Universidad de Cantabria
Facultad de Ciencias
Departamento de Electrónica y Computadores

Planificación de Tareas en Sistemas Operativos de Tiempo Real Estricto para Aplicaciones Empotradas

Tesis Doctoral

Mario Aldea Rivas

Santander, noviembre de 2002

Universidad de Cantabria
Facultad de Ciencias
Departamento de Electrónica y Computadores

Planificación de Tareas en Sistemas Operativos de Tiempo Real Estricto para Aplicaciones Empotradas

Memoria
presentada para optar al grado de
DOCTOR EN CIENCIAS (FÍSICAS)
por
Mario Aldea Rivas
Licenciado en Ciencias, Sección Físicas,
Especialidad Electrónica

Universidad de Cantabria
Facultad de Ciencias
Departamento de Electrónica y Computadores

Planificación de Tareas en Sistemas Operativos de Tiempo Real Estricto para Aplicaciones Empotradas

Memoria
presentada para optar al grado de Doctor en Ciencias (Físicas) por el Licenciado en Ciencias
Mario Aldea Rivas

El Director,

Dr. Michael González Harbour
Catedrático de Universidad

Declaro:
Que el presente trabajo ha sido realizado en el Departamento de Electrónica y Computadores de la Universidad de Cantabria, bajo mi dirección y reune las condiciones exigidas a los trabajos de Doctorado.

Santander, Noviembre de 2002

Fdo. Mario Aldea Rivas

Fdo. Michael González Harbour

En primer lugar me gustaría expresar mi reconocimiento hacia toda la comunidad de código libre en Internet, por su desinteresado e inestimable apoyo al progreso tecnológico y social. Gracias a su labor es posible obtener información y herramientas que permiten resolver la mayor parte de los problemas preliminares de una investigación, así como disponer de una base desde la que alcanzar los objetivos perseguidos. De lo contrario, sería muy difícil que proyectos como el desarrollado en esta memoria pudieran ser llevados a cabo en plazos de tiempo razonables. Esta tesis pretende constituir una modesta aportación a esa tarea con la libre distribución de sus resultados en la forma del sistema operativo MaRTE OS.

Quisiera agradecer a Michael su excelente labor como director de esta tesis. Creo que acertó desde el comienzo al proponerme un tema que me ha resultado sumamente interesante, y que ha acertado posteriormente en todas las decisiones tomadas para lograr que este trabajo llegara a buen término. Asimismo, querría agradecerle todos los conocimientos que, gracias a él, he adquirido durante estos años de trabajo conjunto, que espero continúen.

También me gustaría mostrar mi agradecimiento a todos los miembros del grupo de Computadores y Tiempo Real, por su ayuda y por haber logrado crear un ambiente realmente agradable y cómodo en el que trabajar.

Quiero dedicar este trabajo a mi familia, en especial a mis padres que siempre me han apoyado. A ellos debo en gran medida el haber realizado esta tesis, primero, porque fueron los que me animaron a realizar mis estudios en la carrera de Ciencias Físicas y posteriormente, porque me respaldaron durante mis primeros años de investigación en la Universidad, cuando parecía que nunca iban a llegar las becas o los contratos que me permitiesen seguir adelante con mi doctorado. Me gustaría que sintieran este trabajo como algo suyo.

Por último, también querría dedicar de forma especial este trabajo a Marisa, por su apoyo y ánimo constantes. También por su pacientes ratos de escucha mientras la hablaba de cosas como tareas, planificación, kernels o mutexes, sobre las que, a este paso, va a acabar siendo una “experta”. Por todo esto y mucho más, también me gustaría que sintiera que parte de este trabajo la pertenece.

Tabla de contenidos

Resumen

***Palabras clave**: tiempo real, sistema operativo, sistemas empotrados, Ada 95, POSIX, planificación a nivel de aplicación.*

Este trabajo se ha centrado en la mejora de los mecanismos proporcionados por los sistemas operativos POSIX para la planificación de tareas en sistemas empotrados de tiempo real. Con el objeto de disponer de una herramienta sobre la que realizar la citada investigación, se ha diseñado e implementado el sistema operativo de tiempo real para sistemas empotrados MaRTE OS. Se trata de un sistema operativo escrito utilizando el lenguaje Ada 95, que implementa la funcionalidad descrita en el perfil de sistema de tiempo real mínimo definido en el estándar POSIX.13 y que permite ejecutar aplicaciones escritas tanto en Ada como en C.

En MaRTE OS se han implementado algunos de los nuevos servicios propuestos para el perfil mínimo de sistema de tiempo real, como son el reloj monótono, la operación para suspensión absoluta de alta resolución, la política de planificación de servidor esporádico y los relojes y temporizadores de tiempo de ejecución. Esto nos ha permitido evaluar el impacto que su incorporación supone en el tamaño, complejidad y prestaciones de un núcleo conforme con el subconjunto mínimo, obteniéndose en todos los casos que el aumento de funcionalidad compensa con creces la complejidad añadida al sistema.

Por otra parte, en lo que constituye el principal resultado de esta tesis, se ha diseñado una interfaz que permite a las aplicaciones definir los algoritmos de planificación bajo los cuales desean planificar sus tareas. El modelo en el que se apoya está basado en la definición de un tipo especial de tareas (planificadoras) que pueden activar o suspender a otras tareas (planificadas), lo que las permite implementar algoritmos de planificación definidos por la aplicación.

La interfaz anteriormente mencionada se ha extendido de forma que también permita a las aplicaciones definir sus propios protocolos de sincronización. Para ello se define un tipo especial de mutexes cuyo protocolo no es uno de los definidos por el estándar POSIX sino que es implementado por una tarea planificadora.

La interfaz extendida, que incluye tanto la planificación como la sincronización definida por la aplicación, supera a otras soluciones existentes en la bibliografía al constituir una solución general, permitir la definición de protocolos de sincronización, ser válida para sistemas multiprocesadores, proteger al sistema operativo ante fallos en un algoritmo de planificación y perseguir su integración en el estándar POSIX, siendo compatible con las políticas de planificación y protocolos de sincronización definidos en ese estándar.

Se han desarrollado dos versiones equivalentes de la interfaz, una en lenguaje C y otra en lenguaje Ada, ambas integradas en el conjunto de interfaces POSIX. Se ha procedido a su implementación en el sistema operativo MaRTE OS, lo que nos ha permitido comprobar su validez y generalidad mediante la escritura de varias políticas de planificación y protocolos de sincronización para su prueba.

1. Introducción

1.1. Sistemas empotrados de tiempo real

Existen sistemas computacionales fuertemente relacionados con el entorno que les rodea y con el que se encuentran en constante interacción. Ejemplos de este tipo de sistemas son los sistemas de adquisición de datos y los controladores industriales, los cuales están compuestos por diversos elementos (actuadores, sensores, unidades de cómputo, redes de interconexión, etc.) que deben trabajar de forma conjunta y coordinada. Debido a la naturaleza cambiante del entorno con el que interactuan, junto con la necesidad de coordinación entre sus componentes, los resultados obtenidos sólo podrán ser considerados válidos cuando, además de ser correctos desde el punto de vista lógico, hayan sido generados a tiempo. Resultará, por tanto, preciso imponer restricciones temporales cuyo cumplimiento garantice el correcto funcionamiento del sistema. Este tipo de sistemas, capaces de realizar tareas y responder a eventos asíncronos externos dentro de unos plazos temporales determinados son los denominados “Sistemas de Tiempo Real” [STA88].

Para que sea posible la predecibilidad temporal del sistema completo, todas las partes que le componen deberán de presentar un comportamiento predecible. En consecuencia, en el caso de utilizar un sistema operativo, los servicios que éste proporcione a las aplicaciones deberán presentar tiempos de respuesta acotados, de forma que así sea capaz de garantizar los requerimientos temporales de los procesos bajo su control. Mientras que en un sistema operativo de tiempo compartido, como Unix [BAC86][KER84], lo importante es proporcionar a los usuarios unos buenos tiempos de respuesta promedios [PET91], la clave en los sistemas operativos de tiempo real será garantizar los requerimientos temporales; el tiempo de respuesta promedio pasa así a un segundo plano.

Dentro de los sistemas de tiempo real, un tipo particular son los “sistemas empotrados”. En este tipo de sistemas, el computador constituye una parte más de un sistema mayor en el que se encuentra altamente integrado y en el que se dedica a realizar una función (o un pequeño conjunto de ellas). Las aplicaciones tradicionales de este tipo de sistemas incluirían sistemas de control en aviones, trenes, nudos de telecomunicaciones, motores de automóviles, procesos industriales, teléfonos móviles, etc.

Sin embargo, la utilización de estos sistemas está extendiéndose paulatinamente a otras muchas aplicaciones a medida que los avances tecnológicos permiten disponer de procesadores baratos cada vez más potentes y de memorias de menor coste y mayor escala de integración. Así, ya es habitual encontrarse con computadores empotrados en productos como televisores, juguetes, reproductores de discos DVD, pequeños electrodomésticos, etc. Su utilización es tan extensa que “World Semiconductor Trade Statistics” [WSTS] en su “blue book” de 2002 estima que en el mundo hay 5.000 millones de sistemas empotrados en uso, con unas expectativas de ventas de 2.000 millones de unidades/año para los próximos años [EMBED]. Estas ventas, que ya superaron los 200.000 millones de dólares en el año 2000 [COO00], suponen según “World

Semiconductor Trade Statistics", el 94% del mercado mundial de semiconductores, frente al pequeño 6% que corresponde a los computadores personales.

Existe una gran variedad de sistemas informáticos utilizados en entornos empotrados, desde pequeños microcontroladores de 4 bits hasta potentes computadores con procesadores digitales de señal (DSPs) o procesadores RISC de 64 bits. Sin embargo, la mayor parte de los sistemas empotrados ven limitadas sus prestaciones por razones de tamaño, peso, consumo o coste. Por las citadas razones, lo normal es que no dispongan de un terminal de propósito general para interfaz con el usuario, ni de sistema de ficheros o dispositivos de almacenamiento magnéticos, y tengan un procesador y capacidad de memoria muy reducidos. Normalmente no tienen hardware de gestión de memoria o, si lo tuvieran, permanecería sin usar e inhabilitado.

Las limitaciones mencionadas eran mucho más importantes hace unos años debido al menor desarrollo tecnológico, lo que exigía reducir al máximo los tiempos de computo y el tamaño de las aplicaciones empotradas. En consecuencia, la mayor parte de ellas se escribían en lenguaje ensamblador y sin utilizar ningún sistema operativo. En la actualidad, las mejoras tecnológicas y el consiguiente incremento en la complejidad de las aplicaciones han hecho casi imprescindibles la utilización de lenguajes de alto nivel y sistemas operativos en la mayoría de los entornos empotrados. Así, según datos de un estudio realizado por "Venture Development Corporation" [VDC00], ya en el año 2000 el 48% de los fabricantes de sistemas empotrados utilizaban sistemas operativos comerciales, el 26% sistemas operativos propios y solamente quedaba otro 26% que no utilizaban ningún sistema operativo.

En este avance también ha influido la necesidad constante de los fabricantes de sistemas empotrados de reducir tiempos de desarrollo y aumentar la fiabilidad de las aplicaciones generadas. Esta mejoras se han logrado en gran medida mediante la aplicación en el desarrollo de las aplicaciones empotradas de los aspectos mas avanzados de la ingeniería software para sistemas de tiempo real, como son la programación concurrente, estrategias sofisticadas de planificación de tareas, programación orientada a objetos, etc. Tecnologías éstas, que difícilmente habrían podido ser aplicadas sin la utilización de lenguajes de alto nivel y sistemas operativos.

En un principio, fueron los fabricantes de los sistemas hardware los que gradualmente procedieron a incorporar algunos de estos conceptos en sus sistemas de desarrollo específicos. La consecuencia de esta acción descoordinada y casi siempre competitiva, era una gran Babel conceptual, que requería de las empresas un considerable esfuerzo para transferir o adaptar el software a las diferentes plataformas hardware de sus controladores industriales, que por su naturaleza son en general poco homogéneos. La solución a esta problemática se está buscando a nivel internacional mediante la elaboración de normas de estandarización, que permiten independizar los diferentes aspectos del diseño del sistema --arquitectura hardware, software de base, sistema operativo, aplicación--, consiguiendo con ello una mayor modularidad, reusabilidad y, en definitiva, la reducción del tiempo de diseño y de los costos.

Entre las normas internacionales de estandarización destaca el estándar POSIX (conocido en el ámbito internacional con la referencia ISO/IEC-9945) [PSX96] [PSX01], cuyo objetivo es permitir la portabilidad de aplicaciones a nivel de código fuente entre diferentes sistemas operativos. Con este fin, y basándose en la interfaz del extensamente utilizado sistema operativo Unix, el estándar POSIX ("Portable Operating System Interface") define la interfaz que los sistemas operativos deben ofrecer a las aplicaciones, así como la semántica de los servicios ofrecidos por esa interfaz.

La interfaz descrita en el estándar permite escribir aplicaciones de tiempo real de forma portable [GON93]. Sin embargo, debido al tamaño del estándar POSIX, resulta inabordable su implementación completa en un sistema operativo destinado a computadores empotrados pequeños. Por esta razón en el estándar POSIX.13 [PSX98] se han definido cuatro subconjuntos de servicios del sistema operativo, que se han orientado a cuatro tipos de plataformas comúnmente utilizadas en aplicaciones comerciales de tiempo real. El menor de dichos perfiles, el denominado "Sistema de Tiempo Real Mínimo" [GON97A], está pensado para aplicaciones empotradas pequeñas. Dicho perfil incluye un pequeño subconjunto de toda la funcionalidad definida en el estándar POSIX, lo que permite su implementación como un núcleo de sistema operativo pequeño y eficiente.

El estándar POSIX está gozando de gran aceptación entre los fabricantes. Así en los últimos años han ido apareciendo numerosos sistemas operativos de tiempo real conformes (en mayor o menor medida) con el estándar POSIX. La mayoría de ellos, entre los que se encuentran los más utilizados por la industria a nivel mundial, son sistemas propietarios tales como VxWorks [WINDR] [VXW97], pSOSystem [pSOS], QNX [QNX] [QNX92] o LynxOS [LYNXO] (entre ellos sumaban en 2000 más del 42% del mercado mundial según "Dedicated Systems Experts" [TIM00]). Pero también este estándar es seguido por gran cantidad de sistemas operativos distribuidos bajo políticas de libre distribución y código libre, entre ellos que cabe citar MiThOS [MUE95], RTEMS [RTE96], RT-Linux [RTLIN][BAR97] y S.Ha.R.K [GAI01].

Otro estándar de facto en sistemas operativos lo constituye Microsoft Windows en sus diversas versiones. Sin embargo, a pesar de su extensa utilización no podemos considerarlo un estándar real, desde el momento que la especificación de su interfaz depende de una sola compañía que puede cambiarla cuando así lo desee. Por otro lado, la mayor parte de las versiones de Windows constituyen sistemas operativos de propósito general, no resultando por tanto apropiadas para aplicaciones empotradas ni de tiempo real. Existe una versión de tiempo real, denominada Windows CE [WINCE], pensada para sistemas empotrados de tamaño medio (Windows CE tiene un tamaño mínimo o "footprint" de 400Kbytes) que también adolece de los problemas de falta de estandarización ya comentados.

Otro importante conjunto de estándares en el área de las aplicaciones de tiempo real está constituido por los lenguajes de programación. Así, el C++ [C++98], y el lenguaje Ada 95 [ADA95] [ADA00] [BAR96], aportan mecanismos para poder aplicar programación orientada a objetos. Ada 95 es un descendiente directo de Ada 83 [ADA83], el cual fue desarrollado por iniciativa del Departamento de Defensa de los EE.UU. para ser utilizado en sistemas empotrados. Ada 95 soporta los aspectos mas avanzados de la ingeniería software para sistemas de tiempo real, como son la programación concurrente con tiempos de respuesta predecibles, estrategias sofisticadas de planificación de tareas y programación orientada a objetos. Su tipificación estricta permite detectar muchos problemas en tiempo de compilación, lo que incrementa considerablemente la fiabilidad del código generado.

Un lenguaje de programación que ha experimentado un importante avance en el campo de los sistemas empotrados es el Java [GOS00] debido principalmente a la portabilidad a nivel binario que proporciona. Sin embargo, la definición original del lenguaje impedía su uso en sistemas de tiempo real estricto. Esta limitación está tratando de resolverse [BRO01] (aunque por desgracia de forma competitiva) por medio de las extensiones al lenguaje propuestas por el "J Consortium Real-Time Java Working Group" (RTJWG) [JCONS] y por el "RT Java Expert Group" (RTJEG) [BOL00].

1.2. Algoritmos de planificación

Como se expuso anteriormente, un sistema de tiempo real se caracteriza por la fuerte interacción que mantiene con el entorno que le rodea, debiendo responder a los diferentes eventos generados por éste en unos plazos de tiempo preestablecidos. La naturaleza compleja y concurrente del entorno, conduce a la utilización de arquitecturas de software concurrente para los sistemas de tiempo real. En este tipo de arquitecturas, el sistema global se divide en un conjunto de actividades concurrentes, que denominaremos tareas, cada una de ellas encargada de responder a un determinado evento o conjunto de eventos generados por el entorno. El software producido siguiendo este modelo será más próximo al sistema real, resultando por tanto mucho más sencillo y comprensible.

Puesto que el número máximo de tareas que es posible ejecutar simultáneamente en un computador es limitado (como máximo igual al número de procesadores), es necesario definir un conjunto de reglas que permitan determinar qué tarea o tareas deben ser ejecutadas en cada momento. Estas reglas constituyen los denominados algoritmos o políticas de planificación, y de su elección depende en gran medida el que se satisfagan o no las restricciones temporales impuestas al sistema.

