la solución

Los protocolos no bloqueantes para sincronizar y compartir datos desarrollados en el presente capítulo tienen que satisfacer tres propiedades:

- **Parte de tiempo real sin bloqueos o repeticiones**: En un sistema donde hay aplicaciones de tiempo real es estrictamente necesario tener sincronización con un comportamiento *wait-free* para las aplicaciones con requisitos temporales, de tal modo que no necesiten repetir las operaciones de escritura o lectura. Por lo tanto, el tiempo de peor caso para las aplicaciones de tiempo real no se verá afectado negativamente por contenciones en los recursos compartidos ocasionados por la parte de no tiempo real que acceda a dichos recursos.

- **Integridad de los datos**: Se necesita que los protocolos garanticen la integridad de los datos compartidos tanto en las operaciones de escritura como de lectura. Las aplicaciones sin requisitos temporales deberán verificar en cada una de las operaciones si los datos son válidos o no.

- **Uso de operaciones atómicas disponibles en procesadores x86, x86-64 y ARM**: Para desarrollar protocolos de sincronización no bloqueantes en muchos casos se necesita hacer uso de operaciones atómicas que nos garanticen que su ejecución es indivisible. En algunos de los protocolos que se exponen en este capítulo hemos utilizado la operación *comparar e intercambiar*, también llamada *CAS* (del inglés *Compare And Swap*). La operación *CAS* compara el contenido de una variable con otro valor esperado pasado por parámetro, si los valores

coinciden, se intercambia el valor de la variable con el de la otra (ver Código 5.1). Esta operación atómica está disponible tanto en procesadores Intel (x86 y x86-64) como en ARM, por lo tanto, es posible utilizarla en dispositivos Android.

```
1 function cas(p : pointer to int, old : int, new : int)
2     if *p != old
3         return false
4     endif
5     *p := new
6     return true
```

Código 5.1: Pseudocódigo de la operación atómica *CAS* (*Compare And Swap*)

Además, se hace uso de otras dos operaciones atómicas llamadas *Fetch And Add* y *Fetch And Sub*. Ambas están disponibles en las arquitecturas x86, x86-64 y ARM. Estas operaciones de forma atómica incrementan o sustraen respectivamente el contenido de una localización de memoria con un valor dado por parámetro. En el Código 5.2 se muestra el pseudódigo de la operación atómica *Fetch And Add*. La operación *Fetch And Sub* tiene la misma lógica de funcionamiento y estructura que el Código 5.2, salvo que en la línea 3 la operación es una resta.

```
1 function FetchAndAdd(location : pointer to int, inc : int)
2     int value := *location
3     *location := value + inc
4     return value
```

Código 5.2: Pseudocódigo de la operación atómica *Fetch And Add*

5.4 Protocolos de sincronización no bloqueantes desarrollados

Para los patrones productor/consumidor descritos en la Subsección 5.3.2 hemos desarrollado cinco protocolos de sincronización no bloqueante. En la Figura 5.3 se ilustra la estructura usada en los protocolos desarrollados en el presente capítulo. Por sencillez de la imagen en las colas se almacenan datos enteros, aunque en la implementación real podrán ser datos de cualquier tamaño. Los detalles de cada uno de ellos se irán explicando en las secciones siguientes.

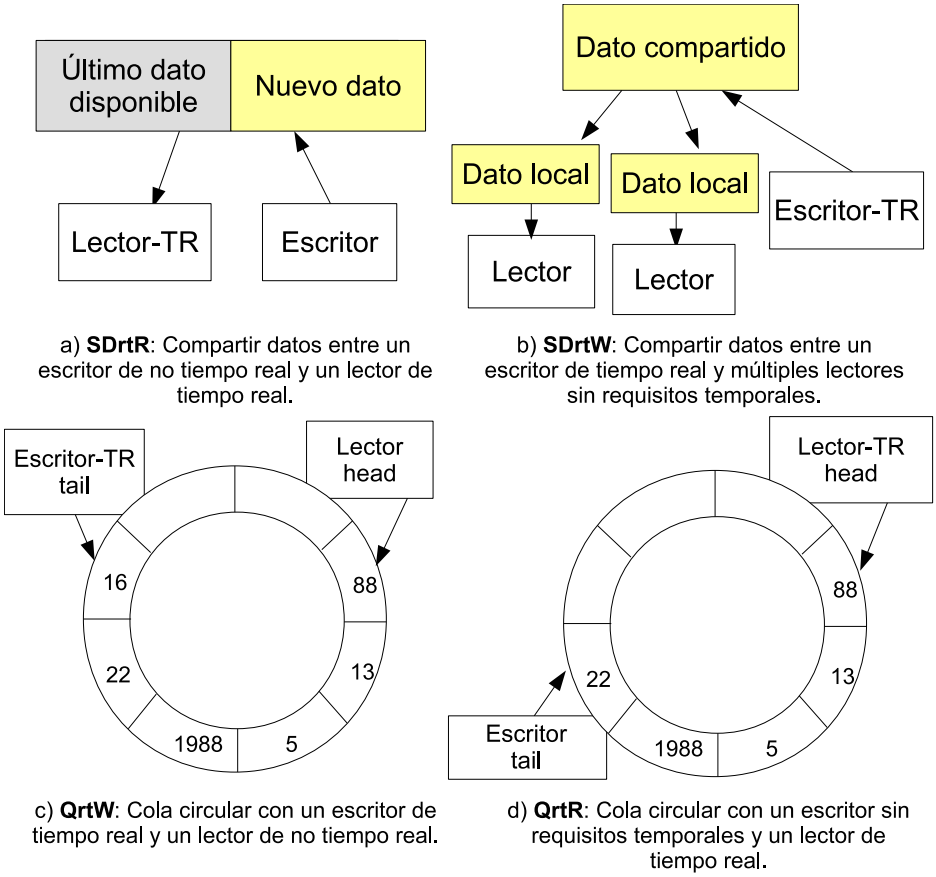

Fig. 5.3: Estructura de los protocolos de sincronización no bloqueantes para compartir datos desarrollados en este capítulo.

5.4.1 SDrtR: Compartir datos entre un escritor de no tiempo real y un lector de tiempo real

El protocolo que se va a describir está diseñado para sincronizar y compartir datos entre un lector con requisitos de tiempo real y un escritor sin restricciones temporales. El lector siempre tendrá acceso al último dato escrito válido sin la necesidad de bloquearse o repetir la operación. Sin embargo, en algunos escenarios el escritor necesitará reiterar la operación hasta que no exista conflicto con el lector que está accediendo a los datos.

Para conseguir que las operaciones se ejecuten de forma correcta se utiliza un doble *buffer*, de tal modo que el escritor nunca modificará el dato cuando el lector esté accediendo al *buffer*. Para llevar a cabo la implementación de este protocolo con doble *buffer* se han utilizado los elementos mostrados en la Figura 5.4. El doble *buffer* y el valor del índice (*pt_reader_index*) para identificar el *buffer* que se está leyendo son almacenados en memoria compartida accesible tanto para el lector como para el escritor. Además, el

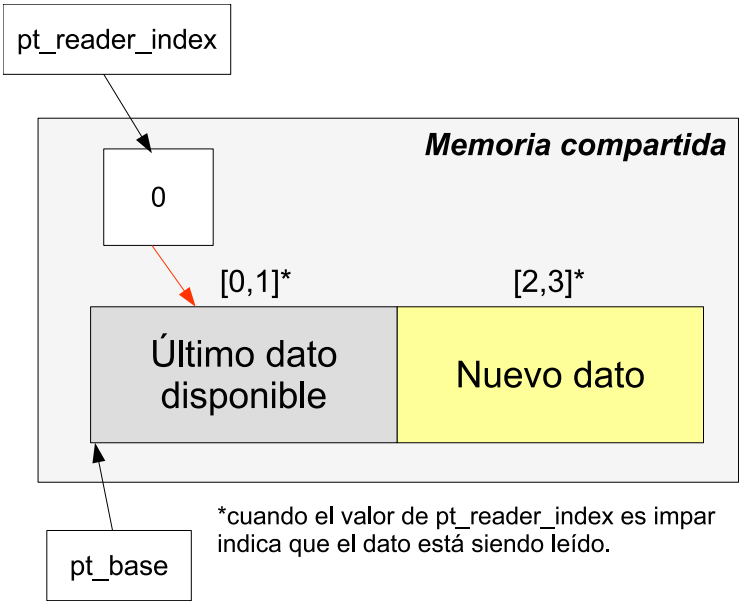

Fig. 5.4: Elementos usados en el protocolo *SDrtR*.

valor del índice puede identificar un *buffer* utilizando dos valores: 0 ó 1 para el primero y 2 ó 3 para el segundo. El uso de dos valores diferentes para identificar un *buffer* es necesario para saber si el lector está accediendo al *buffer*. Cuando el valor del índice es impar el lector está accediendo al dato y cuando el valor es par nos indica que no está siendo leído. Por ejemplo, en la Figura 5.4 el valor 0 de *pt_reader_index* indica que en ese momento el primero de los *buffers* es el que contiene el dato que puede ser leído, además al poseer un valor par nos indica que el lector no está accediendo al contenido del *buffer*.

A continuación se detalla a través de pseudocódigo el protocolo de sincronización no bloqueante para leer y escribir en un doble *buffer*:

```
1  data_size : unsigned int
2  pt_base : pointer to shared memory area of data_size*2 bytes
3  volatile pt_reader_index : pointer to shared memory area of
       unsigned int
4  volatile pt_has_data : pointer to shared memory area of
       unsigned int
5
6  function initialization (d_size: unsigned int)
7      data_size := d_size
8
9      pt_base := allocate data_size*2 bytes in the shared memory
10
```

```
11      volatile pt_reader_index : allocate unsigned int to store
            an index := 0
12
13      volatile pt_has_data : allocate unsigned int := 0
14
15  //Non-real-time operation
16  function write (pt_data: pointer)
17      //Obtain the index of the buffer to write
18      writer_index := 1 - (*pt_reader_index>>1)
19      //Expected value of the index when there is no read
            operation active
20      non_reading_index := *pt_reader_index & 0xFFFFFFFE
21
22      //Copy the new data to the write buffer
23      pt_next := pt_base + writer_index*data_size // pointer to
            the next buffer
24      copy data_size bytes from *pt_data to *pt_next*data_size
25      //Wait until no read operation is active to update the
            index
26      reader_new_index := writer_index*2
27      while (!CAS(pt_reader_index, non_reading_index,
            reader_new_index))
28          //Busy wait
29      endwhile
30      *pt_has_data := 1
31
32  //Real-time operation
33  function read (pt_data: pointer) return int
34      if (*pt_has_data == 0)
35          return NO_DATA
36      endif
37
38      //Atomic operation to start the reading
39      fetch_and_add (pt_reader_index, 1)
40
41      //Pointer to the last write buffer
42      pt_actual := pt_base +(*pt_reader_index>>1)*data_size
43      //Read the data
44      copy data_size bytes from *pt_actual to *pt_data*data_size
45
46      //Atomic operation to finish the reading
47      fetch_and_sub (pt_reader_index, 1)
```

```
48      return SUCCESS
```

Código 5.3: Pseudocódigo del algoritmo de sincronización no bloqueante para un doble buffer con un lector de tiempo real y un escritor sin requisitos temporales.

Para la implementación del protocolo se han utilizado un conjunto de datos en memoria compartida (ver líneas 1-4 del Código 5.3). Dichos datos son accesibles por el lector y el escritor y son inicializados a través de la función *initialization* que recibe como parámetro el tamaño de los datos que serán compartidos. Esta función debe ser llamada antes de empezar a utilizar las funciones *write* o *read*.

La operación *write* (líneas 16-30 del Código 5.3) utiliza una variable local denominada *writer_index*, la cual es un índice que identifica el *buffer* donde se va a escribir el nuevo dato. Para obtener el valor de esta variable local el contenido del puntero *pt_reader_index* es desplazado un bit hacia la derecha (es equivalente a realizar una división entera entre 2) obteniéndose un valor (0 ó 1) que identifica el *buffer* en el que se encuentra el último dato escrito y seguidamente se resta 1 menos el valor obtenido para conseguir un valor (1 ó 0) que identifica el *buffer* donde escribir el nuevo dato (ver línea 18 en el Código 5.3). En la variable local *non_reading_index* (línea 20 en el Código 5.3) se almacena el valor que la variable compartida *pt_reader_index* tomará cuando el lector de tiempo real ya no esté accediendo al dato (un valor par).

La variable *pt_next* declarada en la línea 23 es un puntero al *buffer* en el cual el nuevo dato tiene que ser escrito. La escritura del nuevo dato en el *buffer* libre se realiza en la línea 24.

Después, el nuevo valor del índice (correspondiente al *buffer* en el que se acaba de escribir el nuevo dato) se almacena en la variable *reader_new_index* (ver línea 26 del Código 5.3). Para cambiar el puntero *pt_reader_index* compartido entre el lector y el escritor se utiliza la operación atómica *CAS* (*Compare And Swap*). Esta operación solo cambiará el contenido del puntero *pt_reader_index* por el valor de *pt_reader_new_index* si es igual al valor de *non_reading_index*, en caso de que no sea así, el escritor debe reintentar la operación atómica hasta que se satisfaga (ver líneas 27-28 en el Código 5.3). Finalmente, con la variable *pt_has_data* se indica que se ha añadido algún dato al menos una vez.