El algoritmo de planificación históricamente más utilizado en sistemas empotrados de tiempo real es el “ejecutivo cíclico” [BAK88], el cual utiliza una tabla o “plan estático” para indicar los instantes en que cada tarea debe tomar y abandonar la CPU. Dicha tabla es recorrida cíclicamente durante la ejecución de la aplicación. Una de las ventajas de este algoritmo radica en su simplicidad (no existe un sistema operativo propiamente dicho), siendo fácilmente implementable con muy poco código y, por tanto, muy eficiente en tiempo de ejecución. Otra ventaja es la predecibilidad absoluta derivada de que los instantes de ejecución de las tareas son fijos y conocidos, lo que además permite asegurar la planificabilidad del sistema desde el momento de la construcción de la tabla. Por otro lado, sus principales inconvenientes son su rigidez (un pequeño cambio en una tarea puede obligar a cambiar toda la tabla), el gran tamaño que puede llegar a alcanzar la tabla y su ineficiencia a hora de gestionar eventos aperiódicos.

La complejidad de las aplicaciones empotradas de tiempo real ha ido aumentando con el tiempo, y en la actualidad es normal que éstas requieran servicios como la multitarea avanzada o las comunicaciones. Dichos servicios, típicamente facilitados por un sistema operativo complejo, no están disponibles cuando se utilizan los “ejecutivos cíclicos”. Esta deficiencia es solventada mediante la utilización de sistemas operativos de tiempo real [LOC92].

Los sistemas operativos de tiempo real utilizan algoritmos de planificación en tiempo de ejecución basados en prioridades, en los que los problemas de contención de recursos se resuelven en el instante que se producen, eliminándose la necesidad de que exista un plan de ejecución previo. En dichos algoritmos, a cada tarea le es asignada una prioridad y, en función de ella, se resuelven los posibles conflictos de utilización del procesador y de los demás recursos del sistema.

Dependiendo de si la prioridad de las tareas es constante o cambia en función del estado del sistema los algoritmos de planificación en tiempo de ejecución se dice que están basados en prioridades estáticas o dinámicas. Además, sea cual sea el tipo de prioridades, puede diferenciarse entre políticas expulsoras y no expulsoras: en una política de planificación no expulsora la tarea en posesión del procesador no verá detenida su ejecución hasta que así lo desee o bien hasta que se bloquee al tratar de acceder a algún recurso no disponible en ese momento. Por el contrario, en una política expulsora una tarea perderá la posesión del procesador en el momento en que aparezca otra tarea de mayor prioridad lista para ejecutar. El

planificador no expulsor es más sencillo de implementar, aunque a cambio, presenta la importante desventaja de que su uso puede provocar retrasos importantes en la ejecución de las tareas más prioritarias.

1.2.1. Prioridades fijas

La inmensa mayoría de sistemas operativos de tiempo real comerciales utilizan políticas de planificación expulsoras basadas en prioridades fijas. De hecho, de este tipo son las políticas soportadas tanto por el lenguaje de programación Ada como por el estándar POSIX. Esta elección es debida, por un lado, a que estos algoritmos permiten un alto aprovechamiento de la CPU, mientras que su implementación resulta relativamente sencilla, y por otro, a que para ellos se dispone de la amplia y consolidada base teórica aportada por las teorías RMA ("Rate Monotonic Analysis") y su posterior extensión como teoría DMA ("Deadline Monotonic Analysis") [LIU73] [LEU82] [AUD91A] [AUD91B] [KLE93]. Mediante esta teoría es posible determinar si un conjunto de tareas resulta planificable, o lo que es lo mismo, si se cumplirán los requisitos temporales en forma de plazos de finalización, marcados para ellas. También esta teoría permite demostrar que si se desea realizar una asignación óptima de prioridades, éstas deben de ser asignadas a las tareas en relación inversa a su plazo de ejecución, esto es, a menor plazo mayor prioridad[1].

De entre las políticas de planificación expulsoras basadas en prioridades fijas la más extensamente utilizada es la denominada "FIFO con prioridades". En dicha política se respeta el orden de llegada (FIFO, "first in first out") entre las tareas de igual prioridad, con lo que una tarea se mantendrá en ejecución hasta que se bloquee o hasta que sea expulsada por otra de mayor prioridad. La política "FIFO con prioridades" es una de las incluidas en el estándar POSIX y la única soportada por el lenguaje Ada 95. Esta política es una de las de más amplia utilización, encontrándose implementada en la inmensa mayoría de los sistemas operativos de tiempo real.

La política cíclica o de rodaja temporal (también conocida por su nombre en inglés "round-robin"), es otra de las que se encuentran soportadas en numerosos sistemas operativos, y al igual que la FIFO incluida en el POSIX. Esta política es igual que la anterior salvo por una limitación adicional: el tiempo durante el que una tarea puede mantenerse en ejecución de forma continuada está limitado a un intervalo máximo. Transcurrido dicho intervalo deberá ceder la CPU a otras tareas de su misma prioridad. Sólo en el caso de que no haya ninguna, la tarea original retomará la CPU para comenzar otro intervalo. La política "round-robin" puede ser utilizada para planificar tareas sin requisitos temporales o con requisitos temporales laxos, junto con tareas con requisitos temporales estrictos planificadas bajo política "FIFO" a niveles de prioridad mayores.

Las políticas anteriormente citadas resultan apropiadas para realizar la planificación de tareas periódicas, esto es, tareas cuyas activaciones se producen a intervalos fijos de tiempo. Sin embargo, muchos de los eventos generados por el entorno no se producen a intervalos regulares. Se han propuesto varias estrategias de gestión de eventos aperiódicos mediante tareas de prioridad fija [SHA86], algunas de las cuales pasaremos a describir a continuación. Posiblemente la más sencilla de ellas, sea el procesado inmediato del evento por parte de una tarea de alta prioridad. Aunque este método permite responder al evento de forma prácticamente instantánea, presenta el grave problema de que su efecto sobre las tareas de menor prioridad no está acotado ante llegadas masivas de eventos. Una solución alternativa consiste en utilizar una

1. Excepto en el caso de plazos mayores que los periodos de las respectivas tareas.

tarea de baja prioridad para realizar el procesado directo de los eventos. Esta solución no afecta a las demás tareas del sistema, pero a cambio, el tiempo promedio de respuesta puede ser muy largo dependiendo de la carga del sistema. Una solución mejor que las anteriores consiste en realizar el chequeo periódico de la llegada de eventos mediante una tarea de alta prioridad. Ajustando el periodo de esta tarea es posible acotar su efecto sobre el resto de tareas de menor prioridad del sistema. Sin embargo, esta estrategia aún presenta un tiempo promedio de respuesta a los eventos excesivamente alto.

Para resolver los problemas y limitaciones mencionadas para las soluciones anteriores, se han propuesto políticas de planificación de servidores aperiódicos basadas en la reserva de una porción del tiempo de la CPU para la gestión de eventos aperiódicos ("Servidor Esporádico" [SPR89], "Intercambio de Prioridades" [SPR88], "Servidor Diferido" [STR95]), o en el aprovechamiento de los intervalos en los que el procesador está desocupado, como ocurre en los denominados servidores de holgura ("Slack Stealing" [LEH92]). El uso de estas políticas permite acotar el efecto de los servidores aperiódicos sobre las demás tareas (incluso en situaciones de llegada masiva de eventos), a la vez que tratan de optimizar el tiempo de respuesta promedio a los eventos. De entre ellas cabe destacar la denominada "Servidor Esporádico" por el buen tiempo de respuesta promedio que proporciona y la baja sobrecarga que supone para el resto de tareas del sistema. Esta política también merece ser destacada por su amplia utilización al ser la elegida en el estándar POSIX para la gestión de eventos aperiódicos.

Las tareas planificadas bajo la política de "Servidor Esporádico" se comportan de forma similar a como lo hacen las de política "FIFO con prioridades" pero utilizando dos niveles de prioridad. Los cambios entre ambos niveles se producen en base a dos parámetros propios de cada tarea: su periodo de relleno y su capacidad máxima de ejecución. La política asegura que durante cualquier intervalo de tiempo igual a su periodo de relleno, una tarea ejecutará a prioridad alta como máximo un tiempo igual a su capacidad máxima de ejecución. Las reglas que rigen esta política son las siguientes:

- Si se produce la llegada de un evento cuando hay capacidad de ejecución disponible, la tarea pasa a estado ejecutable al nivel de prioridad alto.
- Una vez que la tarea comienza a ejecutar a nivel de prioridad alto, continuará en posesión de la CPU hasta que se bloquee, sea expulsada o agote su capacidad de ejecución.
- Cuando la tarea agota su capacidad de ejecución, continúa en estado ejecutable pero a nivel de prioridad bajo.
- Cada porción de ejecución consumida a prioridad alta es repuesta transcurrido un tiempo igual al periodo de relleno. Además, si cuando se produce una de estas recargas la tarea se encuentra en estado ejecutable a nivel de prioridad bajo, su prioridad es elevada al nivel alto.

Esta política también es útil para eliminar los efectos negativos que la activación diferida de una tarea ("jitter") tiene sobre la planificabilidad de tareas de menor prioridad [GUT95].

1.2.2. Prioridades dinámicas

Como ya mencionamos con anterioridad, en los sistemas empotrados las prioridades estáticas son mucho más utilizadas que las dinámicas. Las principales razones para ello son que la sobrecarga causada por el algoritmo suele ser menor para este tipo de políticas, que su base teórica fue desarrollada con anterioridad y que son estables ante sobrecargas del sistema. Esta última propiedad consiste en que es posible conocer y controlar a priori que tarea o tareas incumplirán sus plazos en el caso de que otra tarea consuma más tiempo del estimado para ella.

Por otro lado, los algoritmos de planificación basados en prioridades dinámicas presentan dos importantes ventajas: aprovechan al máximo la potencia del procesador, de forma que con ellos son planificables conjuntos de tareas que no lo serían de haber utilizado otros algoritmos (en particular los basados en prioridades estáticas), y se adaptan muy bien a entornos altamente dinámicos en los que la carga del sistema no puede ser conocida de antemano. Estas ventajas, junto con el aumento de la potencia de los procesadores y el desarrollo de la base teórica necesaria, permiten que en la actualidad pueda resultar práctica y factible la utilización de este tipo de algoritmos en sistemas empotrados.

El más importante de los algoritmos basados en prioridades dinámicas es el EDF ("Earliest Deadline First") [LIU73], en el que la prioridad de una tarea es tanto mayor cuanto más cercano se encuentre el instante en que finaliza su plazo de ejecución, o lo que es lo mismo, cuanto más cerca se encuentre de incumplir los requisitos temporales marcados para ella. El plazo de ejecución de una tarea no es un valor constante: cuando la tarea finaliza el trabajo correspondiente a una activación, su plazo pasa a ser el referido a la activación siguiente. Con esta política una tarea se mantendrá en ejecución hasta que se bloquee, finalice el trabajo correspondiente a la presente activación o sea expulsada como consecuencia de la activación de otra tarea con un plazo de finalización más cercano.

Un algoritmo similar al anterior es el denominado LLF ("Least Laxity First") [AUD90], en el que, además de los plazos de las tareas, también se toma en consideración el tiempo de ejecución que éstas tienen pendiente en un momento dado. Se denomina holgura al valor obtenido de restar el tiempo que falta hasta el cumplimiento del plazo menos el tiempo de ejecución pendiente. Una tarea será tanto más prioritaria cuanto menor sea su holgura.

También merece ser destacado el algoritmo denominado "Best-Effort" [LOC85], muy similar a los anteriores pero en el que además se tiene en cuenta la importancia de cada tarea, cuantificada como un valor numérico. Con este algoritmo, será elegida para ejecutar aquella tarea con la mejor relación entre la cercanía del cumplimiento de su plazo de ejecución y su importancia. Este algoritmo, al igual que los basados en prioridades estáticas, presenta la importante ventaja de comportarse de forma estable ante sobrecargas del sistema.

También para las políticas basadas en prioridades dinámicas se han propuesto algoritmos que permiten planificar de forma eficiente los eventos aperiódicos, ya sean basadas en la extracción de holgura ("Servidor de holgura" [RIP96]), o en la reserva de ancho de banda, como la política denominada CBS ("Constant Bandwidth Server") [ABE98]. Esta última asigna un porcentaje de tiempo de CPU para la gestión de eventos aperiódicos que no se verá superado incluso bajo llegadas masivas de eventos, lo que asegura que su efecto sobre las demás tareas del sistema sea siempre acotado.

En la política CBS, las tareas se caracterizan por tres parámetros: periodo, capacidad máxima de ejecución y capacidad de ejecución disponible. Esta política se apoya en un algoritmo de planificación EDF, definiendo las reglas que, basándose en el valor de los parámetros anteriormente citados, rigen cómo debe modificarse el plazo de ejecución asignado a las tareas. Estas reglas son:

- Una tarea se crea con plazo de ejecución igual al instante inicial, capacidad de ejecución al valor máximo y sin trabajos pendientes (en estado inactivo).
- Cuando llega un nuevo trabajo para una tarea inactiva ésta pasa a estado activo. En el caso de que su capacidad de ejecución disponible sea demasiado grande para que, respetando el ancho de banda asignado a la tarea, pueda ser agotada antes del plazo

actual, la capacidad de ejecución se recarga al máximo valor y el plazo actual se hace igual al plazo anterior más el periodo.

- Cuando llega un nuevo trabajo para una tarea activa (esto es, con trabajos pendientes) es encolado para ser completado cuando le corresponda.
- El tiempo durante el que la tarea se encuentra en ejecución es restado de su capacidad de ejecución disponible.
- Cuando la tarea agota su capacidad de ejecución, la capacidad de ejecución se recarga al máximo valor y el plazo actual se hace igual al plazo anterior más el periodo.

Además de la ya mencionada propiedad de presentar un efecto acotado sobre las demás tareas del sistema (incluso bajo llegadas masivas de eventos), la política CBS proporciona un mejor tiempo de respuesta promedio que otras políticas para planificación de eventos aperiódicos basadas en reserva de ancho de banda [ABE98].

1.2.3. Reparto proporcional

Existe otro tipo de algoritmos de planificación no basados en prioridades que se utilizan principalmente en el campo de las comunicaciones y de las aplicaciones multimedia (muestreo de voz, adquisición de imágenes, reproducción de vídeo, etc.). Estos algoritmos persiguen el reparto proporcional de los recursos del sistema entre las distintas tareas o procesos que lo componen.

En el tipo de aplicaciones mencionadas la carga de trabajo es variable o no se conoce de antemano. Con esas características, garantizar el cumplimiento estricto de los plazos temporales puede resultar imposible o requerir la adopción de unas suposiciones excesivamente pesimistas, las cuales provocarían bajos niveles promedio de utilización del sistema. Sin embargo, aunque no es posible imponer requisitos temporales estrictos, este tipo de aplicaciones sí que requieren el cumplimiento de requisitos temporales laxos, que al menos permitan acotar sus tiempos de respuesta promedio de forma que sea posible asegurar que se proporciona a los usuarios una determinada calidad del servicio (QoS).

Los algoritmos de planificación de reparto proporcional permiten asegurar que a cada tarea le será concedido el porcentaje del tiempo de procesador solicitado [MER94] [JEF98] [GOY96] [PAR93]. Los requisitos son expresados por las tareas como la necesidad de disponer de al menos X unidades de tiempo de cada Y. Existen versiones de este tipo de algoritmos que, no sólo garantizan que la porción de tiempo de CPU requerida será satisfecha en un intervalo de tiempo relativamente largo, sino que incluso permiten garantizar para cualquier instante de tiempo que Y unidades de tiempo más tarde la tarea habrá dispuesto de al menos X unidades de tiempo para ejecutar [BAR96A] [JON97].

Estos algoritmos permiten el cambio dinámico de los atributos de planificación de las tareas, de forma que es posible adaptar la calidad de servicio proporcionada en función de la carga total del sistema.

1.3. Protocolos de sincronización

Los algoritmos de planificación, algunos ejemplos de los cuales han sido presentados en el apartado anterior, constituyen un conjunto de reglas que determinan el modo en que se reparte el uso del procesador entre las tareas. Sin embargo, éste no es el único recurso que puede ser compartido en el sistema. Dependiendo de la aplicación, varias tareas también pueden utilizar

de forma concurrente otros dispositivos (típicamente de almacenamiento secundario o de entrada/salida), así como estructuras de datos en memoria utilizadas para compartir información entre ellas.

En la mayor parte de los casos la coherencia de la información contenida en un recurso sólo podrá ser preservada si se evita que éste sea utilizado de forma concurrente por dos o más tareas. El uso excluyente de un recurso significa que, durante el tiempo en que una tarea lo está utilizando (que denominaremos "sección crítica"), las demás que pretendan hacerlo serán bloqueadas a la espera de que aquel sea liberado. Este bloqueo deberá producirse siempre, independientemente de cuales sean las prioridades de las tareas bloqueante y bloqueadas.

La utilización de la exclusión mutua tal y como se ha explicado en el párrafo anterior, esto es, sin que las propiedades de planificación de una tarea en posesión de un recurso se vean afectadas en forma alguna, puede provocar el efecto denominado inversión de prioridad no acotada [SHA90]. Este efecto se produce cuando una tarea de alta prioridad bloqueada en un recurso debe esperar, no sólo a que la tarea de baja prioridad dueña del recurso finalice su sección crítica, sino además a que ejecuten otras tareas de prioridad intermedia que puedan expulsar de la CPU a dicha tarea de baja prioridad. En este caso está claro que el tiempo de bloqueo puede ser extremadamente alto, tanto como tarden en ejecutarse todas las tareas intermedias que puedan activarse, y por tanto no está acotado por la duración de las secciones críticas. Para resolver este problema y conseguir una inversión de prioridad acotada por la duración, generalmente pequeña, de las secciones críticas se recurre a los protocolos de sincronización que veremos a continuación, los cuales modifican temporalmente las propiedades de planificación de las tareas.

Otro efecto indeseado que los algoritmos de planificación tratan de evitar es el bloqueo en cadena. Esta situación se produce cuando una tarea de alta prioridad que comparte varios recursos con tareas menos prioritarias, trata de utilizarlos de forma sucesiva encontrándolos todos tomados, debiendo por tanto esperar en cada uno de ellos. Cada una de estas esperas supone una operación de cambio de contexto en la que la tarea bloqueada abandona el procesador para que sea entregado a la tarea más prioritaria en disposición de ejecutar en ese momento. Esta operación resulta relativamente costosa, por lo que conviene tener que realizarla el menor número de veces posible. Como veremos, existen protocolos que aseguran que cuando una tarea de alta prioridad comienza su ejecución, se encuentran libres todos los recursos que pueda precisar, o que como máximo deberá esperar en uno de ellos. En esos casos será posible reducir, e incluso eliminar, los cambios de contexto debidos al protocolo de sincronización.

También resultaría deseable evitar las situaciones de bloqueo mutuo, las cuales se producen cuando, en un conjunto de tareas que comparten recursos entre ellas, ninguna puede ejecutar puesto que todas necesitan un recurso que se encuentra en posesión de otra tarea del mismo conjunto.

1.3.1. Protocolos de sincronización para prioridades fijas

Con el objetivo principal de evitar la inversión de prioridad no acotada y el resto de problemas planteados, se han propuesto numerosos protocolos de sincronización, los cuales definen cómo deben modificarse los atributos de planificación de las tareas cuando se encuentran en posesión de uno o más recursos. Los principales protocolos de sincronización basados en políticas de prioridades fijas [SHA90] [RAJ89] se describen en los párrafos siguientes.

Uno de los más utilizados es el protocolo de "Techo de Prioridad Inmediato", también denominado protocolo de "Protección de Prioridad". Este protocolo asigna a cada recurso un

techo de prioridad igual a la mayor prioridad base de todas las tareas que vayan a usarle. Mientras que una tarea se encuentra en posesión de uno o más recursos, su prioridad activa se ve elevada a un valor igual al mayor de los techos de prioridad de esos recursos.

Este algoritmo elimina la inversión de prioridad no acotada que podrían sufrir las tareas de prioridad mayor que la que toma el recurso y además, en el caso de que las tareas no se suspendan mientras se encuentran en una sección crítica, presenta la ventaja de evitar las situaciones de bloqueo mutuo. También para esta misma situación de no suspensión, el protocolo presenta la ventaja de evitar el bloqueo en cadena, puesto que cuando una tarea comienza a ejecutar tiene la garantía de que encontrará libres todos los recursos que pueda precisar. Esta última propiedad permite simplificar en gran medida la implementación del protocolo cuando las tareas no se suspenden durante la duración de sus secciones críticas, ya que en ese caso no será necesario definir colas de tareas bloqueadas asociadas a los recursos.

Otro protocolo extensamente utilizado es el denominado “Herencia Básica de Prioridad”. Bajo este protocolo, una tarea hereda las prioridades de todas las tareas que se encuentran bloqueadas en los recursos que posee. De esta forma, una tarea en posesión de uno o más recursos tendrá una prioridad activa igual al mayor valor de entre su prioridad base y las prioridades activas de todas las tareas por ella bloqueadas.

Con este protocolo también se elimina totalmente la inversión de prioridad no acotada que podrían sufrir las tareas de mayor prioridad que la que toma el recurso, pero a cambio presenta la desventaja de no evitar el bloqueo mutuo y de presentar peores tiempos de respuesta de peor caso al no impedir los bloqueos en cadena.

Un algoritmo más complejo que los anteriores es el denominado “Techo de Prioridad”, también conocido como PCP (“Priority Ceiling Protocol”) [SHA90]. Las reglas que rigen este protocolo son las siguientes:

- A cada recurso le es asignado un techo de prioridad igual a la mayor prioridad de todas las tareas que vayan a usarle.
- Se define el techo de prioridad global para una tarea en un instante dado como el mayor de todos los techos de los recursos tomados por el resto de las tareas que componen la aplicación.
- Una tarea tomará un recurso si éste está libre y además su prioridad es estrictamente mayor que el techo global.
- Una tarea en posesión de un recurso hereda las prioridades tanto de las tareas bloqueadas en dicho recurso, como de las tareas bloqueadas indirectamente por ella en otros recursos debido al bloqueo de techo global.

Este protocolo aúna las principales ventajas de los protocolos de “Techo de Prioridad Inmediato” y de “Herencia Básica de Prioridad”, eliminando totalmente la inversión de prioridad no acotada y evitando el bloqueo mutuo incluso en el caso de que las tareas se bloqueen mientras se encuentran en posesión de algún recurso. Permite asegurar que una tarea se bloqueará como máximo en una sección crítica, con lo que evita el bloqueo en cadena aunque no de una forma tan eficaz como el protocolo de protección de prioridad, puesto que en este caso no se evitan totalmente los cambios de contexto debidos a la sincronización. Su principal desventaja radica en que su implementación es menos eficiente, debido principalmente a la complejidad añadida que supone el cálculo del techo de prioridad global.

1.3.2. Protocolos de sincronización para prioridades dinámicas

También para las políticas basadas en prioridades dinámicas se han definido protocolos de sincronización. El más sencillo es el denominado “Monitor Centralizado” [MOK84], que simplemente impone que la ejecución dentro de cualquier sección crítica sea no expulsable. Este protocolo presenta la importante desventaja de generar un importante número de bloqueos innecesarios, por lo que únicamente resulta aplicable en los casos en que las secciones críticas sean lo suficientemente cortas. Existen mejores soluciones, como la denominada “Stack Resource Protocol” (SRP) [BAK91] que constituye una solución equivalente al protocolo de “Techo de Prioridad Inmediato” para prioridades fijas que describíamos en el apartado anterior.

Una solución más compleja que las anteriores es el protocolo de “Techo de Prioridad Dinámico” [CHE90], que representa la solución equivalente al “Techo de Prioridad” (PCP) descrito en el apartado anterior, pero para un planificador EDF. Las reglas que rigen este protocolo son muy parecidas a las ya descritas para el PCP, aunque en este caso la prioridad de una tarea está determinada por el plazo de ejecución de su activación actual, tal y como dicta la política de planificación EDF:

- A cada recurso le es asignado un techo de prioridad igual al menor de los plazos de todas las tareas que le utilizan. Es importante notar que no se trata de un valor fijo, ya que el plazo de ejecución de las tareas cambia a medida que éstas van completando los trabajos correspondientes a activaciones anteriores.
- Se define el techo de prioridad global para una tarea en un instante dado como el mayor de todos los techos de los recursos tomados en ese instante por el resto de las tareas que componen la aplicación.
- Una tarea tomará un recurso si éste está libre y además su prioridad es estrictamente mayor que el techo global.
- Una tarea en posesión de un recurso hereda las prioridades tanto de las tareas bloqueadas en dicho recurso, como de las tareas bloqueadas indirectamente por ella en otros recursos debido al bloqueo de techo global. Esto es, la tarea bloqueante será tratada por el planificador EDF como si su plazo de ejecución fuera igual al menor de los plazos de todas las tareas que bloquea.

Al igual que su versión para prioridades estáticas (PCP), el protocolo de techo de prioridad dinámico también presenta las dos importantes propiedades de evitar el bloqueo mutuo y el bloqueo en cadena. Su principal desventaja radica en la su complejidad, incluso mayor que la del PCP debido a la variación de los techos de prioridad de los recursos.

1.4. Modelos de concurrencia

Los conceptos expuestos anteriormente: procesamiento concurrente y/o paralelo, políticas de planificación y protocolos de sincronización, se materializan en los sistemas operativos y en algunos lenguajes de programación (como Ada y Java) en la forma de modelos de concurrencia. En los siguientes subapartados vamos a realizar una pequeña descripción de los dos modelos que consideramos más interesantes por su potencia semántica y amplia utilización en sistemas de tiempo real: el descrito por el estándar POSIX y el utilizado por el lenguaje Ada 95.

1.4.1. Modelo de concurrencia POSIX

En las primeras fases de desarrollo del estándar POSIX, la concurrencia únicamente estaba soportada a nivel de procesos. Los procesos tienen espacios de direcciones independientes y un estado asociado muy voluminoso. Los tiempos de cambio de contexto entre procesos son muy elevados, puesto que normalmente requieren la programación de la unidad de gestión de memoria (MMU). También son muy costosas las operaciones de creación y destrucción. Por estas razones su utilización resulta inadecuada en aplicaciones concurrentes que requieran alta eficiencia, como es el caso de muchas aplicaciones empotradas, o en sistemas que no dispongan de MMU, como es el caso de los sistemas empotrados pequeños.

Para superar los problemas de falta de eficiencia presentados por los procesos, el estándar POSIX permite definir flujos de control concurrentes dentro de cada proceso, denominados "threads". Los threads comparten el espacio de direcciones con los demás threads de su proceso y tienen un estado asociado mucho menor que el de aquellos, por lo que presentan tiempos de cambio de contexto, creación y destrucción mucho menores.

El estándar define tres políticas de planificación para los procesos y los threads: FIFO con prioridades (`SCHED_FIFO`), rodaja temporal con prioridades o "round robin" (`SCHED_RR`) y servidor esporádico (`SCHED_SPORADIC`). Estas tres políticas son compatibles entre sí, de forma que pueden ser utilizadas por threads pertenecientes al mismo proceso. El estándar exige que al menos existan 32 niveles de prioridad diferentes para los threads.

Dos son las principales primitivas de sincronización definidas para los threads: los "mutexes" y las variables condicionales. Los mutexes permiten implementar secciones mutuamente exclusivas (el término "mutex" proviene de la expresión inglesa "mutual exclusion"). En todo momento un mutex sólo puede encontrarse en uno de estos dos estados: libre o tomado por un único thread (denominado propietario del mutex). Cualquier thread que trate de tomar un mutex que no se encuentra libre se quedará bloqueado en dicho mutex. Cuando el thread propietario libera el mutex, éste será tomado por el thread de mayor prioridad de entre los encolados en él, quedando libre en caso de que no hubiera ninguno. Para evitar la inversión de prioridad no acotada, los mutexes pueden utilizar los protocolos de herencia básica de prioridad (`PRIO_INHERIT`) o de protección de prioridad (`PRIO_PROTECT`). Con el primero de los protocolos, un thread hereda las prioridades de todos los threads bloqueados en mutexes de su propiedad, mientras que con el segundo, el thread propietario hereda los techos de prioridad asignados a todos los mutexes en su poder.

Por su parte, las variables condicionales son utilizadas para la espera y señalización de eventos entre threads. Se utilizan conjuntamente con un predicado lógico y un mutex, según el pseudocódico que mostramos a continuación:

```
toma mutex
while not predicado lógico loop
  espera en variable condicional
end loop
libera mutex
```

La activación de un thread bloqueado en una variable condicional se realiza desde otro thread que, mientras se encuentra en posesión del mutex, señaliza la variable condicional y posiblemente modifica el predicado lógico. El thread se activa estando en posesión del mutex, por lo que el chequeo del predicado lógico se hace de forma mutuamente exclusiva. Cuando el predicado lógico es verdadero, la tarea libera el mutex y continúa su ejecución.

1.4.2. Modelo de concurrencia Ada 95

La unidad de concurrencia en el lenguaje Ada es la tarea. El lenguaje, en su anexo de tiempo real, define una única política de planificación para las tareas: FIFO con prioridades (`FIFO_Within_Priorities`). Además, exige que existan al menos 30 niveles de prioridad distintos para asignar a las tareas.

El lenguaje Ada proporciona dos mecanismos de sincronización entre tareas: los "objetos protegidos" y el paso de mensajes o "rendezvous". Ambos constituyen mecanismos de más alto nivel de abstracción que los mutexes y las variables condicionales.

Los objetos protegidos permiten proteger un recurso compartido ante accesos simultáneos realizados por varias tareas. Asimismo, también permiten la sincronización de señalización y espera entre tareas. Un objeto protegido engloba la definición de la información compartida junto con el conjunto de operaciones que constituyen el único medio de acceso a dicha información. Es posible definir tres tipos de operaciones: las funciones, que constituyen operaciones de lectura de la información; los procedimientos, desde los que es posible modificar la información y las entradas ("entry"), que son iguales que los procedimientos salvo que además se encuentran protegidas por un predicado lógico (barrera). Mientras se está ejecutando un procedimiento o entrada, ninguna otra tarea puede acceder al objeto protegido. Por el contrario, el modelo permite que varias tareas ejecuten funciones de forma simultánea.

Las entradas permiten implementar sincronización de señalización y espera. Para que una tarea pueda ejecutar una entrada de un objeto protegido, además de que éste se encuentre libre, deberá ser verdadero el predicado lógico que la protege. En caso contrario, la tarea permanecerá bloqueada en la entrada. Las barreras son reevaluadas cada vez que una tarea finaliza la ejecución de un procedimiento o entrada de ese objeto protegido.

Los objetos protegidos evitan el problema de la inversión de prioridad no acotada ya que implementan el protocolo de techo de prioridad inmediato. Así, mientras una tarea se encuentra ejecutando una operación protegida, ve elevada su prioridad a la del techo del objeto protegido.

El "rendezvous" (encuentro) es un mecanismo de sincronización consistente en la invocación desde una tarea de una entrada ("entry") de otra. Para que se produzca, ambas tareas deben estar dispuestas para la sincronización, una esperando en una instrucción `accept` y la otra invocando la entrada correspondiente. La tarea que alcanza primero el punto de sincronización se bloquea a la espera de que éste también sea alcanzado por la otra. Tras producirse el encuentro, la tarea llamante permanece suspendida hasta que la tarea llamada finalice la ejecución del código correspondiente a la aceptación de la entrada. Las entradas de una tarea pueden tener parámetros de entrada, salida o entrada/salida, por lo que la transmisión de información entre tareas puede realizarse en ambos sentidos.

1.5. Objetivos de la tesis

Como se ha expuesto anteriormente, existe una amplísima variedad de políticas de planificación y protocolos de sincronización, presentando cada uno de ellos ventajas e inconvenientes que le hacen apropiado para un determinado tipo de aplicaciones e inadecuado para otras. De entre esta amplia variedad, sólo un pequeño subconjunto de algoritmos y protocolos está soportado en los sistemas operativos y lenguajes de programación existentes en la actualidad.

Resultaría posible ampliar el número de políticas y protocolos soportados por los sistemas operativos y lenguajes de programación, aunque no parece factible que con un pequeño

subconjunto de ellas se puedan satisfacer los requerimientos de todo tipo de aplicaciones. En todo caso, y aunque tal subconjunto existiera, resultaría pronto insuficiente al surgir nuevos ámbitos de aplicación con nuevos requisitos de planificación no contemplados inicialmente. Una solución general al problema de la planificación flexible pasaría por que los sistemas operativos y/o los lenguajes de programación proporcionaran un mecanismo que permitiera a las aplicaciones definir sus propios algoritmos de planificación y protocolos de sincronización.

Siguiendo esta línea de razonamiento, los principales objetivos perseguidos en esta tesis han sido:

- Desarrollar una interfaz que permita a las aplicaciones definir los algoritmos de planificación bajo los cuales pretenden que sean planificadas sus tareas.
- Extender la interfaz para que englobe la gestión de recursos compartidos, permitiendo a las aplicaciones definir los protocolos de sincronización que mejor se adapten a cada política de planificación.
- Implementar la interfaz anteriormente mencionada en un sistema operativo POSIX de tiempo real, de forma que sea posible evaluar aspectos como su facilidad de uso, complejidad de implementación, sobrecarga introducida, etc.

Para la interfaz se ha establecido un conjunto de requerimientos cuyo cumplimiento constituirá una mejora sobre las soluciones existentes hasta la fecha. A continuación procedemos a enumerar los más importantes de dichos requerimientos:

- Deberá perseguir la compatibilidad y la estandarización: la interfaz será diseñada como una ampliación de las interfaces POSIX (en sus versiones Ada y C); además, los algoritmos de planificación definidos con ella deberán ser compatibles con los existentes en el POSIX y en el Ada 95.
- Deberá englobar la gestión de los recursos compartidos, lo que es fundamental para lograr que las aplicaciones puedan cumplir sus requisitos temporales, ya que, según lo expuesto en el apartado 1.3, “Protocolos de sincronización”, la utilización de protocolos de sincronización inadecuados puede provocar efectos como la inversión de prioridad no acotada.
- Deberá constituir una solución general que permita implementar una amplia variedad de políticas de planificación y protocolos de sincronización.
- Permitirá implementar algoritmos de planificación para sistemas multiprocesadores.
- Permitirá aislar las políticas de planificación para evitar que un fallo en una de ellas pueda afectar a las demás o al resto del sistema.

La integración y compatibilidad con POSIX facilitará el uso de la interfaz por programadores familiarizados con este estándar, en número creciente debido su alto grado de aceptación entre los fabricantes de sistemas operativos. Además, podría permitir su incorporación en una futura revisión del estándar, lo que constituiría el objetivo final marcado para la interfaz, lógicamente ya fuera del marco de esta tesis doctoral.