La operación de tiempo real de lectura (*read*, líneas 33-48 del Código 5.3) lo primero que lleva a cabo es una comprobación para determinar si el escritor ha añadido al menos un primer dato (ver líneas 34-36 del Código 5.3), después realiza dos operaciones atómicas para asegurar la lectura del

último dato correcto disponible. La primera operación atómica (ver línea 39 del Código 5.3) incrementa el valor de *pt_reader_index* en una unidad para establecer un valor impar que indicará que el lector está accediendo al dato. La segunda operación atómica decrementa el contenido de *pt_reader_index* en una unidad para establecer un valor par, y así indicar que ya no se está accediendo al dato (ver línea 47).

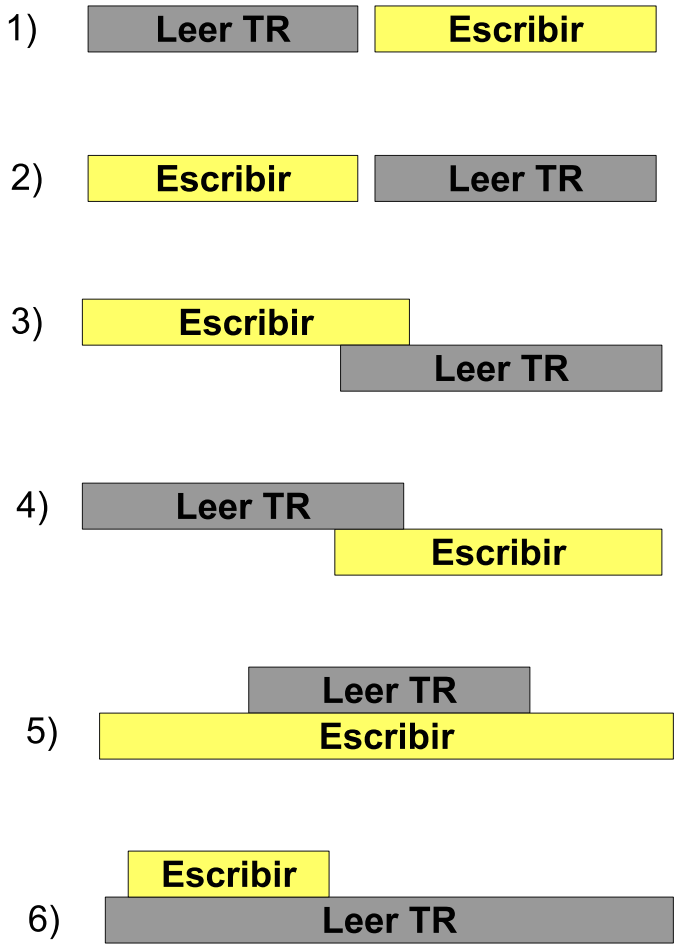

Fig. 5.5: Posibles relaciones temporales entre la operación de tiempo real de lectura y la de escritura en el algoritmo *SDrtR*.

Para proceder a la verificación del protocolo, se han identificado las instrucciones del pseudocódigo donde el puntero compartido *pt_reader_index* es accedido o modificado por el escritor y el lector. Estas instrucciones son todas aquellas que podrían ocasionar una inconsistencia en los datos si no se realizan correctamente. Las instrucciones de la parte del escritor son las siguientes:

- Línea 18: Lee *pt_reader_index*
- Línea 20: Lee *pt_reader_index*

- Línea 27: Lee y modifica *pt_reader_index*

Para la operación de lectura se han identificado las siguientes instrucciones:

- Línea 39: Lee y modifica *pt_reader_index*
- Línea 42: Lee *pt_reader_index*
- Línea 47: Lee y modifica *pt_reader_index*

La operación de escritura nunca acaba hasta que el valor de *pt_reader_index* es par y, cuando finaliza, lo cambia atómicamente a otro valor par. Por su parte, la operación de lectura mantiene *pt_reader_index* en un valor impar mientras está leyendo el dato. De esto se deduce que la operación de escritura nunca puede terminar mientras una operación de lectura se encuentre en la sección de código entre las dos llamadas atómicas (entre las líneas 39-47 del Código 5.3). Por otro lado, los dos accesos a la variable *pt_reader_index* del escritor en las líneas 18 y 20 nunca toman en cuenta el bit menos significativo (que es el modificado por el lector), por lo que el valor leído no se ve afectado por una posible ejecución concurrente del lector de tiempo real.

Para verificar la integridad de los datos cuando se ejecutan las operaciones del protocolo se ha usado el mismo razonamiento seguido por Kopetz [114]. En la Figura 5.5 hemos identificado siete posibles relaciones temporales entre el lector de tiempo real y el escritor.

La integridad de los datos está asegurada con las operaciones no concurrentes de las relaciones (1) y (2) de la Figura 5.5. La relación (3) solo puede ocurrir cuando el escritor ha finalizado la operación atómica de la línea 27 en el Código 5.3, ya que en caso de que el lector de tiempo real comenzase la operación antes de que el escritor finalizase la instrucción de dicha línea, este último tendría que mantenerse en la espera activa de la línea 28 hasta que el lector finalice. Así que en la relación (3) no existe ningún potencial problema porque la escritura del dato habría finalizado cuando el lector fuese a comenzar. En la relación (4) la integridad de los datos está garantizada porque el escritor no termina hasta que el lector cambie el contenido del puntero *pt_reader_index* con un valor par (línea 47); solo cuando eso ocurra el escritor modificará atómicamente el valor de *pt_reader_index* para que apunte al nuevo *buffer* escrito (línea 27). En la relación (5) el escritor solo podrá modificar el valor del puntero *pt_reader_index* cuando la operación de tiempo real de leer finalice y establezca un valor par en dicho puntero para indicar que no está accediendo al dato. Además este razonamiento sería válido para cualquier número de lecturas que ocurriesen durante la duración

de la operación de escritura. En la relación (6) el escritor solo podría finalizar antes que el lector si este último aún no ha modificado atómicamente *pt_reader_index* a un valor impar (ver línea 39); en caso contrario el escritor necesariamente tendrá que esperar hasta que el lector de tiempo real finalice (indicándolo con un valor par en el puntero compartido). De todo ello se concluye el correcto funcionamiento del algoritmo propuesto para todas las posibles ordenaciones del lector y el escritor.

5.4.2 SDrtW: Compartir datos entre múltiples lectores de no tiempo real y un escritor de tiempo real

Como ya hemos descrito en la Subsección 5.2.4 existen dos propuestas [114] [115] para compartir datos entre múltiples lectores sin requisitos temporales y un escritor de tiempo real. Se ha seleccionado la solución de Kopetz [114] por simplicidad y eficiencia. Además, el protocolo descrito en ese trabajo previo encaja perfectamente con nuestros requisitos, y consecuentemente solo se incluye en este capítulo por completitud.

En dicho estudio se describe una solución utilizable en sistemas de tiempo real donde lectores y escritor tienen requisitos temporales. El pseudocódigo del protocolo propuesto en [114] se muestra en el Código 5.4.

Para mantener la integridad de los datos, los lectores tienen que comprobar al final de cada lectura si el escritor ha modificado los datos compartidos. Para detectar si el escritor ha llevado a cabo alguna modificación sobre los datos se utiliza un contador global: la variable *counter*, un entero sin signo alojado en memoria compartida. El contador es incrementado en la operación de escritura de forma que toma un valor impar antes de modificar el dato compartido. Después de la modificación, el contador es incrementado de nuevo para tomar un valor par (ver líneas 14 y 17 en el Código 5.4). Si el lector verifica a través del contador global que el dato compartido no ha sido modificado (ver línea 25 en el Código 5.4), la operación de lectura finalizará, en caso contrario se repite.

La operación de escritura siempre tiene un tiempo de ejecución acotado, mientras que la de lectura puede necesitar realizar reintentos hasta que se encuentre que no existan posibles conflictos con el escritor.

En [114] los autores demuestran la corrección del protocolo, constatando que los lectores siempre obtienen un dato correcto. Además realizan un análisis temporal de las operaciones de escritura y lectura. En primer lugar, asumen que la duración de las operaciones es la misma, de tal modo que

determinan que una interferencia originada por un escritor podría ocasionar hasta tres repeticiones en una operación de lectura. En segundo lugar, aceptan que el tiempo mínimo entre sucesivas operaciones de escritura en el dato compartido es conocido: de no ser así no sería posible determinar el número máximo de interferencias. Con esta hipótesis formulan cuál sería el tiempo máximo de ejecución en una operación de lectura considerando reintentos.

```
1  volatile counter : pointer to shared memory area of an integer
2  data_size : unsigned int
3  pt_base : pointer to shared memory area of data_size bytes
4
5  function initialization (d_size: unsigned int)
6      counter := 0
7
8      data_size := d_size
9
10     pt_base := allocate data_size bytes in the shared memory
11
12 //Real-time operation
13 function write (new_value: pointer)
14     local_counter := *counter
15     *counter := local_counter + 1
16     copy data_size bytes from *new_value to *pt_base*data_size
17     *counter := local_counter + 2
18
19 //Non-real-time operation
20 function read (pt_data: pointer)
21     loop
22         counter_begin := *counter
23         copy data_size bytes from *pt_base to *pt_data
               *data_size
24         counter_end := *counter
25         if (counter_end == counter_begin & counter_begin is
               even)
26             break;
27         endif
28     endloop
```

Código 5.4: Pseudocódigo del protocolo de sincronización no bloqueante para múltiples lectores de no tiempo real y un escritor de tiempo real [114].

El protocolo encaja perfectamente con el escenario *SDrtW* descrito en la Subsección 5.3.2. En nuestro escenario no es posible realizar estimaciones temporales para los lectores (la parte de no tiempo real), puesto que se ejecutan en un entorno sin requisitos temporales en el que las interferencias por parte del resto del sistema no están acotadas.

5.4.3 QrtW: Cola circular con un escritor de tiempo real y un lector sin requisitos temporales

Se han desarrollado dos protocolos de sincronización no bloqueante utilizando una cola circular con un escritor de tiempo real y un lector sin requisitos temporales. El escritor puede encolar elementos sin bloquearse y sin reintentar la operación, es decir, se comporta como un protocolo *wait-free*. Cuando la cola está llena y un nuevo elemento necesita ser encolado se pueden implementar dos comportamientos. Uno consiste en sobrescribir el elemento más antiguo, y el otro descarta todos los elementos existentes (la cola se limpia). Teniendo esto en cuenta se han desarrollado los siguientes dos protocolos:

- **QrtWo (Queue real-time writer with overwrite):** Si la cola está llena y hay un nuevo elemento para encolar, se sobrescribe el elemento más antiguo.
- **QrtWc (Queue real-time writer with clearing):** Si la cola está llena y se necesita encolar un nuevo elemento, todos los datos de la cola se descartan.

Se ha decidido desarrollar los dos protocolos previos porque *QrtWc* es significativamente más simple que *QrtWo*. Sin embargo se pueden llegar a perder datos si el escritor de tiempo real añade elementos a la cola a un ritmo más rápido que el lector, y en este caso se recomendaría la utilización de *QrtWo*.

QrtWo: Queue real-time writer with overwrite

El protocolo denominado *QrtWo* utiliza una cola circular con una entrada vacía para así determinar cuándo está la cola llena o vacía. Además, hay dos variables para representar la cabeza con el elemento más viejo (*head*) y la cola con el elemento más reciente (*tail*). Por lo tanto, cuando la cola está llena la siguiente condición se debe satisfacer: $(tail+2)\ \%queue\_size == head$ y cuando la cola está vacía se cumplirá esta otra: $(tail+1)\ \%queue\_size ==$

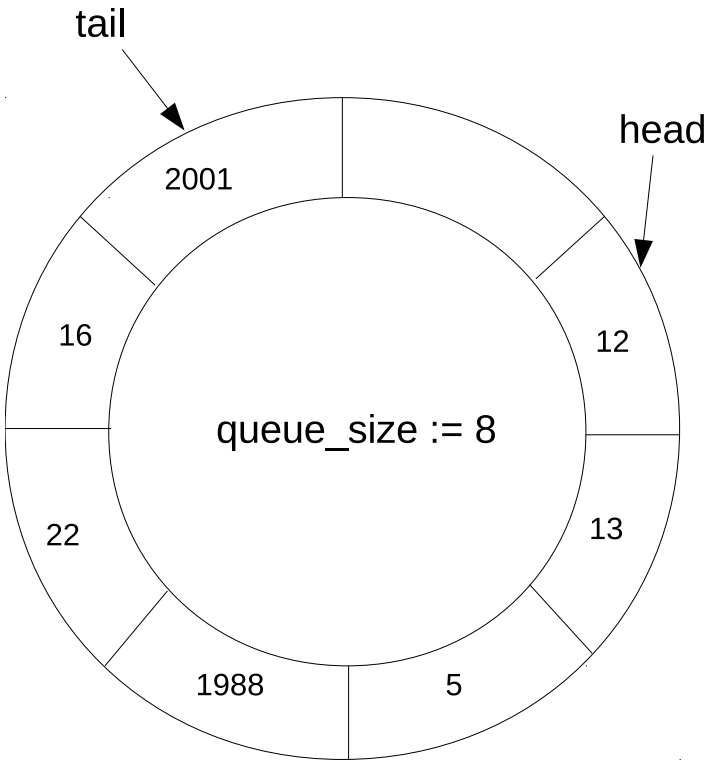

Fig. 5.6: Representación gráfica de las variables compartidas para el protocolo llamado *QrtWo*. En este escenario la cola está llena porque se cumple que *(tail+2) %queue_size == head*

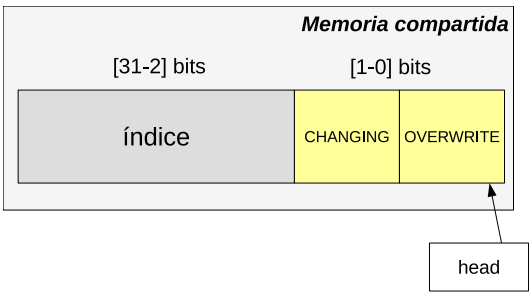

Fig. 5.7: Elementos usados dentro del índice *head* para el protocolo *QrtWo*

head. En la Figura 5.6 se ilustran los elementos descritos previamente para la estructura de la cola. Los datos representados en la Figura 5.6 son del tipo entero, pero en la implementación real se pueden utilizar de cualquier tipo y tamaño.