A pesar de que son numerosos los sistemas operativos que presentan una interfaz POSIX, ninguno de ellos parecía apropiado para la implementación de la interfaz para planificación definida por la aplicación, ya que, como se expondrá con más detalle en la introducción del capítulo 3, en general se trata de sistemas propietarios que no proporcionan su código fuente. Ciertamente, también existen algunos sistemas operativos de código abierto que presentan una interfaz POSIX, pero en general se trata de sistemas que no siguen internamente el modelo de threads POSIX, lo que dificulta la realización de modificaciones basadas en este estándar.

Por esa razón, decidimos añadir a los objetivos de esta tesis el diseño e implementación de un sistema operativo de tiempo real con el principal propósito de que sirviese como base sobre la que probar la interfaz anteriormente mencionada. Este sistema operativo, que hemos denominado MaRTE OS (Minimal Real-Time Operating System for Embedded Applications), está pensado para aplicaciones empotradas, siendo compatible con el perfil mínimo de sistema de tiempo real definido en el estándar POSIX.13. Con el diseño e implementación de MaRTE OS se han perseguido otra serie de objetivos secundarios:

- Evaluar el perfil mínimo de sistema de tiempo real definido en el POSIX.13, del que MaRTE OS constituye una de las primeras implementaciones.
- Demostrar que resulta posible implementar un sistema operativo pequeño y eficiente utilizando el lenguaje de programación Ada 95, con lo que es posible aprovechar las características de fiabilidad y legibilidad de este lenguaje.
- Evaluar la complejidad que supondría incorporar en el perfil mínimo servicios POSIX recientemente aprobados y que se consideran interesantes para los sistemas empotrados de tiempo real. En particular la política de planificación de servidor esporádico, la suspensión temporizada absoluta de alta resolución, el reloj monótono y los relojes y temporizadores de tiempo de ejecución.
- Generar un sistema operativo libre que pueda ser utilizado como herramienta de docencia e investigación en sistemas operativos de tiempo real, así como para el desarrollo de aplicaciones industriales.

1.6. Organización de la memoria de tesis doctoral

En el capítulo 2 se exponen los principales antecedentes en los que se apoya el trabajo de investigación presentado en esta memoria: los estándares POSIX y Ada 95 y las soluciones adoptadas hasta la fecha para permitir a las aplicaciones definir sus propios algoritmos de planificación. En primer lugar se hace un repaso general de la familia de estándares POSIX, para posteriormente centrarse en el perfil del “Sistema de Tiempo Real Mínimo” que está pensado para aplicaciones empotradas en sistemas pequeños. Este perfil incluye un reducido subconjunto de servicios POSIX con el que es posible implementar un núcleo de sistema operativo pequeño y eficiente apropiado para pequeñas aplicaciones empotradas. En el apartado 2.2 se procede a comentar las principales características del lenguaje de programación Ada 95, que será el utilizado para la implementación del sistema operativo MaRTE OS. Por su parte, en el apartado 2.3 se exponen las razones por las que se están buscando mecanismos para permitir a las aplicaciones definir sus propios algoritmos de planificación. Se realiza un resumen de las soluciones encontradas hasta la fecha, planteándose sus principales carencias, las cuales se pretenden superar con la interfaz presentada en los capítulos 4, “Interfaz de usuario para la definición de algoritmos de planificación” y 5, “Interfaz de usuario para la definición de protocolos de sincronización”.

En el capítulo 3 se presenta el sistema operativo MaRTE OS (Minimal Real-Time Operating System for Embedded Applications). MaRTE OS es un sistema operativo de tiempo real que ha sido diseñado y desarrollado en el marco de esta tesis doctoral con el objetivo de que sirva como base sobre la que implementar y probar los servicios definidos en los capítulos 4, “Interfaz de usuario para la definición de algoritmos de planificación” y 5, “Interfaz de usuario para la definición de protocolos de sincronización”. MaRTE OS sigue el perfil de sistema mínimo de tiempo real definido en el estándar POSIX.13 [PSX98] y está escrito utilizando en lenguaje de programación Ada 95. Permite la ejecución de aplicaciones Ada y C, para lo cual el sistema proporciona las interfaces POSIX en ambos lenguajes. El sistema de tiempo de ejecución del

compilador GNAT ha sido adaptado para ejecutarse sobre MaRTE OS, de modo que las aplicaciones Ada pueden hacer uso del completo soporte para programación concurrente proporcionado por este lenguaje. En este capítulo se detallan las principales características de MaRTE OS, justificándose las principales decisiones tomadas en la fase de diseño. Posteriormente, en el apartado 3.4, se procede a la descripción de la arquitectura del núcleo de MaRTE OS, describiéndose las principales capas software que le componen. También se comentan los puntos más importantes en la implementación de los distintos servicios proporcionados por el sistema operativo. Las prestaciones y tamaño del sistema operativo son expuestas en el apartado 3.8. Junto con el núcleo de MaRTE OS se han desarrollado un conjunto de herramientas que posibilitan la creación, carga en el equipo de ejecución y depuración de las aplicaciones. Dichas herramientas componen el entorno de desarrollo de aplicaciones que se describe en el apartado 3.9. Finalmente se exponen las principales conclusiones obtenidas del diseño e implementación de MaRTE OS. Entre ellas se proponen algunas modificaciones al perfil de sistema de tiempo real mínimo después de sopesar las utilidad de los servicios frente a la complejidad que supone su implementación en MaRTE OS.

En el capítulo 4 se describe una interfaz que permite a las aplicaciones definir sus propios algoritmos de planificación. Esta interfaz, implementada en el sistema operativo MaRTE OS, tiene como principales características el ser compatible con los protocolos de planificación estándares (definidos en el POSIX y en el Ada) y permitir el aislamiento de los planificadores del resto del sistema. Estas dos características permiten solventar las principales carencias detectadas en el apartado 2.3, “Soluciones para la planificación a nivel de aplicación” para las soluciones existentes hasta la fecha. En primer lugar, en este capítulo se describen las principales características de la interfaz, para posteriormente presentar el modelo en el que se basa, cuya principal característica es que define un tipo especial de tareas, denominadas planificadoras de aplicación, que tienen el poder de activar y suspender a sus tareas planificadas. En los apartados 4.3 y 4.4 se describen respectivamente las interfaces C y Ada para planificación definida por la aplicación. Posteriormente, se presentan algunos ejemplos de algoritmos de planificación implementados utilizando la interfaz con el fin de demostrar su versatilidad a la hora de implementar algoritmos de diversos tipos. A continuación, en el apartado 4.6, se comentan algunos detalles sobre la implementación de la interfaz en MaRTE OS. Finalmente se evalúa la sobrecarga impuesta por la interfaz sobre las aplicaciones que la utilizan, obteniéndose que es lo suficientemente pequeña como para justificar su utilización, más si se tiene en cuenta las ventajas que proporciona al programador al permitirle definir algoritmos de planificación de una forma flexible y potencialmente portable.

En el capítulo 5 se amplía la interfaz presentada en el capítulo anterior, para que englobe la gestión de recursos compartidos de forma que resulte compatible con los protocolos de sincronización definidos en el estándar POSIX. Con esta ampliación las tareas planificadoras pueden definir sus propios protocolos de sincronización adaptados a la política de planificación que implementan, de forma que es posible evitar la inversión de prioridad no acotada y otros efectos similares e indeseados. Con ella se supera una de las principales carencias presentadas por las soluciones existentes hasta la fecha, analizadas en el apartado 2.3, “Soluciones para la planificación a nivel de aplicación”. Este capítulo comienza con la enumeración y justificación de los requerimientos marcados para la interfaz. A continuación se presenta el modelo en el que se basa, en el que se definen dos tipos de mutexes: los de protocolo definido por el sistema y los de protocolo definido por la aplicación. Las interfaces C y Ada se describen en los apartados 5.3 y 5.4 respectivamente, y algunos ejemplos del uso de las citadas interfaces se presentan en el apartado 5.5. A continuación se comentan algunos puntos relevantes de la implementación de la interfaz en MaRTE OS, para finalizar con la evaluación de la sobrecarga impuesta por el uso de la interfaz y las conclusiones finales de este capítulo.

Finalmente, en el capítulo 6, se presentan las principales conclusiones del trabajo de investigación presentado y se esbozan algunas líneas de investigación futuras.

Además, como anexo A de está memoria se incluye la definición de la interfaz C, que engloba a las descritas en los capítulos 4, “Interfaz de usuario para la definición de algoritmos de planificación” y 5, “Interfaz de usuario para la definición de protocolos de sincronización”. Frente a la descripción que de esta interfaz se hace en los citados capítulos, el anexo constituye una definición formal y completa, escrita en ingles y realizada siguiendo el estilo utilizado en los documentos de los estándares POSIX, al objeto de conseguir su máxima difusión.

2. Antecedentes

2.1. Subconjunto POSIX de sistema de tiempo real mínimo

2.1.1. La familia de estándares POSIX

A finales de la década de los ochenta el sistema operativo Unix constituía un estándar industrial *de facto*, aunque existían notables diferencias entre las versiones de los distintos fabricantes. Este hecho impulsó a fabricantes y usuarios a patrocinar la creación de un estándar internacional que permitiera la unificación de las versiones existentes. Así nació el estándar POSIX, acrónimo de Interfaz de Sistemas Operativos Portables ("Portable Operating Systems Interface"), desarrollado en el marco de la Computer Society de IEEE con la referencia IEEE 1003 y que también constituye un estándar internacional con la referencia ISO/IEC 9945. El estándar define la interfaz que los sistemas operativos deben presentar a las aplicaciones, así como la semántica de cada uno de los servicios ofrecidos. Los principales objetivos que persigue son:

- Lograr la portabilidad de las aplicaciones a nivel de código fuente entre los distintos sistemas operativos.
- Facilitar la adaptación de los programadores a nuevos entornos de trabajo.

El POSIX es una familia de estándares en evolución que pueden ser agrupados en tres categorías:

1. *Estándares base*: definen la interfaz (sintaxis y semántica) que presenta el sistema operativo a las aplicaciones, esto es, el conjunto de funciones, escritas en lenguaje C, que permiten a los programas acceder a los diferentes servicios facilitados por el sistema operativo. Es importante resaltar que no se define la forma de implementación de las citadas funciones, la cual es dejada a la elección del fabricante del sistema operativo. La tabla 2.1 muestra algunos de los estándares básicos de POSIX.

Tabla 2.1: Algunos estándares base de POSIX

Estándar	Descripción
1003.1	Servicios básicos del sistema operativo
1003.1a	Extensiones a los servicios básicos
1003.1b	Extensiones de tiempo real a los servicios básicos
1003.1c	Extensión de threads
1003.1d	Extensiones adicionales de tiempo real
1003.1g	Comunicaciones por red
1003.1j	Extensiones de tiempo real avanzadas

Tabla 2.1: Algunos estándares base de POSIX (cont.)

Estándar	Descripción
1003.3	Métodos para probar la conformidad con POSIX
1003.21	Comunicaciones para sistemas distribuidos de tiempo real

2. *Interfaces en diferentes lenguajes de programación* (“bindings”): estos estándares proporcionan interfaces a los mismos servicios definidos en los estándares base, pero utilizando lenguajes de programación diferentes. Los lenguajes que se han usado hasta el momento son Ada y Fortran 77. En la tabla 2.2 se muestran los más importantes.

Tabla 2.2: Algunos de los “bindings” POSIX

Estándar	Descripción
1003.5	“Binding” de Ada para el estándar 1003.1
1003.5b	“Binding” de Ada para los estándares 1003.1b y 1003.c
1003.5c	“Binding” de Ada para el estándar 1003.1g
1003.5g	Revisión que agrupa todos los estándares anteriores (5, 5b y 5c)
1003.5f	“Binding” de Ada para el estándar 1003.21
1003.9	“Binding” de Fortran 77 para el estándar 1003.1

3. *Perfiles de entornos de aplicación*: definen subconjuntos del POSIX que contienen los servicios necesarios para un ámbito de aplicación determinado. Con su definición se evita la necesidad de implementar todo el POSIX en sistemas operativos que, por estar destinados a un área muy específica, no requieren parte de la funcionalidad descrita en el estándar.

La figura 2.1 muestra la evolución histórica de parte de la familia de estándares POSIX, centrándose principalmente en los relacionados con el tiempo real. En 1988 y posteriormente en 1990 se aprobó el estándar básico (1003.1) que estandarizaba los principales servicios facilitados por un sistema operativo Unix convencional. Estos servicios se centran en dos conceptos fundamentales: los procesos y los ficheros. Los procesos permiten dar soporte a la ejecución concurrente de aplicaciones en espacios de direcciones independientes. Por su parte, los ficheros representan los distintos objetos del sistema operativo sobre los que es posible realizar operaciones de lectura y escritura. Los servicios definidos en el POSIX 1003.1 incluyen la gestión de procesos, identificación y entorno de procesos, notificación de eventos mediante el uso de señales, algunos servicios de temporización muy primitivos, servicios de manejo de ficheros y directorios, dispositivos de entrada/salida, control de terminales y bases de datos del sistema para usuarios y grupos.

En 1993 se aprobó el estándar POSIX 1003.1b [PSX93], en el que se definían extensiones de tiempo real para los servicios facilitados por el POSIX 1003.1, de forma que fuera posible ejecutar en un sistema POSIX aplicaciones con requisitos temporales. Los servicios descritos en este estándar se pueden agrupar en dos categorías:

- *Servicios que facilitan la programación concurrente*: son necesarios porque, como ya se expuso en la introducción de esta memoria, una de las características de las aplicaciones

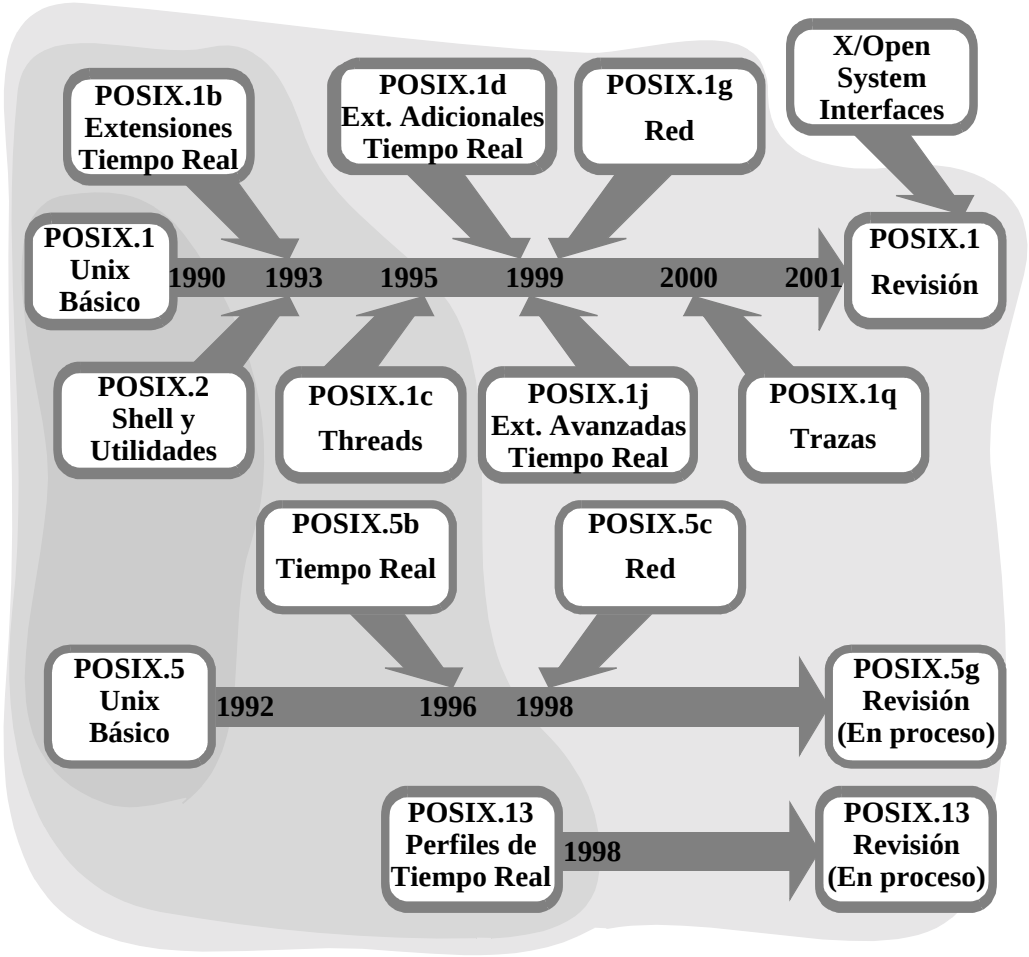

Figura 2.1: Evolución histórica de la familia de estándares POSIX

de tiempo real es que son concurrentes y que sus procesos necesitan cooperar estrechamente entre sí. En este estándar se definen primitivas de sincronización entre procesos como son los semáforos contadores, los objetos de memoria compartida que permiten a los procesos compartir información y las colas de mensajes que permiten transmitir eventos e información entre ellos. También se definen primitivas para la entrada/salida asíncrona, que permite a la aplicación ejecutar en paralelo con las operaciones de entrada/salida y también para la entrada/salida sincronizada, que permite un mayor grado de predecibilidad temporal en dichas operaciones.

- *Servicios necesarios para conseguir un comportamiento temporal predecible*: incluyen la planificación expulsora de procesos mediante políticas basadas en prioridades fijas, la inhibición de la memoria virtual para el espacio de direcciones de un proceso, que permite conseguir tiempos de acceso a memoria predecibles, las señales de tiempo real, que tienen un comportamiento más predecible que las demás señales y, por último, se definen relojes y temporizadores que permiten mejorar la gestión del tiempo desde la aplicación.

Un hito importante en el POSIX de tiempo real lo constituye la extensión de threads (1003.1c) aprobada en 1995 e incluida, junto con el resto de estándares POSIX aprobados hasta esa fecha, en el estándar ISO/IEC 9945-1:1996 [PSX96]. Las unidades de concurrencia definidas en el POSIX básico son los procesos, los cuales tienen espacios de direcciones independientes para proporcionar la protección e independencia entre aplicaciones, necesarias en un sistema

operativo multipropósito. Sin embargo, su utilización resulta inadecuada en aplicaciones concurrentes que requieran alta eficiencia. El problema es que los tiempos de creación, destrucción y cambio de contexto entre procesos son muy grandes debido a que éstos tienen un estado asociado muy voluminoso y a que normalmente requieren la programación de la unidad de gestión de memoria (MMU). Por tanto, con el propósito de mejorar la eficiencia, se introdujeron en el POSIX los threads o procesos ligeros. Con esta extensión cada proceso POSIX puede tener varios flujos de control concurrentes, todos ellos compartiendo el mismo espacio de direcciones. El estándar proporciona servicios para la gestión, cancelación, planificación y sincronización de threads.

En 1999 se aprobaron las extensiones adicionales y las extensiones avanzadas de tiempo real (estándares 1003.1d [PSX99d] y 1003.1j [PSX99j]). En ellos se definen interfaces para servicios que pueden resultar de gran utilidad para muchas aplicaciones de tiempo real. Así en el POSIX.1d se incluye el arranque rápido de procesos, los tiempos límites en servicios bloqueantes, la medida y limitación de tiempos de ejecución, la política de planificación de servidor esporádico e información para la implementación de ficheros de tiempo real. Por su parte, en el POSIX.1j se definen primitivas adicionales para sincronización en multiprocesadores, gestión de memoria de diversos tipos y los relojes monótono y sincronizado.

En el mismo año que los estándares 1003.1d y 1003.1j, fue también aprobado el estándar que define la interfaz de acceso a los protocolos de red por parte de las aplicaciones (1003.1g). Aunque, por supuesto, este estándar no está pensado específicamente para aplicaciones de tiempo real distribuidas, también puede ser utilizado por ellas siempre que el protocolo de comunicación utilizado tenga un comportamiento temporal predecible.

También la funcionalidad definida en el estándar de trazas (1003.1q) [PSX00q] puede resultar interesante para las aplicaciones de tiempo real. Este estándar proporciona una interfaz que permite recabar información sobre las acciones realizadas por los programas de usuario y el propio sistema operativo, tales como cambios de contexto, llamadas al sistema u otros eventos definidos por el usuario. Los datos obtenidos pueden ser analizados directamente por un proceso o almacenados para un análisis posterior. Esta funcionalidad puede permitir obtener información sobre parámetros tan importantes para un sistema de tiempo real como son los tiempos de ejecución o la sobrecarga provocada por el sistema operativo.

En el año 2001 se ha realizado una revisión del estándar POSIX [PSX01] en la que se integran todos los estándares anteriormente mencionados. Además en el citado documento el estándar POSIX converge con estándar X/Open, impulsado por el consorcio “The Open Group” [OPENG], el cual constituía un estándar *de facto* en la industria que incluía las interfaces POSIX junto a otro conjunto de servicios adicionales.

Desde la aprobación de las extensiones de tiempo real y de threads (años 1993 y 1995 respectivamente), es posible escribir aplicaciones con requisitos de tiempo real sobre sistemas operativos POSIX. Sin embargo, un sistema operativo que implemente el estándar POSIX en su totalidad es un sistema muy grande y complejo, que casi con toda probabilidad no podrá ser utilizado en computadores empotrados pequeños. Por esta razón, y para que el POSIX sea aplicable a este y otros entornos restringidos, en el estándar POSIX.13 [PSX98] se han definido cuatro subconjuntos de servicios del sistema operativo (perfiles de entornos de aplicación), que se han orientado a cuatro tipos de plataformas comúnmente utilizadas en aplicaciones comerciales de tiempo real.

Existen muchas pequeñas diferencias entre los citados perfiles, aunque las principales radican en la existencia o no de múltiples procesos y de un sistema de ficheros complejo y jerárquico.

La tabla 2.3 muestra las diferencias de los cuatro perfiles de tiempo real respecto a las citadas características.

Tabla 2.3: Características de los perfiles POSIX de tiempo real

Perfil	*Sistema de Ficheros*	*Múltiples Procesos*	*Threads*
Sistema de Tiempo Real Mínimo	NO	NO	SI
Controlador de Tiempo Real	SI	NO	SI
Sistema de Tiempo Real Dedicado	NO	SI	SI
Sistema de Tiempo Real Multi-Propósito	SI	SI	SI

El menor de dichos perfiles, el denominado “Sistema de Tiempo Real Mínimo” [GON97A], está pensado para aplicaciones empotradas pequeñas. Entre otras características, no requiere el soporte para múltiples procesos ni para un sistema de ficheros jerárquico. Con estas dos simplificaciones se elimina la mayor parte de la complejidad de un sistema POSIX completo, permitiendo por tanto la implementación de un núcleo de sistema operativo pequeño y eficiente, apropiado para pequeñas aplicaciones empotradas. Aplicaciones típicas de este perfil serían el control de electrodomésticos (como microondas o televisores), terminales de punto de venta, controladores de motores de automóviles, controladores de sensores industriales, etc. De este perfil se realiza una descripción mucho más detallada en el apartado 2.1.2.

El siguiente perfil por orden de complejidad es el “Controlador de Tiempo Real”. Está pensado para controladores de propósito especial, los cuales no disponen de MMU pero sí de disco que contiene un sistema de ficheros simplificado. Un ejemplo de aplicación que encajaría en este perfil es el controlador de un robot industrial.

Por su parte, el denominado “Sistema de Tiempo Real Dedicado” se trata de un sistema empotrado grande sin disco, en el que se desea ejecutar una aplicación empotrada compleja que requiere múltiples procesos (dispone de MMU) y comunicaciones por red. Corresponde a aplicaciones del tipo de controladores de aviones o de nudos de control de teléfonos móviles.

El mayor de los perfiles es el denominado “Sistema de Tiempo Real Multipropósito”. Se trata de un sistema de tiempo real grande, que puede ser utilizado a la vez como entorno de desarrollo. Dispone de comunicaciones por red, sistema de ficheros jerárquico en disco, terminales gráficos de interfaz con el usuario, etc. Se trata, en definitiva, de una estación de trabajo con requerimientos de tiempo real. Aplicaciones típicas de este tipo de plataformas son los sistemas de control del tráfico aéreo o los equipos de telemetría empleados para los coches de Fórmula 1.

En la actualidad se encuentra en fase de borrador la revisión de los perfiles de sistema de tiempo real [PSX02]. En ella se plantea la incorporación de muchos de los nuevos servicios añadidos en el POSIX.1-2001, en particular los provenientes de las extensiones adicionales de tiempo real (POSIX.1d), trazas (POSIX.1q), redes (POSIX.1g) y la interfaz X/Open. También se pretende añadir al “Sistema de Tiempo Real Dedicado” un sistema de ficheros básico y ficheros mapeados en memoria, el motivo de esta incorporación es la creciente utilización de sistemas de almacenamiento basados en memoria no volátil, como por ejemplo la “Flash RAM”. Por otra parte se plantea la eliminación de la entrada/salida asíncrona en el perfil “Controlador de Tiempo Real”, así como la eliminación de las colas de mensajes y la librería matemática C en

el perfil “Sistema de Tiempo Real Mínimo”. La figura 2.2 presenta una visión esquemática de las funcionalidades que pretenden ser incluidas en cada uno de los perfiles en la citada revisión. Como puede apreciarse, cada perfil contiene a los anteriores, lo que permite la portabilidad de las aplicaciones hacia perfiles “mayores”. Gracias a esto es posible probar la aplicación a nivel funcional en una estación de trabajo con requisitos de tiempo real, en lugar de tener que hacer todas las pruebas sobre el hardware final o un emulador.

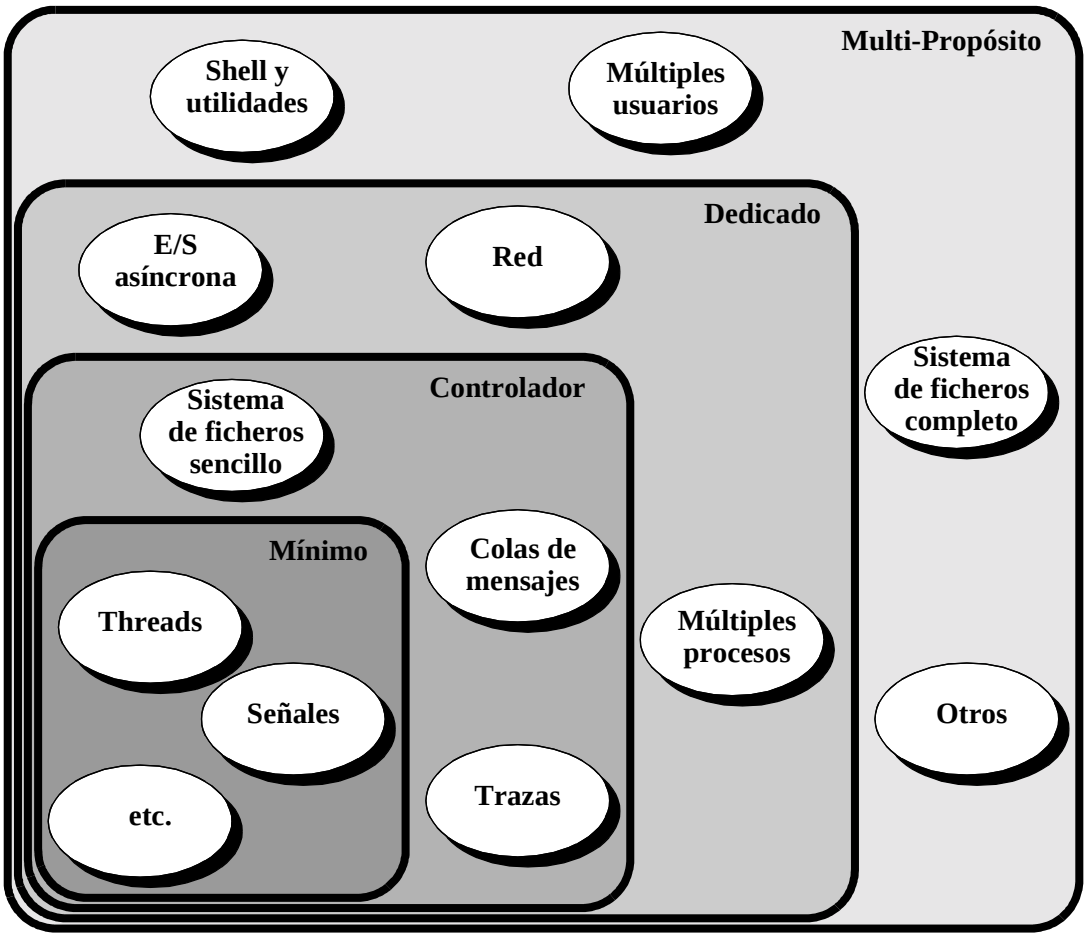

Figura 2.2: Funcionalidad contenida por los perfiles de tiempo real

2.1.2. Descripción del sistema de tiempo real mínimo

Como mencionamos anteriormente, el perfil de “Sistema de Tiempo Real Mínimo”, está pensado para aplicaciones empotradas pequeñas, por lo que en él se incluyen únicamente aquellos servicios POSIX que se consideran necesarios para las aplicaciones a las que está destinado. Con el reducido conjunto de servicios soportados, el estándar POSIX.13 pretende que un núcleo de sistema operativo compatible con el “Sistema de Tiempo Real Mínimo” pueda ser de muy reducido tamaño y altamente eficiente. A continuación procederemos a describir los principales servicios incluidos en este perfil:

- ***Soporte para ejecución concurrente***. El único mecanismo de ejecución concurrente facilitado son los denominados hilos de control o threads. Se incluyen servicios para su creación y terminación, así como para la gestión de sus atributos. También existe la posibilidad de asociar información específica con los threads. Otra funcionalidad incluida en este perfil permite a un thread esperar la terminación de otro.

- ***Planificación de threads.*** La política de planificación de threads es expulsora por prioridades fijas, pudiendo especificarse dos comportamientos diferentes entre threads de la misma prioridad: FIFO o "round robin". Un thread de política FIFO (`SCHED_FIFO`) se mantendrá en ejecución hasta que se bloquee o hasta que sea expulsado por otro thread de mayor prioridad. Los threads con política "round robin" (`SCHED_RR`) se comportan igual que los FIFO salvo por una limitación adicional: sólo se pueden mantener en ejecución de forma continuada durante un intervalo máximo de tiempo definido por la implementación. Al finalizar dicho intervalo el thread deberá ceder la CPU a otros threads de su misma prioridad, sólo en el caso de que no haya ninguno, el thread original retomará la CPU para comenzar otro intervalo.

  El número mínimo de niveles de prioridad diferentes requerido por el estándar es 32. Con este número es posible lograr altos niveles de utilización del procesador incluso para números muy elevados de threads [SHA91]. El perfil mínimo incluye también servicios que permiten especificar los parámetros y políticas de planificación, para lo cual el estándar define los siguientes atributos:

  - Política de planificación: pudiéndose elegir entre las políticas FIFO y "round robin".
  - Parámetros de planificación: para las políticas anteriormente citadas el único parámetro definido es su prioridad.
  - Herencia de los atributos de planificación: permite indicar si los atributos serán heredados del thread padre o son los indicados explícitamente por su objeto de atributos.
- ***Sincronización entre threads.*** Los servicios de sincronización incluidos en el perfil mínimo son los mutexes, las variables condicionales y los semáforos contadores. Los mutexes son utilizados para asegurar la exclusión mutua entre threads accediendo a un mismo recurso. Para ellos se definen dos protocolos que permiten evitar la inversión de prioridad: la herencia básica de prioridad (`PTHREAD_PRIO_INHERIT`) y el techo de protección inmediato (`PTHREAD_PRIO_PROTECT`). Por su parte las variables condicionales permiten implementar mecanismos de sincronización entre threads basados en señalización y espera. Los semáforos contadores constituyen un mecanismo de sincronización para exclusión mutua y señalización y espera menos evolucionado que los anteriores, adoleciendo de inversión de prioridad y resultando su utilización más compleja y más proclive a errores.
- ***Señales***. Las señales POSIX son utilizadas para notificar a la aplicación la ocurrencia de un evento. El perfil mínimo proporciona servicios para enviar, esperar y enmascarar señales, soportando también la instalación de manejadores de señal. Sin embargo, en sistemas en los que existe concurrencia, la ejecución asíncrona de manejadores es preferiblemente reemplazada por operaciones de espera síncrona. La segunda estrategia tiene la ventaja de que los parámetros de planificación bajo los cuales es servida una señal están determinados por los atributos del thread encargado de su gestión.
- ***Acceso a dispositivos***. Aunque el perfil mínimo no requiere la existencia de un sistema de ficheros completo, sí que se incluyen las operaciones básicas que permiten acceder a los dispositivos (`open`, `close`, `write`, `read`). La operación `open` está restringida a la apertura de ficheros ya existentes, definidos durante la configuración del sistema, no pudiendo ser usada para crear nuevos ficheros.
- ***Servicios de temporización***. Se incluye una operación de suspensión temporizada de tareas en la que el intervalo de suspensión puede ser especificado con alta precisión. También existen operaciones para la lectura de un reloj de alta resolución y para manejo

de temporizadores. Los temporizadores sirven para informar a la aplicación de que un intervalo de tiempo dado ha transcurrido o de que se ha alcanzado un tiempo determinado. En ambos casos se puede especificar que se desea utilizar una señal como forma de notificación de la expiración del temporizador.

- ***Paso de mensajes***. Se encuentran definidas operaciones para utilizar el sistema de colas de mensajes. Los mensajes incluyen un campo de prioridad que es usado para indicar el orden de recepción. En la revisión actualmente en marcha de este estándar, se plantea la eliminación de este servicio en el perfil de sistema mínimo de tiempo real debido a la escasa utilización que de él hacen la mayoría de las aplicaciones empotradas de tiempo real. Por otra parte, en el caso de que se desee utilizar las colas de mensajes, éstas pueden ser fácilmente implementadas por la aplicación utilizando mutexes y variables condicionales.
- ***Servicios de configuración***. Se proporcionan operaciones que permiten a la aplicación conocer información relativa a la configuración del sistema.
- ***Gestión de memoria dinámica***. Aunque el estándar POSIX no define ninguna interfaz para la gestión de memoria dinámica, si que requiere que estén soportadas las operaciones de memoria dinámica específicas de los lenguajes de programación, como `malloc()` para el lenguaje C o `new` para el Ada.

Además de los servicios anteriormente citados, el estándar POSIX.13 requiere que algunos otros servicios también sean soportados en el perfil mínimo por razones de compatibilidad con los demás perfiles:

- ***Bloqueo de memoria***. Estos servicios evitan los tiempos de acceso a memoria no acotados debidos a la utilización de memoria virtual, permitiendo mantener en memoria física el espacio de direcciones de los procesos de tiempo real. Aunque lo normal en las implementaciones de sistemas mínimos de tiempo real es que no soporten memoria virtual, la interfaz es incluida por razones de compatibilidad con el resto de los perfiles. Se recomienda a las aplicaciones la utilización de estos servicios incluso en sistemas en los que su implementación sea nula.
- ***Entrada/Salida sincronizada y no sincronizada***. Lo normal en sistemas empotrados pequeños será que la E/S se haga directamente sobre el dispositivo sin ningún tipo de almacenamiento intermedio, por tanto toda la E/S en este tipo de sistemas será sincronizada por omisión. Aún así, y al igual que en caso anterior, se recomienda a las aplicaciones que expresamente requieran E/S sincronizada para mantener la compatibilidad con los restantes perfiles.
- ***Objetos de memoria compartida***. Pueden ser utilizados para implementar E/S mapeada en memoria, lo cual es muy común en muchas arquitecturas empotradas. Aunque la memoria pudiera ser directamente accesible por las aplicaciones, se recomienda realizar la E/S mapeada en memoria utilizando objetos de memoria compartida. De esta forma resultará posible la portabilidad de la aplicación a sistemas mayores en los que está prohibido el acceso directo a dispositivos mapeados en memoria. La implementación de los servicios descritos en sistemas sin protección, en los que toda la memoria es accesible, resulta extremadamente sencilla.