En este protocolo tanto la variable *head* como la variable *tail* están almacenadas en memoria compartida. Además, la variable *head* internamente contiene tres valores. Considerando que *head* es un entero de 32 bits, el primer bit menos significativo indica si se ha sobrescrito algún elemento en la cola, el segundo menos significativo sirve para saber si el escritor está modificando el valor del *head*, y finalmente los 30 bits más significativos son

usados para almacenar el índice dentro del array circular. Esto se ilustra en la Figura 5.7.

A continuación se detalla con pseudocódigo el protocolo *QrtWo*:

```
constant OVERWRITE = 0x01
constant CHANGING = 0x02
queue_size : unsigned int
data_size : unsigned int
volatile tail : pointer to shared memory area of an integer
volatile head : pointer to shared memory area of an integer
pt_queue : pointer to shared memory area of
    data_size*queue_size bytes

function initialization (d_size: unsigned int, q_size:
    unsigned int)
    volatile queue_size := q_size

    data_size := d_size

    volatile tail := queue_size-1

    volatile head := 0

    pt_queue := allocate data_size*queue_size bytes in shared
        memory

// Real-time operation
function enqueue (q_elem: pointer)
    unsigned int current_head := *head
    unsigned int local_head := current_head | CHANGING
    constant unsigned int head_not_overwrite := current_head
        & ~(OVERWRITE)
    constant unsigned int head_overwrite := current_head |
        OVERWRITE
    // pt_queue[*tail+1] := *q_elem
    copy data_size bytes from *q_elem to *pt_queue +
        ((*tail+1)%size)*data_size
    //Check if ovewrites the head of the queue
    if ((*tail+2) % queue_size == local_head>>2 && (CAS(head,
        head_not_overwrite, local_head) || CAS(head,
        head_overwrite, local_head)))
```

```
30          integer unsigned int new_head = ((local_head>>2)+1) %
                queue_size
31          new_head := (new_head<<2) | OVERWRITE
32          new_head := new_head & ~(CHANGING)
33          CAS(head, local_head, new_head)
34      endif
35      *tail = (*tail+1) % queue_size;
36
37  // Non-real-time operation
38  function dequeue (q_elem: pointer) return integer
39      if ((*tail+1) % queue_size == *head>>2)
40          return EMPTY
41      bool success := false
42      integer unsigned int new_head, local_head
43      do
44          local_head := *head
45          local_head := local_head & ~(OVERWRITE+CHANGING)
46          // *q_elem := pt_dequeue[head]
47          copy data_size bytes from *pt_queue +
                (local_head>>2)*data_size to *q_elem
48          new_head := ((local_head>>2)+1) % queue_size
49          new_head := (new_head<<2) & ~(OVERWRITE+CHANGING)
50          //Atomic operation
51          success := CAS(head, local_head, new_head)
52          if (success == false)
53              local_head := *head
54              int not_changing_head := local_head & ~(CHANGING)
55              new_head = local_head & ~(OVERWRITE+CHANGING)
56              //Atomic operation
57              CAS (head, not_changing_head, new_head)
58          endif
59      while(success==false)
60      return SUCCESS
```

Código 5.5: Pseudocódigo del protocolo de sincronización no bloqueante llamado *QrtWo*.

En el anterior protocolo el método de tiempo real llamado *enqueue* siempre comprobará si la cola está llena, y en caso de que así sea, el valor del bit llamado *changing* en la variable compartida *head* será modificado para indicar a la operación sin requisitos temporales (*dequeue*) que ese índice va a cambiar una posición (ver línea 29 del Código 5.5). Para que en todos los casos la

operación de tiempo real de encolar sea válida, en la línea 29 del Código 5.5 se tiene que establecer la variable compartida *head* a su valor completo correcto, es decir, que el bit denominado *changing* tenga el valor 1 para indicar a la operación de no tiempo real que no puede cambiar la variable *head* hasta que la operación de tiempo real finalice. Para conseguir este cambio se hace a través de dos posibles operaciones atómicas *CAS (Compare And Swap)* en la línea 29, tanto si el bit *overwrite* está a 1 o a 0 la operación de tiempo real modifica la variable *head* para que esta tenga exactamente el mismo valor completo que su variable local *local_head* (bit *changing* a 1). A continuación, se realizará la modificación de la variable compartida *head* avanzando una posición (línea 30), y finalmente se lleva a cabo la modificación de los bits *overwrite* y *changing* (ver líneas 31-33) para que así la operación de no tiempo real pueda avanzar.

El método sin requisitos temporales llamado *dequeue* tiene que verificar, después de acceder al elemento correspondiente apuntado por la variable *head*, si se ha producido una sobrescritura, o si se está llevando a cabo un cambio en la operación de tiempo real (ver líneas 51-52 en el Código 5.5). Si se detecta algún cambio en los bits *overwrite* y *changing* del índice *head*, la operación de desencolar se repetirá, aunque si solo detecta un cambio en el bit *overwrite*, este se pondrá a un valor 0 (ver líneas 53-57) y ya será en la siguiente repetición de la operación cuando la lectura se pueda completar (siempre y cuando no se vuelve a producir conflicto con el escritor de tiempo real).

En el protocolo desarrollado hay dos variables compartidas para las que tenemos que verificar cómo son modificadas y accedidas, de tal modo que se pueda verificar la corrección del protocolo. Estas variables son *head* y *tail*.

En el método llamado *enqueue* las variables compartidas son accedidas y modificadas en las siguientes líneas del Código 5.5:

- Línea 22: Leer *head*
- Línea 27: Leer *tail*
- Línea 29: Leer *tail* y modificar *head*
- Línea 33: Modificar *head*
- Línea 35: Leer y modificar *tail*

El método *dequeue* lleva a cabo modificaciones y accesos a las variables compartidas en las siguientes líneas:

- Línea 39: Leer *tail* y *head*

- Línea 44: Leer *head*
- Línea 51: Modificar *head*
- Línea 53: Leer *head*
- Línea 57: Modificar *head*

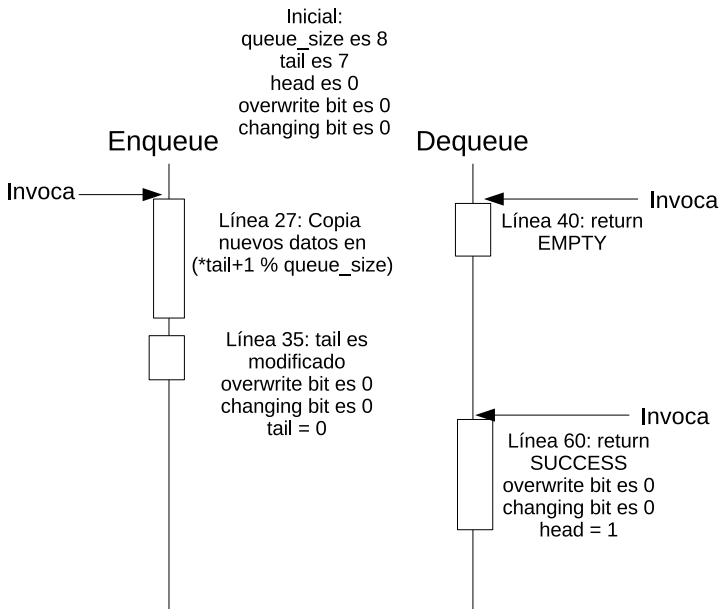

Fig. 5.8: *QrtWo*: Secuencia temporal de posibles operaciones cuando la cola no tiene elementos y se acaba de inicializar.

Todos las modificaciones de las anteriores variables compartidas se realizan a través de la operación atómica *CAS* (*Compare And Swap*). Además, el hecho de tener tres datos en una sola variable compartida (*head*) permite que se puedan modificar atómicamente en una sola operación. Por otro lado, los potenciales conflictos entre la operación de tiempo real y no tiempo real pueden ocurrir únicamente cuando la cola está llena o vacía, ya que en las demás situaciones las operaciones de encolado o desencolado no se interfieren entre sí. Por lo tanto, para demostrar que la integridad de los datos se mantiene durante la ejecución del protocolo, hemos identificado los escenarios donde las operaciones serían susceptibles de tener conflictos:

- En la Figura 5.8 se ilustra una posible secuencia temporal de operaciones. En este caso la cola está vacía y se acaba de inicializar, por lo que la primera llamada a *dequeue* finaliza inmediatamente al resultar verdadera la condición de la línea 39. Una vez el escritor encola un nuevo dato el lector de no tiempo real tiene un elemento disponible que ya puede desencolar.

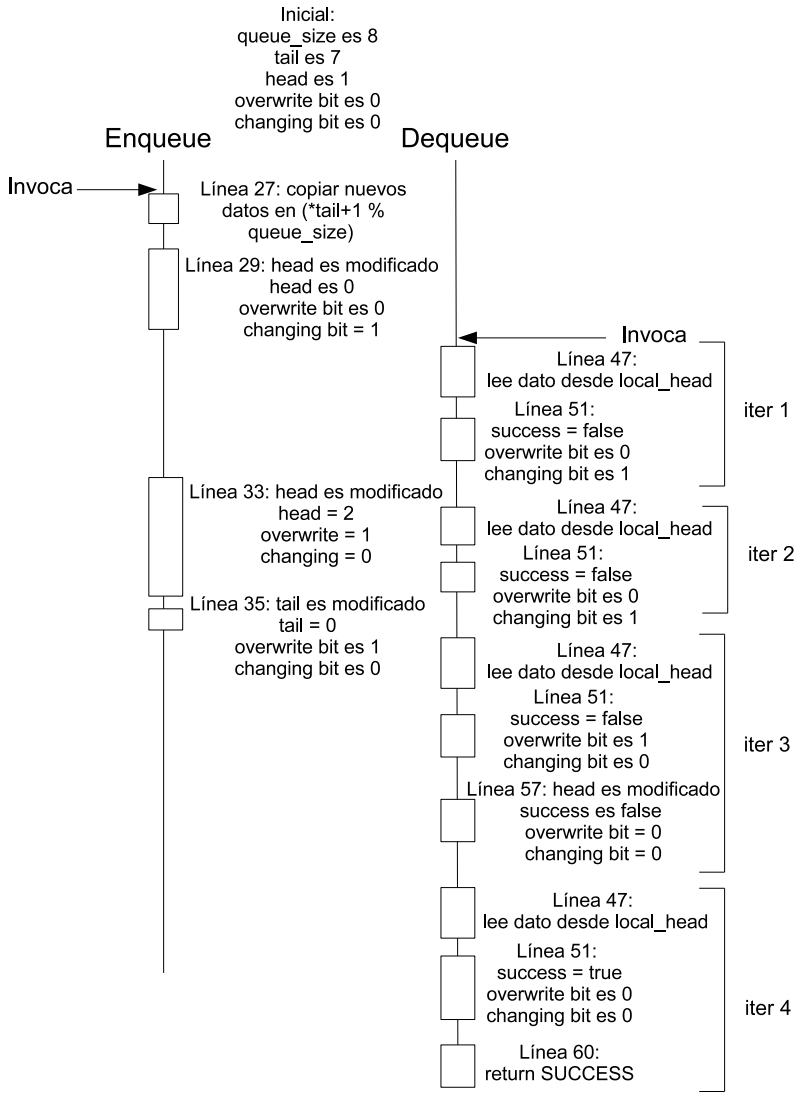

Fig. 5.9: *QrtWo*: Secuencia temporal de posibles operaciones cuando la cola está llena y se intenta encolar un nuevo elemento.

- La Figura 5.9 muestra una secuencia temporal donde la cola está llena y se intenta encolar un nuevo elemento. En la primera invocación del método *dequeue* no se puede desencolar ningún elemento porque el bit llamado *changing* de la variable *head* indica que el escritor de tiempo real está realizando una modificación. Cuando el escritor finaliza la modificación del valor de *head* el bit *changing* pasa a valer 0, indicando que ya se puede acceder. A continuación, el lector necesita dos iteraciones para desencolar el dato. En la iteración 3 de la Figura 5.9 detecta que se ha producido una sobrescritura a través del bit *overwrite*. Después, en la siguiente iteración, con los bits *changing* y *overwrite* con un valor igual a 0 puede desencolar el dato apuntado por la variable *head*.

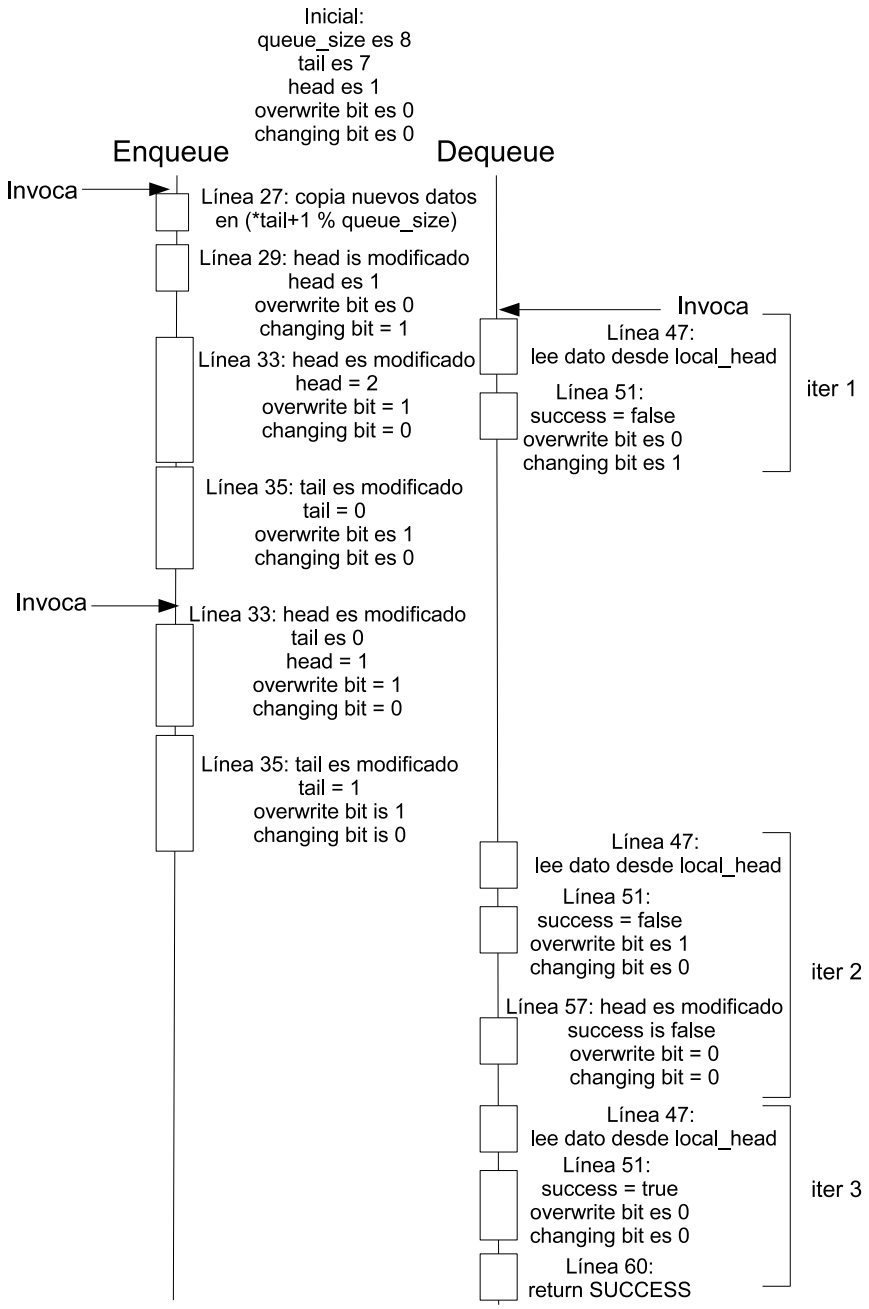

Fig. 5.10: *QrtWo*: Secuencia temporal de posibles operaciones cuando la cola está llena y se intenta encolar dos nuevos elementos consecutivos.

- La Figura 5.10 ilustra un escenario donde en una cola llena se intentan encolar dos nuevos elementos de forma consecutiva. Durante el encolado de los elementos el lector intenta obtener el dato apuntado por el índice de la variable *head*, pero en la primera iteración no es posible porque el bit llamado *changing* tiene un valor de 1. Cuando el escritor de tiempo real finaliza el encolado del segundo elemento, el

bit *changing* pasa a valer 0 y es cuando el lector puede acceder al dato apuntado por *head*.

De este análisis a través de los escenarios previos se puede concluir que el protocolo propuesto tiene un correcto funcionamiento para todas las posibles ordenaciones de las operaciones de encolado y desencolado.

QrtWc: Queue real-time writer with clearing

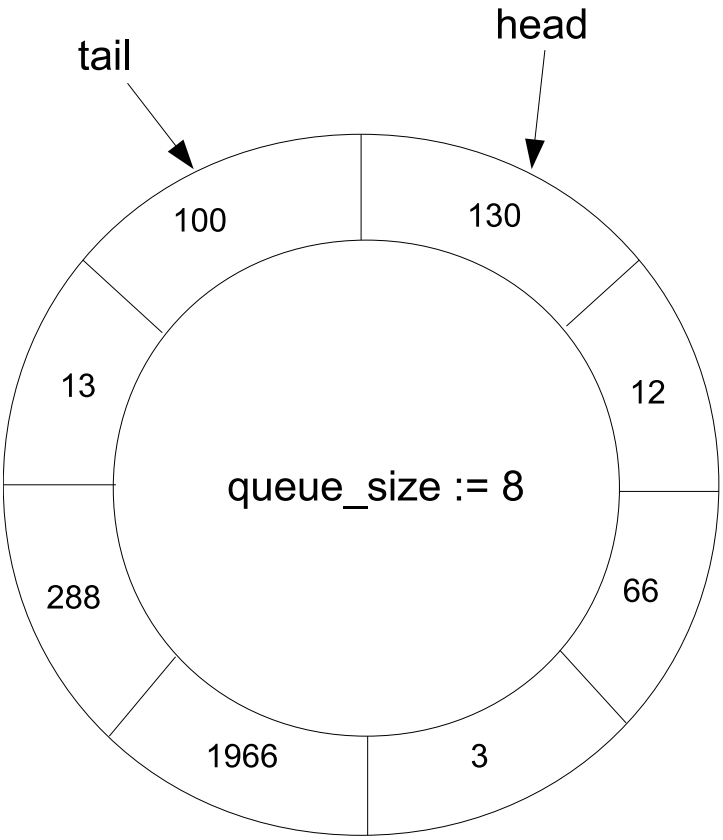

Fig. 5.11: Representación gráfica de las variables compartidas para el protocolo llamado *QrtWc*. En este escenario la cola está llena porque no hay ningún hueco disponible.

Se ha desarrollado otro protocolo en el que si la cola está llena y se intenta encolar un nuevo dato todos los datos actuales de la cola son descartados. El escritor nunca tendrá que esperar o repetir la operación ya que siempre tendrá un hueco disponible (vacío o descartando todos los elementos antiguos) en la cola para poder colocar el nuevo dato. En la Figura 5.11 se ilustra una cola llena y las dos variables para determinar el último elemento encolado (*tail*, variable compartida) y el siguiente elemento que debería ser desencolado (*head*, solo usado por el lector). En el escenario de la figura, la cola está llena,

por lo tanto, si el escritor necesita encolar un nuevo elemento se descartarán todos los datos previos.

Este protocolo se ha bautizado como *QrtWc* y se detalla en forma de pseudocódigo a continuación:

```
1  queue_size : unsigned int
2  data_size : unsigned int
3  volatile tail : pointer to shared memory area of an integer
4  volatile counter : pointer to shared memory area of an integer
5  pt_queue : pointer to shared memory area of
       data_size*queue_size bytes
6
7  function initialization (d_size: unsigned int, q_size:
       unsigned int)
8      volatile queue_size := q_size
9
10     data_size := d_size
11
12     volatile tail := queue_size-1
13
14     volatile counter := 0
15
16     pt_queue : allocate data_size*queue_size bytes in shared
           memory
17
18 // Real-time operation
19 function enqueue (q_elem: pointer)
20     *counter := *counter + 1
21     *tail := (*tail+1) % queue_size
22     // pt_queue[*tail+1] := *q_elem
23     copy data_size bytes from *q_elem to *pt_queue +
           (*tail%size)*data_size
24     *counter := *counter + 1
25
26 // Non-real-time operation
27 head : unsigned int := 0
28 function dequeue (q_elem: pointer) return integer
29     if ((*tail+1) % queue_size == head)
30         return EMPTY
31     loop
32         counter_begin := *counter
```

```
33          // *q_elem := pt_dequeue[head]
34          copy data_size bytes from *pt_queue + head*data_size
                to *q_elem
35          counter_end := *counter
36          exit when counter_begin == counter_end & counter_begin
                is even
37      endloop
38      head := (head+1) % queue_size
39      return SUCCESS
```

Código 5.6: Pseudocódigo para el protocolo de sincronización no bloqueante llamado *QrtWc*.

El anterior protocolo controla los posibles conflictos entre el lector de tiempo real y el escritor sin requisitos temporales utilizando un contador. Cuando el escritor está encolando un nuevo elemento incrementa el contador indicando que está modificando el dato, y cuando dicha modificación finaliza se vuelve a incrementar el contador para dejar un valor par (ver líneas 20 y 24 del Código 5.6). Además, en esta cola no es necesario dejar un hueco vacío, ya que para determinar si está vacía se comprueba la siguiente condición *(*tail+1) % queue_size == head*. En este protocolo la variable *head* solo es modificada por el método *dequeue*, mientras que la variable *tail* solo se modifica en el método de tiempo real *enqueue*. Del comportamiento anteriormente descrito se desprende que cuando la variable *tail* alcance el valor de *head* el método *dequeue* detecte la cola como vacía, y por lo tanto todos los elementos almacenados en ella son descartados (ver líneas 29-30 en el Código 5.6). El método *dequeue* usado por un lector sin requisitos temporales detecta cuándo se ha producido algún cambio en cualquiera de los elementos de la cola, pero no es capaz de discernir en qué elementos concretos ha ocurrido. Esto provoca que siempre sea necesario reintentar la operación de desencolado cuando esta se solapa con una operación de encolado (ver línea 36 del Código 5.6).

La Figura 5.12 ilustra un escenario donde podría existir un conflicto de no haberse realizado bien el diseño e implementación del protocolo. Un nuevo dato es encolado mientras desde el método *dequeue* se está leyendo el elemento apuntado por la variable *head*. Por lo tanto, el lector tiene que repetir la operación hasta que no detecte ningún cambio en el contador y además este tenga un valor par.

Existe una situación muy poco probable que ocurra pero hay que tenerla en cuenta. Si durante la lectura de un dato, esta se detiene, podría ocurrir que

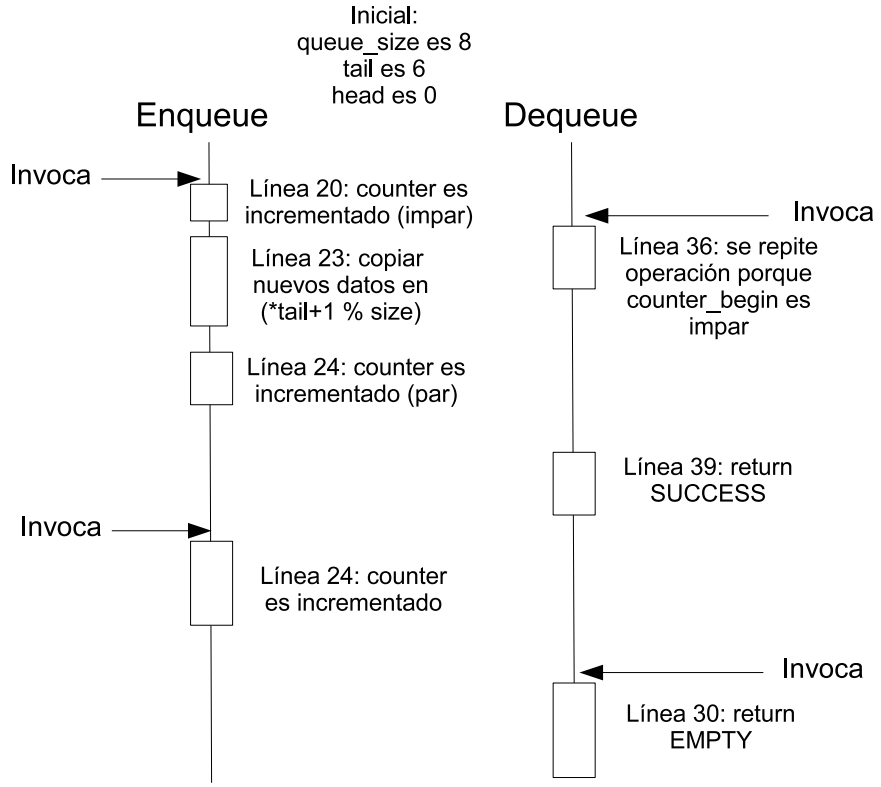

Fig. 5.12: *QrtWc*: Secuencia temporal de posibles operaciones cuando el lector intenta desencolar un elemento mientras el escritor está encolando un nuevo elemento.

el escritor se ejecutase más de 2^{32} veces ocasionando que el entero sin signo utilizado con la variable *tail* se desborde, quedándose con el valor esperado por el método *dequeue*. En ese caso el lector, cuando retome su ejecución, detectará que no se ha producido ningún cambio en el dato pero en realidad sí se ha producido. No obstante, si suponemos por ejemplo que que el escritor añade un nuevo dato cada 100 microsegundos, tendrían que transcurrir unas 119 horas para que se produzca la situación descrita anteriormente. Por lo tanto, se puede afirmar que es altamente improbable que esto ocurra.

Como se ha comentado anteriormente cuando la cola está llena en el protocolo *QrtWc* y un nuevo elemento es encolado, el lector en el siguiente intento de desencolar un elemento retornará que la cola está vacía (ver líneas 29-30 en el Código 5.6). Por lo tanto, el lector solo encontrará elementos disponibles cuando el escritor encole nuevos datos. Este hecho realmente provoca que todos los elementos existentes de la cola sean descartados.

Este protocolo tiene la ventaja de ser más simple que el descrito anteriormente (*QrtWc*), sin embargo su desventaja reside en que todos los elementos existentes son descartados cuando la cola se llena y se necesita sobrescribir datos.