El estándar POSIX.13 define formalmente cada perfil de sistema de tiempo real en base a las unidades funcionales que incluye y las opciones que soporta. La tabla 2.4 muestra las unidades funcionales del estándar POSIX.1 que se encuentran incluidas en el perfil de sistema de tiempo real mínimo.

Tabla 2.4: Unidades funcionales incluidas en el perfil de sistema de tiempo real mínimo.

Unidad funcional	Descripción
POSIX_SINGLE_PROCESS	Funciones: `sysconf()`, `time()`, `uname()`
POSIX_SIGNALS	Gestión de señales.
POSIX_DEVICE_IO	Funciones básicas de acceso a dispositivos: `open()`, `close()`, `read()`, `write()`.
POSIX_C_LANG_SUPPORT	Librería estándar C. Entre otras, contiene funciones para: gestión de caracteres, operaciones matemáticas, saltos no locales, gestión de memoria dinámica, manejo de cadenas de caracteres y lectura de la fecha y la hora.
POSIX_FILE_LOCKING	Bloqueo de ficheros.
POSIX_C_LANG_SUPPORT_R	Funciones estándar C reentrantes.

Las opciones que se encuentran soportadas en el perfil de sistema mínimo de tiempo real son las mostradas en la tabla 2.5:

Tabla 2.5: Opciones soportadas en el perfil de sistema mínimo de tiempo real.

Opción	Descripción de la funcionalidad soportada
_POSIX_NO_TRUNC	Los nombres de ficheros se truncan a la máxima longitud permitida
_POSIX_MEMLOCK	Es posible bloquear todo o parte del espacio de direcciones de un proceso para que se mantenga residente en memoria
_POSIX_MEMLOCK_RANGE	Es posible bloquear regiones del espacio de direcciones de un proceso para que se mantengan residentes en memoria
_POSIX_SEMAPHORES	Semáforos
_POSIX_SHARED_MEMORY_OBJECTS	Objetos de memoria compartida
_POSIX_REALTIME_SIGNALS	Señales de tiempo real
_POSIX_TIMERS	Relojes y temporizadores
_POSIX_MESSAGE_PASSING	Colas de mensajes
_POSIX_SYNCHRONIZED_IO	E/S sincronizada
_POSIX_THREADS	Threads
_POSIX_TRHEAD_ATTR_STACKSIZE	Es posible configurar el tamaño de la pila de cada thread
_POSIX_TRHEAD_ATTR_STACKADDR	Es posible indicar la dirección de comienzo de la pila de cada threads

Tabla 2.5: Opciones soportadas en el perfil de sistema mínimo de tiempo real. (cont.)

Opción	Descripción de la funcionalidad soportada
`_POSIX_TRHEAD_PRIO_INHERIT`	Mutexes con protocolo de herencia básica de prioridad
`_POSIX_TRHEAD_PRIO_PROTECT`	Mutexes con protocolo de protección de prioridad
`_POSIX_TRHEAD_PRIORITY_SCHEDULING`	Políticas de planificación de threads

Tal como hemos comentado, debido a la reciente aprobación de las extesiones adicionales de tiempo real, estándares POSIX.1d (1999) y POSIX.1j (2000) y la fusión con el estándar X/Open, se ha puesto en marcha el proceso de revisión de los perfiles de tiempo real para que incluyan algunos de los servicios definidos en esos nuevos estándares. En el último borrador generado en dicho proceso de revisión [PSX02], se plantea añadir al perfil mínimo los siguientes servicios:

- *Reloj monótono*. La hora indicada por este reloj crece monótonamente con el transcurso del tiempo, no pudiendo ser cambiada por la aplicación. Es importante para las aplicaciones de tiempo real puesto que su utilización asegura que el cumplimiento de sus requerimientos temporales no se verá afectado por cambios de hora.
- *Operación de suspensión temporizada absoluta de alta resolución*. Permite la suspensión de tareas hasta la llegada de un instante de tiempo. Su utilización evita el indeseado efecto de deriva que se produce al usar una suspensión relativa para la activación de tareas periódicas y resulta más sencilla y eficiente que la utilización de temporizadores al no implicar utilización de señales. La utilización de esta operación es muy común en aplicaciones de tiempo real con threads periódicos.
- *Relojes y temporizadores de tiempo de CPU*. Útiles para las aplicaciones de tiempo real ya que permiten a las aplicaciones detectar las situaciones en las que un thread supera su tiempo de ejecución estimado de peor caso.
- *Política de planificación de "Servidor esporádico"*. Se considera necesaria para las aplicaciones empotradas de tiempo real ya que permite mejorar la planificación de threads aperiódicos.
- *Operaciones con tiempo límite* para la toma de mutexes y la espera en semáforos. Fundamentales en aplicaciones de tiempo real para poder ejecutar acciones correctoras en el caso de que un recurso permanezca tomado durante demasiado tiempo.

Las unidades de funcionalidad de la opción X/Open de estándar que se proponen para ser incluidas en el perfil mínimo por considerarlas interesantes para las aplicaciones empotradas de tiempo real son:

- XSI_THREAD_MUTEX_EXT: incluye opciones para detectar el uso incorrecto de los mutexes.
- XSI_THREADS_EXT: proporciona funciones para mejorar el control sobre la pila de los threads.

Además de incluir nueva funcionalidad, en la revisión en curso también se está planteando eliminar del perfil mínimo algunos servicios como:

- *Librería matemática C*. La razón esgrimida para su eliminación es que se trata de una librería muy grande que no es necesaria para muchas aplicaciones empotradas.

- *Colas de mensajes*. Se considera que no son utilizados por la mayoría de las aplicaciones empotradas pequeñas y el mantener esta funcionalidad aumenta el tamaño de los sistemas operativos. Además pueden ser fácilmente implementadas mediante mutexes y variables condicionales.

En el caso de los semáforos, el actual borrador explica que se mantienen en el perfil por razones de compatibilidad con las aplicaciones antiguas, pero recomienda utilizar en su lugar las variables condicionales y los mutexes ya que con estos últimos es posible evitar la inversión de prioridad no acotada.

2.2. El lenguaje Ada 95

El lenguaje de programación Ada 95 es un descendiente directo de Ada 83, el cual fue desarrollado por iniciativa del Departamento de Defensa de los EE.UU. (DoD) para ser utilizado en sistemas empotrados.

A principio de la década de los 70, el DoD usaba gran cantidad de software, en su mayor parte en aplicaciones empotradas, que había sido desarrollado por compañías independientes que utilizaban una enorme variedad de lenguajes de programación. El rápido progreso de la tecnología en el campo de los dispositivos electrónicos que tuvo lugar en aquella época parecía indicar que sería posible abordar aplicaciones software mucho más ambiciosas y complejas, pero la realidad era que los grandes programas no se finalizaban dentro de los plazos programados, su costo final excedía enormemente lo inicialmente presupuestado, contenían muchos errores y no satisfacían a los clientes ya que rara vez cubrían todos sus requerimientos. Este fenómeno, denominado “crisis del software”, era debido a que los lenguajes y paradigmas de programación no habían evolucionado lo suficiente y no resultaban apropiados para las grandes aplicaciones que se pretendían abordar.

Como respuesta a esta crisis, el Departamento de Defensa estudió las características que debía reunir un lenguaje de programación, principalmente enfocado a los sistemas empotrados, de forma que se redujesen los costos de desarrollo y mantenimiento del software [STE78], descubriendo que ninguno de los lenguajes existentes cumplía los requisitos deseados. Por esta razón lanzó un concurso para el diseño de un nuevo lenguaje de programación que permitiera aplicar de forma eficiente los principios de la ingeniería software.

El proceso culminó en 1983 con la generación del estándar ANSI para el lenguaje de programación Ada [ADA83]. El nombre fue elegido en honor a Ada Augusta, hija del poeta ingles del siglo XIX Lord Byron. Ada fue la programadora de la “máquina analítica” del científico Charles Babbage (en cierto sentido la primera calculadora programable), por lo que puede ser considerada como el primer programador de la historia. Este lenguaje, también denominado Ada 83, aunque específicamente desarrollado para los sistemas empotrados, fue extensamente utilizado como lenguaje de propósito general ya que englobaba los conocimientos de ingeniería software de la época.

En el año 1988, y de nuevo por iniciativa del Departamento de Defensa de los EE.UU., se puso en marcha el proyecto Ada 9X, con el que se pretendía revisar el lenguaje Ada 83 para adaptarle a los nuevos paradigmas de programación, principalmente a la “programación orientada a objetos”. La revisión fue provocada en gran medida por el hecho de que el lenguaje, inicialmente pensado para sistemas empotrados, estaba siendo utilizado en muchas otras áreas de aplicación. El proceso de revisión finalizó en 1995 con un estándar ISO que describe el manual de referencia del lenguaje Ada 95, a partir de ahora también denominado simplemente

Ada en esta memoria. Recientemente fueron publicadas las correcciones técnicas del estándar anteriormente mencionado constituyendo el “Manual de Referencia Ada Consolidado” [ADA00].

Ada soporta los aspectos más avanzados de la ingeniería software, rama de la ingeniería que persigue la reducción de los costos de desarrollo y mantenimiento de las aplicaciones mediante el aumento de la fiabilidad del código, la reusabilidad de componentes y el establecimiento de métodos de trabajo disciplinados.

Los módulos o componentes software son implementados en Ada mediante el concepto de paquete. Los paquetes Ada permiten un alto nivel de abstracción y ocultamiento de la información (la claves para la escritura de software reusable) imponiendo una clara distinción entre las partes visibles para el resto de la aplicación, descritas en la interfaz del paquete, y las dependientes de la implementación que se hayan contenidas en su cuerpo. La existencia del concepto de paquete genérico, el cual permite independizar los algoritmos de los tipos de datos sobre los que operan, facilita aún más el proceso de construcción de código reutilizable.

La abstracción de datos se obtiene mediante la utilización conjunta de los paquetes y de la característica del lenguaje que permite al programador definir sus propios tipos de datos junto con los operadores asociados. Los nuevos tipos así creados serán tratados por el compilador igual que se tratara a los tipos predefinidos por el lenguaje.

Los paquetes constituyen unidades de compilación independiente. Esto facilita el desarrollo de grandes aplicaciones, puesto que los módulos pueden ser compilados y probados de forma independiente por los distintos equipos de trabajo sin necesidad de disponer aún de la aplicación final. Además la estructura del lenguaje permite conocer en tiempo de compilación las dependencias entre los distintos módulos sin que sea necesaria la intervención del programador como ocurre en otros lenguajes (como el C y el uso de los “makefiles”). Esto permite automatizar el proceso de compilación de grandes aplicaciones, con lo que se puede optimizar el número de recompilaciones a realizar tras una modificación y se evitan los errores que la intervención del programador puede provocar.

Una de las características más relevantes del lenguaje Ada es la fiabilidad del código generado, conseguida gracias a que las reglas del lenguaje permiten detectar numerosos errores en tiempo de compilación. Así, entre otras situaciones, se producirá un error de compilación cuando no se respete el modo (entrada, salida o entrada-salida) de un parámetro de un procedimiento o cuando no se hayan considerado todas las alternativas en una sentencia condicional múltiple (`case`). Pero la mayor parte de los errores detectados durante la fase de compilación lo son gracias a la propiedad de tipología estricta del lenguaje. Dicha propiedad obliga al programador a hacer explícita cualquier conversión de tipos que desee realizar, suponiendo un error de compilación la utilización de un dato de tipo indebido en una expresión, como parámetro de un procedimiento o función o en cualquier otra construcción del lenguaje.

A pesar de las características del lenguaje y del cuidado diseño de la aplicación, es inevitable que en ocasiones se produzcan errores durante la ejecución del programa. Por ejemplo que como resultado de una operación aritmética a un dato le sea asignado un valor fuera de rango, que se pretenda realizar una división por cero o que se acceda a un “array” con un valor incorrecto del índice. En estas circunstancias, lo normal en la mayoría de los lenguajes de programación es que la aplicación finalice con un mensaje más o menos explícito, o incluso que el error no sea ni siquiera detectado. Por el contrario en Ada el error sería detectado y notificado a la aplicación mediante la generación de una excepción, la cual puede ser tratada de forma que se ejecuten las acciones correctoras pertinentes. Gracias a esta propiedad del lenguaje Ada, gran parte de los

errores no detectados durante la fase de compilación, pueden ser detectados en tiempo de ejecución antes de que produzcan resultados catastróficos o impredecibles.

El uso de excepciones no está limitado al tipo de errores anteriormente citado, sino que las aplicaciones pueden definir sus propias excepciones y generarlas (o elevarlas) explícitamente cuando detecten una situación de error. Las excepciones en Ada son el sistema preferido para la notificación de errores ya que constituyen un mecanismo muy potente que evita la necesidad de incluir numerosos chequeos en el código, además de permitir la separación del tratamiento de los errores del punto donde se producen, con la consiguiente mejora en la legibilidad del código. El mecanismo de excepciones ha demostrado ampliamente su validez, como lo prueba el hecho de que otros lenguajes posteriores al Ada, como son el Java o el C++, han adoptado también este mecanismo de propagación de errores.

Otra característica destacable la constituye el hecho de que la concurrencia esté directamente soportada por el lenguaje, existiendo primitivas que permiten definir las actividades concurrentes, denominadas tareas, así como sus interacciones. La planificación de las tareas se realiza mediante una política expulsora basada en prioridades con orden FIFO dentro de la misma prioridad, como la definida en el apartado 1.2, “Algoritmos de planificación”. Se definen dos formas de interacción entre tareas: el envío de mensajes o “rendezvous” y la utilización de objetos protegidos. Éstos últimos permiten el acceso exclusivo a datos compartidos mediante la utilización del protocolo de techo de prioridad inmediato, también denominado protección de prioridad que fue explicado en el apartado 1.3, “Protocolos de sincronización”. Tanto para la política, como para los mecanismos de sincronización elegidos se dispone de una amplia base teórica, lo que permite programar aplicaciones con tiempos de respuesta predecibles.

Una de las más importantes aportaciones del Ada 95 frente a su antecesor Ada 83 es que soporta de forma completa el paradigma de programación orientada a objetos. Además del encapsulamiento y ocultación de la información permitido por los paquetes (ya disponible en Ada 83), en Ada 95 se definen los registros etiquetados (tagged). Este nuevo tipo de dato permite aplicar los conceptos de herencia y polimorfismo, que son básicos en la programación orientada a objetos.

Otros lenguajes de programación muy utilizados en la actualidad no proporcionan muchas de las características que hemos citado para el Ada [WHE97]. Así, el mecanismo para programación modular proporcionado por el lenguaje C (compilación separada y ficheros de cabecera) [C99], no permite el encapsulamiento de información con partes privadas, ni el nivel de chequeo de integridad facilitado por los paquetes Ada. Este lenguaje tampoco proporciona mecanismos equivalentes a las excepciones ni a los paquetes genéricos, ni facilita ningún tipo de soporte para la programación orientada al objeto. Pero posiblemente, la mayor desventaja del lenguaje C sea que no potencia ni impulsa la fiabilidad y mantenibilidad del código generado, entre otras razones debido a que la detección de errores en tiempo de compilación que realiza es muy limitada.

El lenguaje C++ [C++98] supera al C en muchos aspectos, como son el soporte para excepciones, programación genérica y programación orientada al objeto. Sin embargo, continúa utilizando el limitado sistema de tipos de su antecesor y su detección de errores en tiempo de compilación, aunque mejor que la del C, sigue siendo muy inferior a la proporcionada por el lenguaje Ada. Otra desventaja tanto del C como del C++ es que no incorporan la concurrencia como una característica del lenguaje, sino que ésta debe lograrse mediante llamadas al sistema operativo.

Java [GOS00] supera muchas de las limitaciones del C y C++, puesto que soporta la multitarea y las excepciones y trata de detectar más errores en tiempo de compilación. Sin embargo, sigue presentando importantes limitaciones frente a Ada, como por ejemplo la no existencia de genéricos, ni de tipos enumerados, ni de soporte para la programación de sistemas de tiempo real (al menos en su actual versión). Otra importante carencia derivada de su filosofía de ejecución independiente de la plataforma, es su falta de primitivas para acceso directo al hardware. Esta limitación cobra especial importancia a la hora de considerar este lenguaje para la escritura de un sistema operativo, lo que como ya expusimos, constituye uno de los objetivos de esta tesis.

En definitiva, Ada 95 es un lenguaje moderno que soporta los paradigmas de programación más avanzados. Sus características potencian y facilitan la escritura de código modular fácilmente legible y mantenible, permitiendo reducir los costes de desarrollo de las aplicaciones [ZEI95]. Su tipado estricto y otras propiedades del lenguaje posibilitan la detección de muchos errores en tiempo de compilación lo que aumenta enormemente la fiabilidad del código generado. Ada 95 es un lenguaje apropiado para ser utilizado en muchos tipos de aplicaciones, aunque en la actualidad su más amplia utilización tiene lugar en aplicaciones empotradas de tiempo real con requisitos de alta seguridad, especialmente en el campo aeroespacial.