5.4.4 QrtR: Cola circular con un lector de tiempo real y un escritor sin requisitos temporales

En este caso se ha implementado un protocolo de sincronización no bloqueante con una cola donde el lector tiene requisitos temporales. El lector de tiempo real obtendrá un dato sin bloqueos o repeticiones en la operación (siempre y cuando existan datos disponibles en la cola). Por otro lado el escritor será capaz de encolar elementos cuando no haya ningún conflicto con el lector de tiempo real. En caso de que se detecte un conflicto se tendrá que reintentar la operación hasta que finalice la operación de lectura.

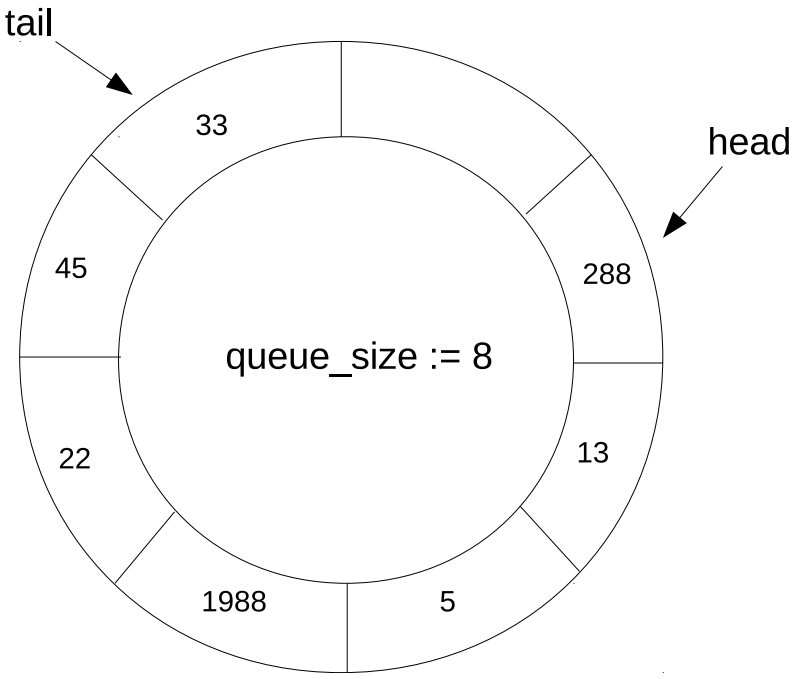

Fig. 5.13: *QrtR*: Representación gráfica de todos los elementos usados en el protocolo de sincronización no bloqueante.

La cola es circular y cuando está llena el escritor tendrá que esperar hasta que el lector desencole algún elemento para tener un hueco donde añadir nuevos datos. Además, este protocolo utiliza un hueco vacío para evitar tener una variable que contabilice el número de elementos añadidos en dicha cola. Tal y como se ilustra en la Figura 5.13 se utilizan dos variables compartidas para preservar la posición del último elemento encolado y para conocer el elemento que debe ser desencolado.

El protocolo desarrollado para este tipo de cola se detalla en forma de pseudocódigo a continuación:

```
1 volatile queue_size : unsigned int
```

```
volatile head : pointer to shared memory area of an integer
volatile tail : pointer to shared memory area of an integer
data_size : unsigned int
pt_queue : pointer to shared memory area of
    data_size*queue_size bytes

function initialization (d_size: unsigned int, q_size:
    unsigned int)
    volatile queue_size := q_size

    head := 0

    tail := queue_size-1

    data_size := d_size

    pt_queue := allocate data_size*queue_size bytes in the
        shared memory

// Non-real-time operation
function enqueue (q_elem: pointer) return integer
    if ((*tail+2) % queue_size == *head)
        return FULL
    new_tail : unsigned int := (*tail+1) % queue_size
    // pt_queue[new_tail] := *data_elem
    copy data_size bytes from *q_elem to *pt_queue +
        new_tail*data_size
    *tail := new_tail

// Real-time operation
function dequeue (q_elem: pointer) return integer
    if ((*tail+1) % queue_size == *head)
        return EMPTY
    // *q_elem := pt_queue[head]
    copy data_size bytes from *pt_queue + *head*data_size to
        *q_elem
    *head := (*head+1) % queue_size
    return SUCCESS
```

Código 5.7: Pseudocódigo para el protocolo de sincronización no bloqueante llamado *QrtR*.

El método denominado *enqueue* del anterior protocolo realiza una comprobación inicial para determinar si la cola está llena (ver línea 20 en el Código 5.7). Si la cola está llena el método retorna inmediatamente. Este comportamiento nos permite implementar una espera activa hasta que la cola tenga un hueco que haga posible el encolado de nuevos elementos. El método *dequeue* necesita comprobar si la cola tiene elementos disponibles (ver línea 29 en el Código 5.7), en caso de que existan elementos encolados se puede extraer uno e incrementar el valor de la variable *head*.

En este protocolo para evitar conflictos entre el escritor y el lector de tiempo real no se permite realizar sobrescritura de datos cuando la cola está llena, por lo tanto, la variable *tail* solo es modificada desde el método *enqueue*, y la variable *head* solo se actualiza en el método *dequeue*.

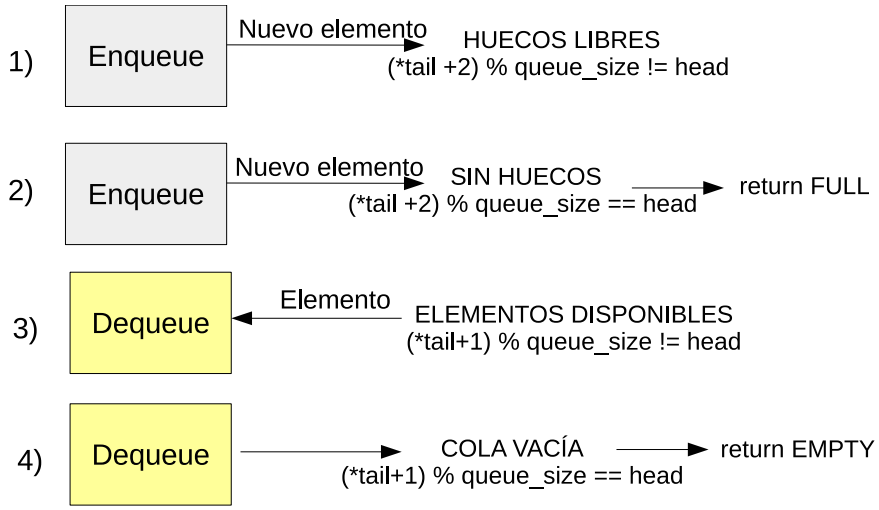

Fig. 5.14: *QrtR*: Escenarios posibles para los métodos *dequeue* y *enqueue* dependiendo del estado de la cola.

Para demostrar que los datos leídos o escritos son consistentes, en la Figura 5.14 se han identificado los cuatro posibles escenarios más representativos en el uso del protocolo. Cuando el método *enqueue* se encuentra con huecos disponibles no existe ningún potencial conflicto (caso 1 de la Figura 5.14). Si no existen huecos libres el método *enqueue* retorna una constante (caso 2 de la Figura 5.14). En el caso 3 de la Figura 5.14 no hay posibles conflictos porque el lector nunca detectará un elemento disponible hasta que el escritor finalice el encolado de un nuevo elemento. Finalmente, si la cola está vacía el lector inmediatamente retornará una constante (caso 4 de la Figura 5.14).

5.5 Evaluación de los protocolos desarrollados

Para evaluar los protocolos de sincronización no bloqueante desarrollados en este capítulo, hemos realizado una serie de tests que permiten medir los tiempos de respuesta del acceso o modificación a un dato compartido. Para esta evaluación se ha usado el sistema operativo Android 6.0 en un teléfono inteligente Nexus 5 que posee un procesador ARM Snapdragon 800 de cuatro núcleos y 2 GB de memoria RAM. Además, se han aplicados los mecanismos de aislamiento descritos en el Capítulo 3 de esta tesis, para así poder medir de forma aislada y sin interferencias los tiempos de respuesta.

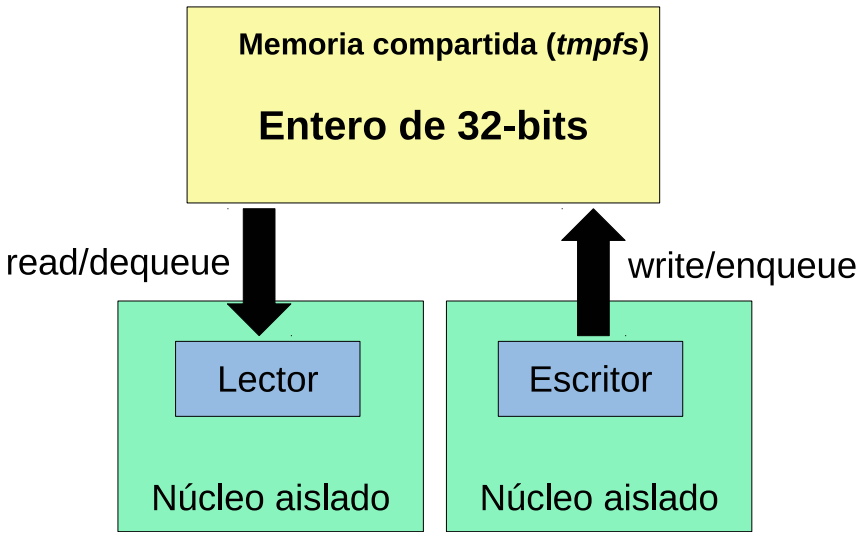

Fig. 5.15: Entorno donde se han ejecutado los tests para medir los tiempos de respuesta de los protocolos desarrollados.

Los tests llevados a cabo consisten en medir el tiempo de lectura o escritura de un entero de 32-bits en memoria compartida usando los protocolos no bloqueantes desarrollados. Todos los tests, tanto los que miden las operaciones de tiempo real como las que no, se realizan en núcleos aislados del procesador. Por lo tanto, se han aislado dos núcleos del procesador del teléfono Nexus 5 para ejecutar en paralelo la parte de tiempo real y no tiempo real de nuestros protocolos de sincronización no bloqueante (ver Figura 5.15). Además, se ha utilizado el lenguaje de programación C con un compilador cruzado para la arquitectura ARM y la librería tradicional *glibc* (usando el procedimiento para su utilización descrito en el Capítulo 4 de esta tesis), de tal forma que los tests se ejecuten directamente sobre el kernel y las librerías nativas del sistema operativo. Por otro lado, como los protocolos necesitan utilizar memoria compartida se ha usado el mecanismo *tmpfs*, el cual se

ha evaluado en la Sección 5.1 determinando que es el más idóneo para el sistema operativo Android.

En los tests realizados para obtener medidas relativas a los tiempos de respuesta se han calculado el tiempo medio y la desviación estándar tanto de la parte de tiempo real como de la de no tiempo real. Además, para la parte sin requisitos temporales se han contabilizado y evaluado aquellas operaciones que necesitaban ser repetidas debido a interferencias ocasionadas por la escritura o lectura de tiempo real.

5.5.1 Evaluación de los protocolos no bloqueantes para compartir datos

La tabla 5.3 ilustra las medidas obtenidas para el protocolo denominado *SDrtR* descrito en la Subsección 5.4.1. De forma análoga la tabla 5.4 muestra las medidas para el protocolo *SDrtW* descrito en la Subsección 5.4.2.

Tabla 5.3: Medidas para la lectura y escritura de un entero de 32 bits compartido utilizando el protocolo llamado *SDrtR*.

	Tiempo de respuesta de peor caso	Media	Desviación estándar	Observaciones
Lector (tiempo real)	29.430 μs	0.945 μs	0.325 μs	10^8
Escritor (no tiempo real sin repeticiones)	1.250 μs	0.662 μs	0.171 μs	10^8
Escritor (no tiempo real con 1 repetición)	0.938 μs	0.937 μs	0.004 μs	7
Escritor (no tiempo real con 2 repeticiones)	0.941 μs	0.926 μs	0.021 μs	9
Escritor (no tiempo real con 3 repeticiones)	1.286 μs	0.941 μs	0.032 μs	104
Escritor (no tiempo real con 4 repeticiones)	-	-	-	0

En los tests llevados a cabo el lector y el escritor se han ejecutado de forma periódica cada 100 microsegundos y muestran que la variabilidad temporal en la parte de tiempo real (la cual ha sido medida como la diferencia entre el máximo y el valor medio del tiempo de respuesta) no se ha visto incrementada por nuestros protocolos, ya que los peores tiempos de respuesta se producen

Tabla 5.4: Medidas para lectura y escritura de un entero de 32 bits compartido utilizando el protocolo denominado *SDrtW*

	Tiempo de respuesta de peor caso	Media	Desviación estándar	Observaciones
Escritor (tiempo real)	54.951 μs	0.624 μs	0.111 μs	10^8
Lector (no tiempo real sin repeticiones)	40.993 μs	0.636 μs	0.132 us	10^8
Lector (no tiempo real con 1 repetición)	1.146 μs	0.771 μs	0.130 μs	43
Lector (no tiempo real con 2 repeticiones)	1.354 μs	1.146 us	0.143 μs	90
Lector (no tiempo real con 3 repeticiones)	-	-	-	0

por las interferencias generadas en el sistema operativo (interrupciones no enmascarables e hilos del kernel).

En el caso de la parte de no tiempo real los tiempos de respuesta son variables en función del número de repeticiones que se requieren para completar la operación. Tanto en la tabla 5.3 como en la 5.4, los tests donde las repeticiones han ocurrido se han podido observar tras la ejecución de 10^8 iteraciones. Por lo tanto, podemos afirmar que las repeticiones ocurren con poca frecuencia y el impacto en los tiempos de respuesta no es muy significativo. Además, tanto el tiempo de respuesta medio como el peor caso son pequeños. Sin embargo, en la Tabla 5.3 el tiempo de respuesta para el escritor sin requisitos temporales tiene una variabilidad significativamente menor que la del lector de tiempo real. Esto es causado por las operaciones atómicas empleadas, ya que estas son las que más tiempo de ejecución utilizan en la plataforma hardware usada en las pruebas. Tal y como se ilustra en la tabla 5.5 la operación atómica *CAS* necesita 0.1 microsegundos para completarse, por lo tanto, como el lector de tiempo real usa dos operaciones atómicas y el escritor de no tiempo real solo una, se produce un mayor tiempo de ejecución para la parte de tiempo real, y por lo tanto, una mayor probabilidad de que las interrupciones no enmascarables o los hilos del kernel interrumpan la ejecución. Cabe recordar que los mecanismos de aislamiento utilizados en estas pruebas no son perfectos y ya medimos en el Capítulo 3 que podrían ocurrir interferencias de hasta unos 250 microsegundos. Por esta razón los

tiempos de peor caso son significativamente mayores que el tiempo medio. Por lo tanto, no son los protocolos no bloqueantes desarrollados los que provocan los tiempos de respuesta de peor caso. En consecuencia, los tiempos medios son los que nos proporcionan un valor más realista sobre el impacto de nuestros protocolos en los tiempos de respuesta.

Tabla 5.5: Tiempos medios para las instrucciones más relevantes de los protocolos de sincronización no bloqueante desarrollados. Las medidas han sido tomadas en un Nexus 5 realizando 10^6 iteraciones.

Instrucciones	**Tiempo medio**
Iteración lazo for	0.006 μs
fetch_and_add	0.099 μs
CAS (éxito)	0.102 μs
CAS (no se cumple)	0.102 μs
Asignación entero en memoria compartida	0.005 μs
Asignación variable local	0.005 μs

La tabla 5.3 muestra que se han observado hasta 3 repeticiones para el escritor de no tiempo real, y la tabla 5.4 ilustra que se han podido ver hasta 2 repeticiones para el lector de no tiempo real. En ambos casos, es el máximo de repeticiones que hemos conseguido observar para estos protocolos en las 10^8 iteraciones realizadas.

5.5.2 Evaluación de las colas circulares con los protocolos no bloqueantes

También se han evaluado los protocolos no bloqueantes desarrollados que utilizan una cola como estructura de datos para almacenar los datos a compartir. Tanto la tabla 5.6 como la tabla 5.7 muestran los tiempos de lectura y escritura para la manipulación de un entero de 32 bits almacenado en ese tipo de colas utilizando los protocolos *QrtWo* y *QrtWc* respectivamente. Del mismo modo, la tabla 5.8 ilustra las mismas medidas realizadas para el protocolo *QrtR*.

Los tiempos de respuesta de peor caso de las tablas 5.6 y 5.7 son menores cuando más repeticiones hay y esto es provocado por el número de observaciones para estos casos. Un número bajo de ejecuciones significa que existe una probabilidad muy pequeña de sufrir las interferencias ocasionadas por las interrupciones no enmascarables o los hilos del kernel del sistema. Tal

Tabla 5.6: Medidas para lectura y escritura de un entero de 32 bits compartido en una cola utilizando el protocolo denominado *QrtWo*

	Tiempo de respuesta de peor caso	Media	Desviación estándar	Observaciones
Escritor (tiempo real)	15.365 μs	0.647 μs	0.103 μs	10^8
Lector (no tiempo real sin repeticiones)	188.207 μs	0.681 μs	1.889 μs	10^8
Lector (no tiempo real con 1 repetición)	16.772 μs	1.553 us	1.947 μs	433
Lector (no tiempo real con 2 repeticiones)	1.927 μs	1.146 us	0.086 μs	94
Lector (no tiempo real con 3 repeticiones)	-	-	-	0

Tabla 5.7: Medidas para lectura y escritura de un entero de 32 bits compartido en una cola utilizando el protocolo denominado *QrtWc*

	Tiempo de respuesta de peor caso	Media	Desviación estándar	Observaciones
Escritor (tiempo real)	13.2213 μs	0.619 μs	0.104 μs	10^8
Lector (no tiempo real sin repeticiones)	26.591 μs	0.621 μs	1.52 μs	10^8
Lector (no tiempo real con 1 repetición)	129.856 μs	1.975 μs	0.882 μs	322
Lector (no tiempo real con 2 repeticiones)	-	-	-	0

Tabla 5.8: Medidas para lectura y escritura de un entero de 32 bits compartido en una cola utilizando el protocolo denominado *QrtR*

	Tiempo de respuesta de peor caso	Media	Desviación estándar	Observaciones
Lector (tiempo real)	11.874 μs	0.891 μs	0.199 μs	10^8
Escritor (no tiempo real sin repeticiones)	13.646 μs	0.883 μs	2.071 μs	10^8

y como ya hemos mencionado en la subsección previa estas interferencias ocurren porque los mecanismos de aislamiento utilizados con estos tests no pueden evitar todas las interrupciones e hilos del kernel.

En las tres tablas se observa que los tiempos de peor caso para las operaciones sin requisitos temporales son significativamente mayores con respecto al tiempo medio. Esto, como ya se ha explicado anteriormente, es debido a las interferencias causadas por los hilos del kernel y las interrupciones no enmascarables. De hecho, su impacto temporal es mayor que el de las repeticiones causadas por los propios protocolos. En cualquier caso, podemos afirmar que las operaciones de tiempo real obitenen tiempos de respuesta suficientemente acotados para utilizar con aplicaciones de control que posean plazos en el rango de los milisegundos.

5.6 Conclusiones

En este capítulo se han desarrollado protocolos de sincronización no bloqueantes para sistemas multinúcleo donde coexisten aplicaciones de tiempo real con aplicaciones sin requisitos temporales. Los cinco protocolos desarrollados cubren los casos básicos del modelo productor/consumidor.

En los protocolos implementados las aplicaciones de tiempo real tienen prioridad sobre aquellas que no tienen restricciones temporales, permitiendo así a las de tiempo real tener tiempos de respuesta acotados. La parte de tiempo real nunca repite las operaciones, es decir, concluye en un número finito de pasos acotado; si se detecta algún conflicto que ponga en peligro la integridad de los datos será la parte de no tiempo real la encargada de repetir la operación hasta que pueda manipular o acceder a los datos compartidos con seguridad. Para comprobar la viabilidad de los protocolos no bloqueantes desarrollados se han evaluado en un sistema operativo Android/Linux, donde dos núcleos han sido aislados para ejecutar de forma concurrente tanto la parte de tiempo real como la de no tiempo real. Los resultados de los experimentos demuestran que en este tipo de plataforma las aplicaciones de tiempo real responden con tiempos suficientemente acotados permitiendo su uso en entornos con restricciones temporales que necesiten plazos en el rango de los milisegundos.

Conclusiones y trabajo futuro

6

> *La vida es el arte de sacar conclusiones suficientes a partir de datos insuficientes.*
>
> **— Samuel Butler**
> Escritor, compositor y filólogo inglés.

6.1 Conclusiones

Esta tesis ha tenido como objetivo desarrollar una solución para ejecutar aplicaciones con requisitos temporales en dispositivos móviles que usen el sistema operativo Android/Linux. Las principales aportaciones del presente trabajo son las que se exponen a continuación:

- Se han estudiado los mecanismos que existen en los sistemas operativos Android/Linux para aislar núcleos en los procesadores multinúcleo actuales. De esta forma, aplicando una serie de mecanismos disponibles a nivel de kernel se consiguen reducir las interferencias del sistema operativo significativamente, alcanzando así en los núcleos aislados una respuesta temporal más predictible para las aplicaciones.

- Para poder ejecutar aplicaciones de tiempo real en un sistema operativo como Android es necesario tener lenguajes que ofrezcan unos servicios y funciones específicos. De forma nativa hemos detectado que Android tiene carencias para ejecutar aplicaciones con restricciones temporales, principalmente provocadas por la librería *Bionic* que sustituye a la *glibc* estándar. Debido a este motivo hemos presentado una propuesta para utilizar la librería *glibc* tradicional en Android. Además, esto ha posibilitado que hayamos podido conseguir ejecutar aplicaciones escritas en el lenguaje Ada sobre este sistema operativo.

- Nuestra solución busca tener núcleos aislados en el procesador, para ejecutar en ellos aplicaciones de tiempo real con las mínimas interferencias posibles. Al mismo tiempo, en el sistema conviven aplicaciones sin criticidad temporal, que pueden requerir compartir datos con aquellas que tienen requisitos temporales. Por lo tanto, hemos desarrollado cinco protocolos no bloqueantes de sincronización para compartir datos entre este tipo de aplicaciones. De este modo, podemos garantizar que la

respuesta temporal de las aplicaciones de tiempo real no se ve afectada negativamente cuando se comparten datos.

A continuación se procede a describir con más detalle los resultados obtenidos.

6.1.1 Aislamiento de procesadores multinúcleo en Linux/Android

Se ha alcanzado una solución que permite la ejecución de aplicaciones de tiempo real laxo en dispositivos multinúcleo con sistemas operativos Android/Linux. Para ello hemos aprovechado los procesadores multinúcleo y los mecanismos disponibles en el kernel de Linux que permiten aislar los núcleos del procesador. De este modo es posible ejecutar en los núcleos aislados las aplicaciones con requisitos temporales sin apenas interferencias del sistema. Esta solución es portable, así que puede ser utilizada en cualquier dispositivo Android actual sin la necesidad de modificar el kernel a nivel de código.

Aplicando los mecanismos estudiados y los parámetros de configuración descritos en nuestra solución, se ha demostrado a través de una batería de experimentos que los tiempos de peor caso en los núcleos aislados disminuyen significativamente. Se ha observado en nuestro entorno de pruebas que en núcleos no aislados podemos encontrarnos con interferencias del sistema que pueden durar hasta 1.5 segundos, mientras que si aplicamos los mecanismos y parámetros de configuración adecuados las interferencias se quedan por debajo de los 250 μs. Esto nos permite confirmar que hay una mejora sustancial en los tiempos de respuesta posibilitando la ejecución de aplicaciones de tiempo real laxo en Android.

Esta parte del trabajo de la tesis ha permitido llevar a cabo una presentación en una conferencia internacional y desarrollar dos trabajos de fin de grado derivados de la solución aquí expuesta:

- *CPU Isolation on the Android OS for running Real-Time Applications.* Alejandro Pérez Ruiz, Mario Aldea Rivás y Michael González Harbour. En 13th International Workshop on Java Technologies for Real-Time and Embedded Systems - JTRES, Paris, Francia, Octubre 2015.

- *Comparación de estrategias para mejorar la latencia de planificación en sistemas Linux*. Trabajo académico de fin de grado de Ingeniería Informática. Autor: Iván Revuelta Fernández, directores: Mario Aldea Rivás y Alejandro Pérez Ruiz.

- *Evaluación del sistema operativo Android para aplicaciones de tiempo real*. Trabajo académico de fin de grado de Ingeniería Informática. Autor: Santiago Sañudo Martínez, directores: Mario Aldea Rivás y Alejandro Pérez Ruiz.

6.1.2 Lenguajes de programación con características de tiempo real en Android

Se detectaron las limitaciones más relevantes para desarrollar aplicaciones de tiempo real en Android. Este sistema operativo utiliza una librería llamada *Bionic* en sustitución de la clásica *glibc*. Hemos propuesto una solución portable y fácilmente aplicable para utilizar el lenguaje de programación C con la librería tadicional *glibc*. Verificando su correcto funcionamiento hemos podido concluir que su uso es posible y desaparecen todas las limitaciones para las aplicaciones de tiempo real derivadas del uso de la librería *Bionic*.

Inicialmente todas nuestras pruebas fueran realizadas con el lenguaje de programación C, ya que es uno de los más extendidos en el desarrollo de sistemas de tiempo real, pero contamos con alternativas como el lenguaje Ada. Ada tiene soporte nativo para muchas de las características requeridas en las aplicaciones con requisitos temporales, además de estar orientado hacia la fiabilidad y de poseer otras características útiles en sistemas embebidos. Por estos motivos, se estudió cómo se podría utilizar este lenguaje en Android. Concluimos que es posible su utilización aplicando una serie de pasos descritos en esta tesis, y además se verificó su correcto funcionamiento realizando una batería de tests.

El estudio realizado y las propuestas llevadas a cabo durante esta parte del trabajo han permitido plasmar los resultados en los siguientes foros de ámbito internacional:

- *Real-Time Ada Applications on Android.* Alejandro Pérez Ruiz, Mario Aldea Rivas y Michael González Harbour. Presentación en 23rd International Conference on Reliable Software Technologies, Ada-Europe 2018, Lisbon (Portugal), in Presentation Abstracts, pp 11-12, Lisboa, Junio 2018.

- *Aplicaciones Ada en Android con requisitos de tiempo real.* Alejandro Pérez Ruiz, Mario Aldea Rivas y Michael González Harbour en Revista Iberoamericana de Automática e Informática industrial (RIAI) 16(3), pp 264-272, Junio 2019.

6.1.3 Sincronización no bloqueante entre aplicaciones de tiempo real y no tiempo real en dispositivos multinúcleo

Como desde el momento de inicio de la presente tesis nuestra propuesta consistía en dar soporte para ejecutar aplicaciones de tiempo real que coexistiesen con el resto de aplicaciones del sistema, nos topamos con la necesidad de desarrollar protocolos de sincronización no bloqueantes que permitiesen compartir datos entre aplicaciones temporalmente heterogéneas.