2.3. Soluciones para la planificación a nivel de aplicación

Hoy en día los algoritmos de planificación más extensamente utilizados en los sistemas de tiempo real son los basados en prioridades fijas, así lo reconoce el hecho de que de esta clase sean los tres definidos en el estándar POSIX y el único existente en el lenguaje de programación Ada. Estas políticas constituyen una excelente elección para gran parte de las aplicaciones de tiempo real, puesto que constituyen una buena combinación de simplicidad y eficiencia, disponiéndose además de una amplia y consolidada base teórica que permite analizar la planificabilidad de los sistemas que utilizan este tipo de algoritmos.

Sin embargo, para determinados tipos de aplicaciones las políticas basadas en prioridades fijas pueden no ser las ideales o incluso resultar inapropiadas. Así por ejemplo, es bien sabido que las políticas de planificación basadas en prioridades dinámicas (EDF, LLF, “Best-Effort”, etc.), aunque más complejas que las basadas en prioridades fijas, presentan la ventaja frente a estas últimas de que con ellas es posible alcanzar un mayor nivel de utilización de los recursos del sistema, lo que puede hacer deseable su utilización en ciertas aplicaciones.

También existen limitaciones a la hora de abordar la planificación de tareas aperiódicas en los sistemas operativos y lenguajes de programación que no implementan ninguna política de planificación de servidores aperiódicos. Este es el caso del lenguaje Ada, aunque no del POSIX que soporta la política de “Servidor esporádico”.

Tampoco resulta posible con los algoritmos definidos en el POSIX y el Ada realizar una planificación óptima en aplicaciones en las que exista una mezcla de tareas con distintos requisitos temporales, esto es, tareas que tienen requisitos temporales estrictos para las que la superación de un plazo supone un error fatal, junto con otras en las que la superación de un plazo únicamente supone una rebaja de las prestaciones del sistema.

Pero el ejemplo paradigmático en el que los algoritmos de planificación estándares resultan inapropiados, lo constituyen los sistemas multimedia (muestreo de voz, adquisición de imágenes, reproducción de vídeo, etc.). En este tipo de sistemas la pérdida de un plazo supondría un decremento en la calidad de servicio (QoS), pero no un error fatal. Es más,

dependiendo de la calidad de servicio solicitada, las tareas podrían aumentar o disminuir sus periodos de ejecución para amoldarse a los requerimientos de CPU de otras actividades concurrentes.

Para tratar de satisfacer los requerimientos de un mayor rango de aplicaciones, algunos sistemas operativos incluyen otros algoritmos de planificación además de los definidos en el POSIX, así por ejemplo MiThOS, un sistema operativo POSIX, incluye además un planificador EDF y, por su parte, LynxOS añade a las políticas estándar POSIX una política no expulsora y amplía la política "round robin" permitiendo especificar la rodaja temporal para cada nivel de prioridad. Sin embargo, resulta evidente que la estandarización e implementación de todos los algoritmos de planificación existentes constituiría una labor inabordable. Tampoco parece que sea posible encontrar un pequeño conjunto de políticas con las que se puedan satisfacer los requerimientos de todo tipo de aplicaciones. En todo caso, y aunque esto último fuera factible, el conjunto elegido resultaría pronto insuficiente según fueran apareciendo nuevos ámbitos de aplicación seguramente con nuevos requisitos de planificación.

Una alternativa a la estandarización de nuevas políticas de planificación consiste en la implementación de estas políticas a nivel de aplicación, utilizando para ello únicamente los mecanismos actualmente facilitados por los sistemas operativos y los lenguajes de programación. Este método puede permitir la implementación de algunas políticas complejas como por ejemplo la política de servidor esporádico [GON91] [GON97] o la planificación conjunta de tareas con requisitos de tiempo real estrictos y no estrictos [ESP98] [BER01]. Sin embargo, como se expondrá a continuación, presenta limitaciones de muy difícil solución.

Cuando una política de planificación está implementada dentro del propio sistema operativo, y puesto que éste conoce en todo momento el estado de sus tareas, siempre es posible tomar las decisiones de planificación en los momentos apropiados. Por el contrario, en el caso de una implementación a nivel de aplicación como las nombradas anteriormente, es necesario que sean las tareas las que incluyan explícitamente en su código llamadas al algoritmo de planificación, de forma que sea posible su correcta sincronización con las demás tareas de la misma política. La inclusión de dichas llamadas puede afectar seriamente a la eficiencia de la implementación a la vez que empeora la legibilidad del código. Además su utilización es proclive a errores, puesto que el olvido o la colocación incorrecta de una de estas llamadas provocará un comportamiento erróneo de difícil detección. Por otra parte, existen situaciones relevantes para el algoritmo de planificación sobre las que difícilmente podrá ser informado, como por ejemplo que una tarea se bloquee al tratar de tomar un recurso compartido o que se suspenda temporalmente utilizando una operación facilitada por el sistema operativo.

Debido a todos estos problemas y limitaciones, están apareciendo sistemas operativos que tratan de proporcionar al programador una interfaz que le permita definir sus propios algoritmos de planificación de una forma más potente y sencilla. Una solución en esa línea es la propuesta por RED-Linux (Real-time and Embedded Linux) [WAN99] usando un planificador de dos niveles. El planificador de alto nivel es normalmente implementado como un proceso de usuario, mientras que el de bajo nivel se encuentra integrado en el núcleo del sistema operativo. Este último tiene la misión de escoger la próxima tarea a ejecutar basándose en sus atributos de bajo nivel: prioridad, tiempo de inicio, tiempo de finalización y máxima capacidad de ejecución. De esta forma, la política de planificación a implementar sólo es tenida en cuenta por el planificador de alto nivel. El conocimiento de dicha política le permite mapear los parámetros de calidad de servicio de las tareas en el conjunto de atributos manejado por el planificador de bajo nivel, pudiendo además indicar la importancia relativa de dichos atributos. Este mecanismo permite implementar algunas políticas de planificación, pero hay muchas otras que no pueden serlo

puesto que no se basan en los mencionados atributos de bajo nivel. Por otra parte, esta solución no permite definir protocolos de acceso a los recursos compartidos, lo que permitiría evitar la inversión de prioridad no acotada y efectos similares.

Una estrategia diferente es la basada en la herencia de CPU ("CPU Inheritance Scheduling") [FOR96], en la que el núcleo del sistema operativo únicamente implementa el bloqueo y activación de tareas y la donación de CPU entre ellas. Con esta solución, los planificadores son implementados mediante tareas de usuario, cuya principal misión consiste en ceder la CPU a sus tareas planificadas de acuerdo con la política que implementan.

Con la citada estrategia, el único método utilizado para evitar la inversión de prioridad consiste en una versión generalizada de la herencia de prioridad: las tareas bloqueadas en un recurso, donan el tiempo de CPU que pueda corresponderles a la tarea en posesión de dicho recurso. Con este método se evita la inversión de prioridad no acotada, pero no constituye un mecanismo general, ya que la inexistencia de una interfaz que permita tomar decisiones de planificación en los instantes de toma y liberación de recursos, dificulta o imposibilita la implementación de algunos protocolos de sincronización básicos (p.e. los basados en techo de prioridad). Esta carencia puede constituir una importante limitación para algunas aplicaciones.

Por otro lado, aunque con la herencia de CPU resulta posible implementar planificadores para sistemas multiprocesadores, no permite utilizar una única tarea para planificar varios procesadores. Algunas arquitecturas multiprocesadoras podrían requerir este modo de funcionamiento, como por ejemplo aquellas compuestas por un procesador multipropósito, en el que ejecuta la tarea planificadora, y un conjunto de procesadores digitales de señal en los que ejecutan las tareas planificadas.

Otra solución comúnmente utilizada consiste en implementar los algoritmos de planificación de usuario como módulos que son incluidos o enlazados con el núcleo del sistema operativo. Estos módulos exportan un conjunto de operaciones para que sean invocadas por el sistema operativo en cada instante en que sea preciso llevar a cabo una acción de planificación.

Esta estrategia es seguida por Vassal [CAN98], una modificación del núcleo de Windows NT que permite incorporar en tiempo de ejecución algoritmos de planificación utilizando el mecanismo ya existente para añadir controladores de dispositivos. La adaptación del núcleo ha consistido básicamente en la modificación del código encargado de la gestión de la interrupción software de planificación. En Vassal este código se modifica de forma que si el planificador estándar no tiene ninguna tarea en disposición de ejecutar procede a buscarla invocando de forma secuencial a los algoritmos de usuario hasta que alguno de ellos se la proporcione. Vassal no proporciona ningún mecanismo que permita definir protocolos de acceso a los recursos.

Otro sistema operativo que utiliza el mecanismo de módulos es S.Ha.R.K (Soft and Hard Real-time Kernel) [GAI01]. Es una solución más completa que la anterior que engloba la gestión de los recursos compartidos. En este sistema operativo los algoritmos de planificación de usuario son módulos que se enlazan dinámicamente con el núcleo, el cual únicamente implementa los mecanismos que permiten la invocación de las funciones exportadas por los citados módulos. Un algoritmo de planificación exporta funciones que serán invocadas por el núcleo para requerirle la tarea más prioritaria de su cola de tareas ejecutables o para informarle que una de sus tareas ha sido creada, es puesta en ejecución o expulsada, ha finalizado su trabajo actual o ha terminado. También es informado el algoritmo de planificación de usuario de cuando una tarea se ha bloqueado o ha tomado un objeto de sincronización.

Además, S.Ha.R.K proporciona un mecanismo genérico que permite evitar la inversión de prioridad no acotada y que es independiente del algoritmo de planificación utilizado por las

tareas. Este mecanismo está basado en la herencia de CPU y por lo tanto, adolece del mismo problema de falta de generalidad ya expuesto para esa solución.

También RT-Linux permite la incorporación al sistema de nuevos algoritmos de planificación utilizando el mecanismo genérico de módulos proporcionado por el sistema operativo Linux [BAR97B]. Sin embargo, ni el citado mecanismo ni el propio núcleo del sistema proporcionan primitivas que faciliten la inclusión de nuevos planificadores. Esto obliga al programador del algoritmo a tener un profundo conocimiento del sistema, ya que deberá utilizar directamente sus estructuras internas.

La implementación de los algoritmos de planificación como módulos a integrar dentro el núcleo del sistema operativo constituye, por lo general, un método muy eficiente y potente puesto que permite la invocación inmediata del algoritmo de planificación en los momentos oportunos. Sin embargo, presenta la desventaja de no permitir el aislamiento de los distintos algoritmos entre sí ni con el núcleo del sistema operativo, con lo que un error en uno de ellos puede afectar a todo el sistema. Otra desventaja de esta solución radica en las diferencias existentes entre los mecanismos de módulos o enlazado dinámico implementadas por los distintos sistemas operativos, lo que dificultaría enormemente la estandarización de este mecanismo y por tanto la portabilidad de los algoritmos de planificación así implementados, lo mismo que sucede hoy en día con los gestores de dispositivos de entrada/salida o “drivers”.

La necesidad de que las aplicaciones puedan elegir entre un mayor abanico de algoritmos de planificación también es reconocida por el “Real-Time CORBA 2.0” [RTC01]. Este estándar proporciona una interfaz orientada a objetos para algoritmos definidos por la aplicación, pero no intenta definir cómo serán las primitivas del sistema operativo que den soporte a dicha interfaz.

Según lo expuesto hasta ahora, la definición de algoritmos de planificación por la aplicación continúa siendo un tema abierto para el que no existe una solución generalmente aceptada. Además, es importante resaltar que todos los mecanismos presentados constituyen soluciones particulares que en ningún caso persiguen la compatibilidad mediante su integración en algún estándar, en particular en el estándar POSIX, lo que constituye otro tema sin resolver dentro de este campo.

3. Sistema Operativo MaRTE OS

3.1. Introducción

En la exposición de los objetivos de este proyecto se citó como el más importante el desarrollo de una interfaz que permitiera a las aplicaciones definir sus propios algoritmos de planificación y protocolos de sincronización, la cual se enmarcaría dentro de la familia de estándares POSIX. También se pretendía evaluar el perfil del "Sistema de Tiempo Real Mínimo" definido en el POSIX.13 y la complejidad que supondría incorporar en dicho perfil servicios POSIX recientemente aprobados y que se consideran interesantes para los sistemas empotrados de tiempo real.

Para el cumplimiento de los objetivos mencionados resulta fundamental disponer de un sistema operativo conforme con el perfil mínimo y cuyo código fuente se encuentre disponible. Su estructura interna debe ser modular y conocida de forma que sea sencillo incorporar nuevos servicios para proceder a su prueba y evaluación. Además de la modularidad, otra característica fundamental en un sistema operativo es la fiabilidad, por lo que el lenguaje en el que se encuentre escrito deberá, además de soportar la programación modular, potenciar la fiabilidad del código generado. Tal y como expusimos en el apartado 2.2, el lenguaje Ada 95 cumple los requisitos mencionados por lo que será el elegido para la implementación del núcleo. En el citado apartado también mencionábamos como el Ada superaba al C en los aspectos aludidos, sin embargo, este último lenguaje es mucho más utilizado en la programación de sistemas operativos de tiempo real, debido probablemente a razones históricas y comerciales. En este sentido MaRTE OS constituye una experiencia pionera, al ser uno de los primeros sistemas operativos escritos en Ada que permite la ejecución de aplicaciones concurrentes escritas en C, Ada, o una mezcla de ambos.

Como se expuso en la introducción de esta memoria, son muchos los sistemas operativos de tiempo real que proporcionan una interfaz POSIX, pero la mayoría son sistemas propietarios para los cuales no está disponible su código fuente. Esta carencia les hace totalmente inadecuados para los objetivos anteriormente citados. Ciertamente existen sistemas operativos de tiempo real con interfaz POSIX cuyo código se encuentra disponible, como RTEMS [RTE96], RT-Linux [RTLIN] y S.Ha.R.K [GAI01], pero el problema en todos ellos es que su diseño interno no sigue el modelo de threads descrito en el estándar lo que, además de constituir una fuente de ineficiencia, complica la realización de modificaciones basadas en POSIX. Ambos problemas no existirían en un sistema operativo cuyo diseño hubiera seguido el modelo POSIX desde el principio.

Un sistema operativo que podría parecer más apropiado para nuestros propósitos es MiThOS [MUE95], el cual está basado en una librería de threads POSIX extensamente utilizada denominada "FSU PThreads library" [MUE93], y constituyó la primera implementación del perfil de "Sistema de Tiempo Real Mínimo". De cara a los objetivos perseguidos en esta memoria, MiThOS presenta dos importantes inconvenientes: está implementado para una arquitectura un tanto específica (la SPARC VME) y está escrito en C, lo que, unido a la

compleja estructura de la “FSU PThreads library”, dificulta en gran medida su modificación para la incorporación de nuevos servicios.

Por todas las razones expuestas se decidió diseñar e implementar (utilizando el lenguaje Ada 95) un sistema operativo de tiempo real que fuera conforme con el subconjunto mínimo definido en el estándar POSIX.13. Dicho sistema operativo, denominado MaRTE OS (Minimal Real-Time Operating System for Embedded Applications), puede ser utilizado para desarrollar aplicaciones empotradas de tiempo real o como herramienta para la docencia en sistemas operativos de tiempo real y sistemas empotrados. Con el fin de facilitar su utilización para estos y otros tipos de proyectos por otras universidades, empresas y organizaciones, MaRTE OS se distribuye como software libre bajo licencia GPL (“GNU General Public License”) [GNU].

Además de los usos anteriormente citados, y haciendo referencia a lo más directamente relacionado con este trabajo de investigación, el disponer de un sistema operativo del que se conoce perfectamente su estructura interna y que ha sido diseñado siguiendo el modelo de threads POSIX, facilita en gran medida la incorporación y prueba de servicios basados o al menos cercanos a los definidos por el estándar POSIX. Por tanto MaRTE OS constituye la herramienta perfecta para:

- Evaluar servicios POSIX recién aprobadas o en proceso de aprobación y que aún no han sido implementadas en los sistemas operativos existentes.
- Analizar la complejidad que supondría la inclusión de servicios POSIX recién aprobados dentro el subconjunto mínimo de sistema de tiempo real, de cara a la revisión en curso del estándar POSIX.13 [PSX02].
- Implementar y evaluar nuevas interfaces a enmarcar dentro de la familia de estándares POSIX, y en particular las expuestas en los capítulos 4, “Interfaz de usuario para la definición de algoritmos de planificación” y 5, “Interfaz de usuario para la definición de protocolos de sincronización”, que constituyen una parte fundamental del trabajo de investigación presentado en esta memoria.

Un sistema operativo como MaRTE OS no consiste únicamente en un núcleo en el que se implementa la funcionalidad incluida en el subconjunto mínimo del POSIX (concurrencia, sincronización, temporización, etc.), sino que además consta de una serie de aplicaciones y servicios que posibilitan la creación, carga y depuración de las aplicaciones. Ha sido necesario, por tanto, crear un entorno de desarrollo cruzado basado en Linux y en los compiladores de GNU GCC y GNAT.

3.2. Principales características

Como ya se ha comentado MaRTE OS es un sistema operativo para aplicaciones empotradas cuya principal característica es que sigue el subconjunto mínimo de sistema de tiempo real definido en el estándar POSIX.13. Otras características importantes de MaRTE OS se enumeran y justifican en los siguientes apartados.