Hemos desarrollado cinco protocolos que cubren los principales escenarios de productor/consumidor. En estos protocolos no bloqueantes se prioriza que la parte de tiempo real finalice en un número acotado de pasos, es decir, que el tiempo de respuesta compartiendo datos sea lo más predecible posible. En cambio, cuando la parte sin requisitos temporales detecta que los datos a los que está accediendo o manipulando se han modificado durante su operación, tendrá que repetir parte de dicha operación hasta que los conflictos desaparezcan. Los algoritmos desarrollados son los siguientes:

- **SDrtR (shared data real-time reader)**: Dato compartido con un escritor sin requisitos temporales y un lector de tiempo real.
- **SDrtW (shared data real-time writer)**: Dato compartido con un escritor de tiempo real y múltiples lectores sin requisitos temporales.
- **QrtW (queue real-time writer)**: Cola circular con un escritor de tiempo real y un lector de no tiempo real.
  - **QrtWo (queue real-time writer with overwrite)**: Cuando la cola esté llena los nuevos datos sobrescriben los datos más antiguos.
  - **QrtWc (queue real-time writer with clearing)**: Cuando la cola esté llena y un nuevo elemento necesite ser encolado todos los datos de la cola serán descartados.
- **QrtR (queue real-time reader)**: Cola circular con un escritor de no tiempo real y un lector de tiempo real.

Con el desarrollo de estos protocolos y los tests que se han llevado a cabo en Android para su evaluación, hemos podido confirmar que su uso nos permite tener tiempos de respuesta suficientemente acotados para las aplicaciones con requisitos temporales que necesiten plazos en el rango de los milisegundos. Además el trabajo llevado a cabo ha permitido realizar una publicación internacional:

- *Non-blocking synchronization between real-time and non-real-time applications*. Alejandro Pérez Ruiz, Mario Aldea Rivas y Michael González Harbour en IEEE Access, vol. 8, pp. 147618-147634, 2020, doi: 10.1109/ACCESS.2020.3015385.

6.2 Trabajos futuros

Los resultados obtenidos en esta memoria han dejado algunas líneas de trabajo abiertas. A continuación citamos brevemente algunas de ellas:

- Para todas las propuestas expuestas en este trabajo sería deseable realizar una evaluación exhaustiva de cómo se comportaría un dispositivo Android en un entorno de control industrial real. De este modo, sería posible evaluar cuál es la complejidad de crear o adaptar software para este tipo de sistemas. Al mismo tiempo, también se podrían evaluar los beneficios de utilizar dispositivos de bajo coste con el sistema operativo Android en entornos industriales.

- Desarrollar más protocolos de sincronización no bloqueante apoyados en más tipos de estructuras de datos, que permitan enriquecer los posibles patrones de comunicación y sincronización entre aplicaciones de tiempo real y no tiempo real que coexistan en el sistema.

- Aplicando todos los mecanismos de aislamiento para procesadores multinúcleo evaluados en esta memoria, nos gustaría estudiar las posibles ventajas de utilizar el sistema operativo *MaRTE OS* [19] sobre Android. En *MaRTE OS* existe una versión arquitectural llamada *Linux Lib* que permite utilizar este sistema operativo de tiempo real como si se tratase de un hilo que se ejecuta sobre Linux, por lo tanto, creemos que sería posible hacer uso de él sobre el sistema operativo Android.

Bibliografía

[1] Nalage Vipul Vijay Kumar y AC Pise. "Android based portable health support system". En: *Paripex-Indian Journal Of Research* 8.4 (2019) (vid. pág. 2).

[2] Rajarshi Bhattacharya, Niladri Bandyopadhyay y S Kalaivani. "Real time Android app based respiration rate monitor". En: *International conference of Electronics, Communication and Aerospace Technology (ICECA)*. Vol. 1. IEEE. 2017, págs. 709-712 (vid. pág. 2).

[3] Natalya Dorosh, H Kuchmiy, O Boyko y col. "Development the software applications for mobile medical systems based on OS Android". En: *13th International Conference on Modern Problems of Radio Engineering, Telecommunications and Computer Science (TCSET)*. IEEE. 2016, págs. 808-810 (vid. pág. 2).

[4] *Android and RTOS together: The dynamic duo for today's medical devices*. `https://www.embedded-computing.com/embedded-computing-design/android-and-rtos-together-the-dynamic-duo-for-todays-medical-devices`. [Accedido el 21-10-2019]. 2010 (vid. pág. 2).

[5] M Ashokkumar y T Thirumurugan. "Integrated IOT based design and Android operated Multi-purpose Field Surveillance Robot for Military Use". En: *International Conference for Phoenixes on Emerging Current Trends in Engineering and Management (PECTEAM 2018)*. Atlantis Press. 2018 (vid. pág. 2).

[6] E Amareswar, G Shiva Sai Kumar Goud, KR Maheshwari y col. "Multi purpose military service robot". En: *International conference of Electronics, Communication and Aerospace Technology (ICECA)*. Vol. 2. IEEE. 2017, págs. 684-686 (vid. pág. 2).

[7] *NASA - PhoneSat Flight Demonstrations*. `https://www.nasa.gov/directorates/spacetech/small_spacecraft/phonesat.html`. [Accedido el 22-11-2019] (vid. pág. 2).

[8] John A Stankovic y Krithi Ramamritham. *Hard real-time systems*. IEEE Computer Society Press, 1988 (vid. pág. 2).

[9] Hermann Kopetz. *Real-time systems: design principles for distributed embedded applications*. Springer Science & Business Media, 2011 (vid. pág. 3).

[10] Alan Burns y Andrew J Wellings. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Pearson Education, 2001 (vid. pág. 3).

[11] Andrew S Tanenbaum y Herbert Bos. *Modern operating systems*. Pearson, 2015 (vid. págs. 3, 4).

[12] William Stallings. *Operating systems: internals and design principles*. Upper Saddle River, NJ: Pearson/Prentice Hall, 2009 (vid. pág. 3).

[13] Yanbing Li, Miodrag Potkonjak y Wayne Wolf. "Real-time operating systems for embedded computing". En: *Proceedings International Conference on Computer Design VLSI in Computers and Processors*. IEEE. 1997, págs. 388-392 (vid. pág. 4).

[14] John A Stankovic y Raj Rajkumar. "Real-time operating systems". En: *Real-Time Systems* 28.2-3 (2004), págs. 237-253 (vid. pág. 4).

[15] David Kalinsky. "A survey of task schedulers". En: *Proc. Embedded Systems Conf*. 1999 (vid. pág. 4).

[16] Prasanna Hambarde, Rachit Varma y Shivani Jha. "The survey of real time operating system: RTOS". En: *International Conference on Electronic Systems, Signal Processing and Computing Technologies*. IEEE. 2014, págs. 34-39 (vid. pág. 4).

[17] Katherine K Sheridan-Barbian. *A survey of real-time operating systems and virtualization solutions for space systems*. Inf. téc. Naval Postgraduate School Monterey CA, 2015 (vid. pág. 4).

[18] *POSIX.1 (2001). IEEE Std 1003.1:2001. Standard for Information Technology-Portable Operating System Interface (POSIX)*. 2001 (vid. pág. 4).

[19] Mario Aldea Rivas y Michael González Harbour. "MaRTE OS: An Ada kernel for real-time embedded applications". En: *International Conference on Reliable Software Technologies*. Springer. 2001, págs. 305-316 (vid. págs. 5, 125).

[20] *VxWorks*. `https://www.windriver.com/products/vxworks/`. [Accedido el 12-12-2019] (vid. pág. 5).

[21] *PikeOS Hypervisor - SYSGO*. `https://www.sysgo.com/products/pikeos-hypervisor`. [Accedido el 12-12-2019] (vid. pág. 5).

[22] Ken Sakamura. "ITRON: An overview". En: *TRON Project 1987 Open-Architecture Computer Systems*. Springer, 1987, págs. 29-34 (vid. pág. 5).

[23] *RTEMS Real Time Operating System (RTOS)*. `https://www.rtems.org/`. [Accedido el 28-11-2019] (vid. pág. 5).

[24] *eCos, Embedded Configurable Operating System*. `http://ecos.sourceware.org/`. [Accedido el 28-11-2019] (vid. pág. 5).

[25] *AUTOSAR - standards*. https://www.autosar.org/standards/. [Accedido el 12-12-2019] (vid. pág. 5).

[26] *RTA-OS - RTA Software Products - ETAS*. https://www.etas.com/en/products/rta_os.php. [Accedido el 20-10-2020] (vid. pág. 5).

[27] *Volcano VSTAR AUTOSAR for automotive ECU design*. https://www.mentor.com/embedded-software/autosar/software. [Accedido el 13-12-2019] (vid. pág. 5).

[28] M Masmano, Y Valiente, P Balbastre y col. "LithOS: a ARINC-653 guest operating for XtratuM". En: *Proc. of the 12th Real-Time Linux Workshop, Nairobi (Kenya)*. 2010 (vid. pág. 6).

[29] Miguel Masmano, Ismael Ripoll, Alfons Crespo y J Metge. "Xtratum: a hypervisor for safety critical embedded systems". En: *11th Real-Time Linux Workshop*. Citeseer. 2009, págs. 263-272 (vid. págs. 6, 29, 32).

[30] *Mobile Operating System Market Share Worldwide*. https://gs.statcounter.com/os-market-share/mobile/worldwide. [Accedido el 20-10-2020] (vid. pág. 6).

[31] Andrei Frumusanu. "A closer look at Android runtime (art) in Android L". En: *AnandTech. Retrieved July* 5 (2014) (vid. pág. 8).

[32] Stephen Samuel y Stefan Bocutiu. *Programming kotlin*. Packt Publishing Ltd, 2017 (vid. pág. 8).

[33] *NDK de Android*. https://developer.android.com/ndk. [Accedido el 19-10-2019]. 2019 (vid. págs. 9, 64).

[34] *Desarrolladores de Android - Panel de distribución*. https://developer.android.com/about/dashboards. [Accedido el 15-01-2020] (vid. pág. 10).

[35] Iliyan Malchev. *Here comes Treble: A modular base for Android*. https://android-developers.googleblog.com/2017/05/here-comes-treble-modular-base-for.html. [Accedido el 17-01-2020] (vid. pág. 11).

[36] Cláudio Maia, Luis Miguel Nogueira y Luis Miguel Pinho. "Evaluating Android OS for Embedded Real-Time Systems". En: *6th international workshop on operating systems platforms for embedded real-time applications*. 2010, págs. 63-70 (vid. págs. 15, 17, 25).

[37] Bhupinder S Mongia y Vijay K Madisetti. "Reliable real-time applications on Android os". En: *IEEE Electrical and Computer Engineering Electrical and Computer Engineering* (2010) (vid. pág. 15).

[38] Hyeong-Seok Oh, Beom-Jun Kim, Hyung-Kyu Choi y Soo-Mook Moon. "Evaluation of Android Dalvik virtual machine". En: *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*. ACM. 2012, págs. 115-124 (vid. pág. 15).

[39] Wolfgang Mauerer, Gernot Hillier, Jan Sawallisch, Stefan Hönick y Simon Oberthür. “Real-time Android: deterministic ease of use”. En: *Proceedings of Embedded Linux Conference Europe, ELCE*. Vol. 12. 2012 (vid. págs. 15, 25, 32).

[40] Luc Perneel, Hasan Fayyad-Kazan y Martin Timmerman. “Can Android be used for real-time purposes?” En: *International Conference on Computer Systems and Industrial Informatics*. IEEE. 2012, págs. 1-6 (vid. pág. 15).

[41] Edwin Peguero, Miguel Labrador y Brittany Cook. “Assessing jitter in sensor time series from Android mobile devices”. En: *IEEE International Conference on Smart Computing (SMARTCOMP)*. IEEE. 2016, págs. 1-8 (vid. pág. 15).

[42] Linux Foundation. *The Real-Time Linux*. `https://wiki.linuxfoundation.org/realtime/start`. [Accedido el 04-05-2019]. 2019 (vid. págs. 18, 26, 27).

[43] Federico Reghenzani, Giuseppe Massari y William Fornaciari. “The Real-Time Linux Kernel: A Survey on PREEMPT_RT”. En: *ACM Computing Surveys (CSUR)* 52.1 (2019), pág. 18 (vid. págs. 19, 20).

[44] Siro Arthur, Carsten Emde y Nicholas Mc Guire. “Assessment of the realtime preemption patches (RT-Preempt) and their impact on the general purpose performance of the system”. En: *Proceedings of the 9th Real-Time Linux Workshop*. 2007 (vid. págs. 20, 32).

[45] Konstantinos Chalas. *Evaluation of Real-time operating systems for FGC controls*. Inf. téc. 2015 (vid. pág. 20).

[46] Victor Yodaiken y col. “The rtlinux manifesto”. En: *Proc. of the 5th Linux Expo*. 1999 (vid. pág. 23).

[47] FSM Labs. *High Perfromance and Deterministic System Software-FSM Labs*. `https://www.fsmlabs.com/`. [Accedido el 24-05-2019]. 2019 (vid. pág. 23).

[48] *RTAI - The Real-Time Application Interface for Linux*. `https://www.rtai.org`. [Accedido el 24-05-2019]. 2019 (vid. pág. 23).

[49] *Xenomai - Real-time systems*. `https://www.xenomai.org`. [Accedido el 20-04-2019]. 2019 (vid. pág. 24).

[50] Jeremy H Brown y Brad Martin. “How fast is fast enough? Choosing between Xenomai and Linux for real-time applications”. En: *proc. of the 12th Real-Time Linux Workshop (RTLWS’12)*. 2010, págs. 1-17 (vid. pág. 24).

[51] Igor Kalkov. “A Real-time Capable, Open-Source-based Platform for Rapid Control Prototyping”. En: (2018) (vid. págs. 26, 32).

[52] Yin Yan y col. “RTDroid: a Real-Time Solution on Android”. Tesis doct. State University of New York at Buffalo, 2018 (vid. págs. 27, 32).

[53] *RTEMS - Real Time Operating System (RTOS)*. https://www.rtems.org/. [Accedido el 24-05-2019]. 2019 (vid. págs. 27, 28).

[54] Yasuyo Murakami. *Android on a Highly Reliable Real-Time OS*. https://pdfs.semanticscholar.org/6f62/60177a1146cc760e11e9b390f0b56f62347e.pdf (vid. págs. 29, 32).

[55] José Félix, Antonio Gutiérrez, Apolinar González y Walter Mata. "Plataforma domótica basada en la integración de un hipervisor con Android-x86". En: () (vid. pág. 30).

[56] *Android-x86 - Run Android on Your PC*. https://www.android-x86.org/. [Accedido el 16-06-2019]. 2019 (vid. pág. 30).

[57] Mark D Hill y Michael R Marty. "Amdahl's law in the multicore era". En: *Computer* 41.7 (2008), págs. 33-38 (vid. pág. 35).

[58] Technical Report CAST 32-A. *Multi-core Processors - Position Paper*. Inf. téc. 2016 (vid. pág. 35).

[59] Software Engineering Institute. *Attacks Multicore Challenges to Mission-Critical DoD Systems*. https://www.sei.cmu.edu/news-events/news/article.cfm?assetid=507890. [Accedido el 05-11-2020]. 2012 (vid. pág. 35).

[60] Shubham Kamdar y Neha Kamdar. "big.LITTLE Architecture: Heterogeneous Multicore Processing". En: *International Journal of Computer Applications* 119.1 (2015) (vid. pág. 36).

[61] Renesas. *R-Car H3*. https://www.renesas.com/us/en/solutions/automotive/soc/r-car-h3.html. [Accedido el 08-07-2019]. 2019 (vid. pág. 36).

[62] Gabriel Fernandez, Francisco J Cazorla, Jaume Abella y Sylvain Girbal. "Assessing Time Predictability Features of ARM Big. LITTLE Multicores". En: *30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE. 2018, págs. 258-261 (vid. pág. 36).

[63] Jan Altenberg. "Using the Realtime Preemption Patch on ARM CPUs". En: *Proceedings of the 11th Real-Time Linux Workshop*. 2009, págs. 229-236 (vid. pág. 36).

[64] Robert I Davis y Alan Burns. "A survey of hard real-time scheduling for multiprocessor systems". En: *ACM computing surveys (CSUR)* 43.4 (2011), pág. 35 (vid. pág. 37).

[65] Ralf Hinze y col. "Constructing red-black trees". En: *Proceedings of the Workshop on Algorithmic Aspects of Advanced Programming Languages, WAAAPL*. Vol. 99. 1999, págs. 89-99 (vid. pág. 39).

[66] Chris Okasaki. "Red-black trees in a functional setting". En: *Journal of functional programming* 9.4 (1999), págs. 471-477 (vid. pág. 39).

[67] Luca Abeni y Giorgio Buttazzo. “Integrating multimedia applications in hard real-time systems”. En: *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No. 98CB36279)*. IEEE. 1998, págs. 4-13 (vid. pág. 40).

[68] Anders Goransson. *Efficient Android Threading: Asynchronous Processing Techniques for Android Applications*. O’Reilly, 2014 (vid. pág. 43).

[69] *Linux Programmer’s Manual*. `http://man7.org/linux/man-pages/man7/cpuset.7.html`. [Accedido el 24-07-2019]. 2019 (vid. pág. 44).

[70] Pamela Thays Bezerra, Leandro AB Araujo, Giovane Boaviagem Ribeiro y col. “Dynamic frequency scaling on Android platforms for energy consumption reduction”. En: *Proceedings of the 8th ACM workshop on Performance monitoring and measurement of heterogeneous wireless and wired networks*. ACM. 2013, págs. 189-196 (vid. pág. 50).

[71] Thomas Gleixner, Paul E McKenney y Vincent Guittot. “Cleaning up Linux’s CPU hotplug for real time and energy management”. En: *ACM SIGBED Review* 9.4 (2012), págs. 49-52 (vid. pág. 53).

[72] K Sai Hari Chandana y Sagar Maliye. “Android activity based intelligent hotplug control”. En: *International Conference on Microelectronics, Computing and Communications (MicroCom)*. IEEE. 2016, págs. 1-4 (vid. pág. 53).

[73] *Real-time Linux frequently asked questions*. `https://rt.wiki.kernel.org/index.php/Frequently_Asked_Questions`. [Accedido el 05-08-2019]. 2019 (vid. pág. 54).

[74] Paul E McKenney y John D Slingwine. “Read-copy update: Using execution history to solve concurrency problems”. En: *Parallel and Distributed Computing and Systems*. 1998, págs. 509-518 (vid. pág. 54).

[75] Dinakar Guniguntala, Paul E McKenney, Josh Triplett y Jonathan Walpole. “The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux”. En: *IBM Systems Journal* 47.2 (2008), págs. 221-236 (vid. pág. 55).

[76] Teresa Higuera, Valérie Issarny, Michel Banâtre y Frédéric Parain. “Memory management for real-time Java: an efficient solution using hardware support”. En: *Real-Time Systems* 26.1 (2004), págs. 63-87 (vid. pág. 64).

[77] Bradford N., Buttlar D. y Farrell J. *Pthreads programming: A POSIX standard for better multiprocessing*. OŔeilly, 1996 (vid. pág. 65).

[78] Mark Klein, Thomas Ralya, Bill Pollak, Ray Obenza y Michael González Harbour. *A practitioner’s handbook for real-time analysis: guide to rate monotonic analysis for real-time systems*. Springer Science & Business Media, 2012 (vid. pág. 65).

[79] *Android Bionic status*. `https://android.googlesource.com/platform/bionic/+/HEAD/docs/status.md`. [Accedido el 20-10-2019]. 2019 (vid. pág. 66).

[80] *Open POSIX Test Suite*. `http://posixtest.sourceforge.net/`. [Accedido el 20-10-2019]. 2019 (vid. pág. 68).

[81] Jose F. Ruiz. "Programming Android in Ada". En: *Free and Open Source Software Developers' European Meeting (FOSDEM)*. 2013 (vid. pág. 71).

[82] *Ada Conformity Assessment Test Suite (ACATS)*. `http://www.ada-auth.org/acats.html`. [Accedido el 21-10-2019]. 2019 (vid. pág. 73).

[83] Dan Eilers y Tero Koskinen. "Adapting ACATS to the Ahven testing framework". En: *International Conference on Reliable Software Technologies*. Springer. 2011, págs. 75-88 (vid. pág. 73).

[84] Xiurong ZHANG. "The analysis and comparison of inter-process communication performance between computer nodes". En: *Management Science and Engineering* 5.3 (2011), págs. 162-164 (vid. pág. 75).

[85] Kwame Wright, Kartik Gopalan y Hui Kang. "Performance analysis of various mechanisms for inter-process communication". En: *Operating Systems and Networks Lab, Dept. of Computer Science, Binghamton University* (2007) (vid. pág. 75).

[86] Aditya Venkataraman y Kishore Kumar Jagadeesha. "Evaluation of inter-process communication mechanisms". En: *Architecture* 86 (2015), pág. 64 (vid. pág. 76).

[87] Marvin Damschen. *Concurrent shared memory access for Android applications and real-time processes*. 2012 (vid. pág. 78).

[88] Ragunathan Rajkumar. *Synchronization in real-time systems: a priority inheritance approach*. Vol. 151. Springer Science & Business Media, 2012 (vid. pág. 81).

[89] Björn B Brandenburg y Andrea Bastoni. "The case for migratory priority inheritance in Linux: Bounded priority inversions on multiprocessors". En: *In RTLWS'12*. Citeseer. 2012 (vid. pág. 81).

[90] Ragunathan Rajkumar, Lui Sha y John P Lehoczky. "Real-Time Synchronization Protocols for Multiprocessors." En: *RTSS*. Vol. 88. 1988, págs. 259-269 (vid. pág. 81).

[91] Ragunathan Rajkumar. "Real-time synchronization protocols for shared memory multiprocessors". En: *Proceedings., 10th International Conference on Distributed Computing Systems*. IEEE Computer Society. 1990, págs. 116-117 (vid. pág. 81).

[92] Alan Burns y Andy J Wellings. *Real-TIme Systems and Programming Languages. 4rd*. Addison-Wesley, 2009 (vid. pág. 81).

[93] Alan Burns y Andy J Wellings. "A schedulability compatible multiprocessor resource sharing protocol–MrsP". En: *25th euromicro conference on real-time systems*. IEEE. 2013, págs. 282-291 (vid. pág. 81).

[94] Philippas Tsigas y Yi Zhang. "Evaluating the performance of non-blocking synchronization on shared-memory multiprocessors". En: *Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. 2001, págs. 320-321 (vid. pág. 82).

[95] Philippas Tsigas y Yi Zhang. "Integrating non-blocking synchronisation in parallel applications: performance advantages and methodologies". En: *Proceedings of the 3rd international workshop on Software and performance*. 2002, págs. 55-67 (vid. pág. 82).

[96] Maurice Herlihy. "Wait-free synchronization". En: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.1 (1991), págs. 124-149 (vid. pág. 84).

[97] Keren Censor-Hillel, Erez Petrank y Shahar Timnat. "Help!" En: *Proceedings of the ACM Symposium on Principles of Distributed Computing*. 2015, págs. 241-250 (vid. pág. 84).

[98] James R Larus y Ravi Rajwar. "Transactional memory". En: *Synthesis Lectures on Computer Architecture* 1.1 (2007), págs. 1-226 (vid. pág. 84).

[99] Nir Shavit y Dan Touitou. "Software transactional memory". En: *Distributed Computing* 10.2 (1997), págs. 99-116 (vid. pág. 84).

[100] Tim Harris y Keir Fraser. "Language support for lightweight transactions". En: *ACM Sigplan Notices* 49.4S (2014), págs. 64-78 (vid. pág. 84).

[101] Virendra Marathe, Michael Spear, Christopher Heriot y col. "Lowering the overhead of nonblocking software transactional memory". En: (2006) (vid. pág. 84).

[102] Dave Dice, Yossi Lev, Mark Moir y Daniel Nussbaum. "Early experience with a commercial hardware transactional memory implementation". En: *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*. 2009, págs. 157-168 (vid. pág. 85).

[103] Kevin E Moore, Jayaram Bobba, Michelle J Moravan, Mark D Hill y David A Wood. "LogTM: Log-based transactional memory". En: *The Twelfth International Symposium on High-Performance Computer Architecture, 2006*. IEEE. 2006, págs. 254-265 (vid. pág. 85).

[104] C Scott Ananian, Krste Asanovic, Bradley C Kuszmaul, Charles E Leiserson y Sean Lie. “Unbounded transactional memory”. En: *11th International Symposium on High-Performance Computer Architecture*. IEEE. 2005, págs. 316-327 (vid. pág. 85).

[105] Peter Damron, Alexandra Fedorova, Yossi Lev y col. “Hybrid transactional memory”. En: *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. 2006, págs. 336-346 (vid. pág. 85).

[106] Bratin Saha, Ali-Reza Adl-Tabatabai y Quinn Jacobson. “Architectural support for software transactional memory”. En: *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE. 2006, págs. 185-196 (vid. pág. 85).

[107] Vladislav Nazaruk y Pavel Rusakov. “Blocking and non-blocking process synchronization: Analysis of implementation”. En: *Scientific Journal of Riga Technical University. Computer Sciences* 44.1 (2011), págs. 145-150 (vid. pág. 85).

[108] Greg Barnes. “A method for implementing lock-free shared-data structures”. En: *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*. 1993, págs. 261-270 (vid. pág. 85).

[109] Maged M Michael y Michael L Scott. “Simple, fast, and practical non-blocking and blocking concurrent queue algorithms”. En: *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. 1996, págs. 267-275 (vid. pág. 85).

[110] Keir Fraser. *Practical lock-freedom*. Inf. téc. University of Cambridge, Computer Laboratory, 2004 (vid. pág. 85).

[111] Håkan Sundell y Philippas Tsigas. “Scalable and lock-free concurrent dictionaries”. En: *Proceedings of the ACM symposium on Applied computing*. 2004, págs. 1438-1445 (vid. pág. 85).

[112] Steven Feldman, Pierre LaBorde y Damian Dechev. “Tervel: A unification of descriptor-based techniques for non-blocking programming”. En: *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. IEEE. 2015, págs. 131-140 (vid. pág. 86).

[113] James H Anderson y Srikanth Ramamurthy. “A framework for implementing objects and scheduling tasks in lock-free real-time systems”. En: *17th IEEE Real-Time Systems Symposium*. IEEE. 1996, págs. 94-105 (vid. pág. 86).

[114] Hermann Kopetz y Johannes Reisinger. “The non-blocking write protocol NBW: A solution to a real-time synchronization problem”. En: *Proceedings Real-Time Systems Symposium*. IEEE. 1993, págs. 131-137 (vid. págs. 86, 97-99).

[115] Philippas Tsigas y Yi Zhang. “Non-blocking data sharing in multiprocessor real-time systems”. En: *Proceedings Sixth International Conference on Real-Time Computing Systems and Applications. RTCSA'99 (Cat. No. PR00306)*. IEEE. 1999, págs. 247-254 (vid. págs. 86, 98).

[116] António Manuel de Sousa Barros. “Real-Time Software Transactional Memory”. Tesis doct. 2018 (vid. pág. 86).