Pensado para aplicaciones principalmente estáticas

En los sistemas empotrados pequeños, lo normal es que el computador se encuentre dedicado a realizar una o un pequeño conjunto de funciones claramente definidas, para las que el máximo número de threads y demás recursos del sistema que serán utilizados en tiempo de ejecución es conocido de antemano. Esta limitación es una consecuencia del requerimiento de predecibilidad

temporal establecido para tales sistemas, el cual difícilmente podría lograrse en una aplicación que pudiera hacer uso de forma simultánea de un número ilimitado o impredecible de recursos.

MaRTE OS, en cuanto que destinado a sistemas de las características anteriormente mencionadas, sólo permite a las aplicaciones utilizar un número limitado de recursos. En la configuración del sistema es posible especificar el número máximo de recursos de cada tipo que serán prealojados durante la inicialización del sistema. De esta forma es posible ajustar el tamaño del ejecutable en función de las necesidades de cada aplicación. Así, entre otros parámetros, es posible especificar:

- el máximo número de threads que pueden existir al mismo tiempo
- el número de niveles de prioridad diferentes
- la longitud máxima de la cola de eventos temporales
- el número máximo de temporizadores
- el número máximo de señales pendientes
- el número de datos específicos de cada thread
- el tamaño de la pila (“stack”) de los threads
- el número máximo de threads con política de servidor esporádico
- el tamaño del área de memoria dinámica

La determinación durante la configuración del sistema del número de recursos a utilizar permite simplificar el diseño del núcleo así como minimizar su tamaño para cada aplicación particular. Por otra parte, aunque la creación de recursos en tiempo de ejecución no tiene por qué provocar indeterminación temporal (en el caso de que se utilice un algoritmo de gestión memoria dinámica de tiempo real), el que todos los recursos sean prealojados e inicializados durante la fase de inicialización del sistema permite ahorrar mucho tiempo cuando, ya en tiempo de ejecución, la aplicación requiera su creación.

Presenta tiempos de respuesta acotados en todos sus servicios

El núcleo está pensado para ser utilizado en aplicaciones de tiempo real, en las que el cumplimiento de los plazos temporales constituye un requisito fundamental para el correcto funcionamiento del sistema. Con este objeto, los tiempos de ejecución de las funciones del núcleo son acotados y pequeños. También son acotados y lo más cortos posible los periodos en los que el núcleo mantiene deshabilitadas las interrupciones, con el fin de permitir una respuesta rápida a los eventos externos y de planificación, como por ejemplo el disparo de los temporizadores.

Existe un sólo espacio de direcciones de memoria compartido por el núcleo y la aplicación

El que no existan barreras de protección entre el núcleo y las aplicaciones supone una merma en la seguridad del sistema pero simplifica en gran medida su implementación y mejora su eficiencia, puesto que de esta forma las llamadas al sistema son mucho más rápidas. En todo caso, la potencial fuente de inseguridad en el sistema que esta solución supone se ve mitigada por el hecho de que las aplicaciones a las que está destinado harán escaso uso de la memoria dinámica. El riesgo de que la aplicación dañe accidentalmente el núcleo se reduce aún más en el caso de aplicaciones escritas en Ada 95, gracias a las características de dicho lenguaje de programación, que fueron expuestas en el apartado 2.2, “El lenguaje Ada 95”.

Además, en los sistemas empotrados pequeños (a los que MaRTE OS va destinado) lo normal es que el computador se encuentre dedicado a realizar una única función (o varias funciones

estrechamente relacionadas). Por lo tanto, en ese tipo de sistemas un fallo en el sistema operativo o en la aplicación serían igualmente graves, puesto que en ambos casos el sistema empotrado dejaría de realizar la labor a la que estaba destinado. En consecuencia, no tiene sentido proteger al núcleo frente a fallos en la aplicación puesto que en todo caso la operación del sistema no podrá continuar.

Permite ejecutar aplicaciones Ada y C

El núcleo presenta las interfaces POSIX Ada y C, siendo posible realizar el desarrollo cruzado de aplicaciones Ada y C mediante el empleo de herramientas basadas en los compiladores de GNU: GNAT y GCC. Además, en el caso de las aplicaciones Ada, la librería de tiempo de ejecución del compilador GNAT (GNARL [GIE94]) ha sido adaptada para ejecutar sobre MaRTE OS. También es posible ejecutar aplicaciones en las que conviven de una forma coherente tareas Ada junto con threads C.

Incluye gestión de memoria dinámica simplificada

No es habitual que las aplicaciones empotradas hagan una utilización intensiva de la memoria dinámica, lo normal es que no la utilicen en absoluto o que lo hagan únicamente durante la inicialización de la aplicación para crear objetos que serán utilizados durante todo su tiempo de ejecución. Se ha optado por una implementación muy sencilla y eficiente, que permite la reserva de bloques de memoria consecutivos hasta agotar el área dedicada a memoria dinámica. Con esta sencilla implementación un bloque de memoria liberado no puede volver a ser utilizado.

También existen algoritmos con una complejidad algo mayor que la solución anteriormente expuesta, pero que permiten la reutilización de la memoria liberada, a la vez que siguen presentando tiempos de respuesta acotados. Un ejemplo es el algoritmo “Buddy” [RUS99], el cual podría ser implementado en nuestro núcleo en un futuro.

Portable a distintas arquitecturas

El núcleo presenta una interfaz abstracta con el hardware que permite independizar su funcionamiento del hardware específico sobre el que se ejecuta. Dicha interfaz proporciona al núcleo operaciones que permiten acceder al reloj y al temporizador del sistema, instalar manejadores de señal y realizar el cambio de contexto entre tareas.

La facilidad de portado del núcleo a sido comprobada con la realización de un proyecto fin de carrera consistente en el portado de MaRTE OS a un microcontrolador MC68332.

Núcleo monolítico

Las estructuras de datos del núcleo deben ser protegidas ante la posibilidad de acceso simultáneo por parte de varias tareas ejecutando llamadas al sistema. Para lograr esa protección, MaRTE OS se ha diseñado como un núcleo de estructura monolítica, en el que las llamadas al sistema, o al menos la parte de ellas en la que se accede a estructuras globales, constituyen secciones críticas en las que las interrupciones se encuentran deshabilitadas. La mayoría de los sistemas operativos, tanto de propósito general como de tiempo real, están basados en esta arquitectura.

Una posible alternativa a la arquitectura de monolítica es la denominada “microkernel”. Los sistemas operativos diseñados de acuerdo con esta arquitectura están formados por un núcleo muy pequeño en el que se implementan los servicios básicos y un conjunto de procesos cooperativos encargados de proporcionar a las aplicaciones servicios de más alto nivel [LIE95].

Sin embargo, no tiene sentido utilizar esta alternativa con un sistema operativo tan reducido como MaRTE OS, ya que la funcionalidad en él implementada es justamente la que debería incluirse en el "microkernel". Así por ejemplo QNX Neutrino [QNXa], el sistema operativo de tiempo real con estructura "microkernel" más utilizado, implementa en su núcleo la gestión de threads, señales, relojes, temporizadores, interrupciones, semáforos, mutexes, variables condicionales y barreras, además del mecanismo de comunicación entre procesos. Como puede observarse, esta funcionalidad es básicamente la soportada por MaRTE OS.

Toma la forma de una librería para ser enlazada con la aplicación

En el computador de desarrollo la aplicación de usuario es compilada y posteriormente enlazada con las librerías que forman el núcleo de MaRTE OS. El ejecutable así generado constituye un programa independiente listo para ser cargado en el computador empotrado y comenzar la ejecución de la aplicación de usuario de forma automática. Puesto que MaRTE OS está pensado para sistemas empotrados pequeños en los que lo normal es que no exista terminal para interfaz con el usuario, no tiene sentido el disponer de ningún tipo de interprete de órdenes, cuya inclusión aumentaría el tamaño del ejecutable.

3.3. Funcionalidad soportada

Como ya expusimos anteriormente, MaRTE OS sigue el subconjunto mínimo definido en el POSIX.13, debiendo por tanto soportar toda la funcionalidad incluida en dicho subconjunto (la cual fue presentada en el apartado 2.1.2, "Descripción del sistema de tiempo real mínimo"). Sin embargo, existen algunas diferencias entre la funcionalidad implementada en MaRTE OS y la definida para el sistema mínimo de tiempo real. Muchas de estas diferencias corresponden a modificaciones del perfil mínimo propuestas en el proceso de revisión del estándar POSIX.13 (ya comentadas en el citado apartado 2.1.2), mientras que en otros casos se trata de servicios específicos de MaRTE OS que expondremos en el apartado 3.3.3.

3.3.1. Restricciones al sistema de tiempo real mínimo

A continuación procederemos a enumerar y justificar las restricciones al perfil mínimo realizadas en MaRTE OS:

No se han implementado las colas de mensajes

Este servicio pretende ser eliminado del perfil mínimo ya que supone un aumento del tamaño y complejidad del núcleo que no se ve compensado por la escasa utilización que de él hacen la mayoría de las aplicaciones empotradas de tiempo real. Por otra parte, en el caso de que se desee utilizar las colas de mensajes, éstas pueden ser fácilmente implementadas por la aplicación utilizando mutexes y variables condicionales.

No se permite la suspensión dentro de los manejadores de señal

Con esta restricción es posible ejecutar los manejadores utilizando una única tarea especial a la máxima prioridad del sistema, en lugar de hacerlo en el contexto de la tarea a la que la señal fue entregada. Esto supone una importante simplificación en el núcleo, ya que evita tener que extraer una tarea de la cola en la que se encuentra bloqueada (p.e. cola de tareas bloqueadas de un mutex) mientras dura la ejecución del manejador para volver a encolarla después.

Sin duda, esta simplificación implica una pérdida de funcionalidad en el uso de manejadores de señal asíncronos. Sin embargo, el estándar POSIX proporciona un mecanismo alternativo: la

utilización de threads encargados de capturar señales de forma síncrona mediante la operación `sigwait()`. Esta segunda alternativa mantiene toda la funcionalidad de los manejadores de señal tradicionales, presentando además la ventaja de que es posible, mediante el ajuste de la prioridad de la tarea manejadora, fijar la prioridad a la que se desea que sean ejecutadas las operaciones de manejo del evento.

Se podría pensar entonces, que lo mejor habría sido la total eliminación de los manejadores de señal. Sin embargo esto no es posible debido a que son utilizados por la librería de tiempo de ejecución del compilador GNAT para implementar la transferencia asíncrona de control (ATC). En este caso los manejadores utilizados no se suspenden por lo que, a pesar de la simplificación adoptada, el ATC puede ser utilizado por las tareas Ada ejecutando sobre MaRTE OS sin ningún tipo de limitación.

La importante simplificación que se consigue con esta restricción nos ha llevado a proponerla para la futura revisión del perfil POSIX.13 de sistema de tiempo real mínimo [PSX02].

No se han implementado los semáforos

Los semáforos constituyen una antigua primitiva de sincronización de bajo nivel proclive a errores y al efecto conocido como inversión de prioridad no acotada. El estándar POSIX define otro conjunto completo de primitivas de sincronización de más alto nivel: los mutexes y las variables condicionales. La inversión de prioridad puede evitarse mediante el uso de mutexes con protocolos de herencia básica de prioridad o de techo de protección inmediato.

Resulta por tanto innecesaria, e incluso inapropiada, la utilización de semáforos en las aplicaciones de tiempo real. El estándar POSIX les reserva el papel de mecanismo de sincronización con los manejadores de señal, pero puesto que en MaRTE OS se ha impuesto la restricción de que los manejadores de señal no pueden suspenderse, resultan también innecesarios para esta función.

La actual revisión del estándar POSIX.13 no elimina los semáforos del perfil mínimo, aunque les declara obsoletos, recomendando expresamente la utilización en su lugar de mutexes y variables condicionales. Se mantienen en el perfil únicamente por razones de compatibilidad con las aplicaciones antiguas.

3.3.2. Funcionalidad propuesta para su inclusión en el perfil mínimo

Como comentamos con anterioridad, en la revisión del estándar POSIX.13 se pretende añadir al perfil mínimo nuevos servicios recién incluidos en la familia de estándares POSIX. De entre ellos, hemos implementado en MaRTE OS los que consideramos más interesantes para aplicaciones empotradas de tiempo real. Con ello pretendemos evaluar la complejidad que supone su inclusión en un sistema operativo mínimo obteniendo información que pueda resultar de ayuda en el proceso de revisión recién comenzado. Los servicios seleccionados son:

Reloj monótono

La hora indicada por este reloj crece monótonamente con el transcurso del tiempo, no pudiendo ser cambiada por la aplicación. Es importante para las aplicaciones de tiempo real puesto que su utilización asegura que el cumplimiento de sus requerimientos temporales no se verá afectado por cambios de hora. Este reloj fue incluido en el estándar POSIX.1j.

Suspensión temporizada absoluta de alta resolución

Este servicio, también incluido en el POSIX.1j, permite la suspensión de tareas hasta que se alcance un instante de tiempo absoluto, frente a la operación ya existente de suspensión relativa que lo hacía durante un intervalo temporal. Su utilización evita el indeseado efecto de deriva que se produce al usar una suspensión relativa para la activación de tareas periódicas y resulta más sencilla y eficiente que la utilización de temporizadores al no implicar utilización de señales. Además, esta operación permite indicar el reloj en base al que se desea efectuar la suspensión, pudiéndose elegir entre el reloj de tiempo real o el monótono.

Relojes y temporizadores de tiempo de ejecución

Los relojes de tiempo de ejecución resultan útiles para las aplicaciones de tiempo real ya que permiten medir de forma sencilla el tiempo de ejecución de los threads. Más importantes son los temporizadores de tiempo de ejecución, puesto que permiten detectar las situaciones en que una tarea supera su tiempo de ejecución de peor caso. Esta funcionalidad es muy interesante para aplicaciones de tiempo real estricto, ya que los resultados del análisis de planificabilidad sólo son válidos si la estimación de los tiempos de ejecución de peor caso es correcta. Los temporizadores de tiempo de ejecución nos permiten validar esta información y tomar acciones correctoras en caso de que un tiempo de ejecución haya sido sobrepasado. También resultan útiles para implementar a nivel de aplicación muchas políticas de planificación, como por ejemplo las denominadas “Constant Bandwidth Server” [ABE98] y “Servidor Diferido” [STR95]. Los relojes y temporizadores de tiempo de ejecución fueron introducidos en el estándar POSIX.1d.

Política de planificación de servidor esporádico

Esta política de planificación, definida en el POSIX.1d, permite procesar actividades aperiódicas con cortos tiempos de respuesta, con la ventaja adicional de que su efecto sobre tareas de menor prioridad está acotado incluso en situaciones de sobrecarga, en las que se produce una gran cantidad de eventos aperiódicos en un intervalo de tiempo pequeño.

3.3.3. Funcionalidad extra

Además de la funcionalidad anteriormente mencionada en los subapartados 3.3.1 y 3.3.2, en MaRTE OS hemos implementado algunos servicios no existentes en la actualidad en el estándar POSIX, pero que consideramos pueden resultar de mucha utilidad para las aplicaciones empotradas de tiempo real.

Gestión de interrupciones a nivel de aplicación

MaRTE OS proporciona una interfaz que define las operaciones básicas de habilitación y deshabilitación de interrupciones, así como de instalación de procedimientos para ser ejecutados a la más alta prioridad del sistema inmediatamente después de producirse la interrupción. Pero, lo que es más importante, la interfaz también incluye una operación para bloquear a una tarea a la espera de la generación de una interrupción hardware. Esto permite especificar la prioridad a la que se desea ejecutar parte o todo el procesado de la interrupción, con la consiguiente reducción de los tiempos de bloqueo producidos sobre tareas de mayor prioridad.

Esta funcionalidad será expuesta en detalle en el apartado 3.5.

Interfaz para la definición de políticas de planificación y protocolos de sincronización

El diseño e implementación de la citada interfaz ha constituido uno de los principales objetivos de este trabajo de investigación que será abordado de forma detallada en los capítulos 4, “Interfaz de usuario para la definición de algoritmos de planificación” y 5, “Interfaz de usuario para la definición de protocolos de sincronización”.

3.4. Arquitectura

3.4.1. Visión general

El núcleo de MaRTE OS se ha escrito principalmente utilizando el lenguaje de programación Ada 95, habiéndose utilizado también los lenguajes C y ensamblador. El uso de estos dos últimos lenguajes representa un porcentaje muy pequeño del tamaño total del núcleo, y corresponde principalmente a código tomado de otros sistemas operativos o librerías de código libre, tales como Linux [LNXa] [LNXb], RT-Linux y el conjunto de librerías para desarrollo de sistemas operativos denominado OSKit [FOR97]. En lenguaje C se encuentra escrito el código encargado de la inicialización del PC, de la salida de caracteres por consola mediante la escritura directa en la memoria de vídeo, así como de la obtención de los caracteres introducidos por teclado. El ensamblador también se utiliza durante la inicialización del sistema y para operaciones de muy bajo nivel como la rutina de cambio de contexto. Se ha optado por no reescribir en Ada el código importado en C y ensamblador por tratarse de pequeños trozos de código extensamente utilizados y probados y por tanto muy fiables.

Las figuras 3.1 y 3.2 muestran el esquema de capas de la arquitectura de una aplicación escrita en Ada y en C respectivamente cuando ejecutan sobre MaRTE OS. La principal diferencia entre ambas radica en la capa utilizada para comunicar la aplicación con el resto del sistema. En el caso de las aplicaciones Ada, esta interacción se realiza en su mayor parte de forma indirecta a través de la librería de tiempo de ejecución del compilador GNAT. También puede hacerse de una forma más directa utilizando la interfaz POSIX.5. En el caso de las aplicaciones C, la interacción se realiza a través de la interfaz POSIX.1, una capa muy delgada consistente únicamente en un conjunto de ficheros de cabecera que definen directamente las funciones exportadas por el núcleo de MaRTE OS y la librería estándar C.

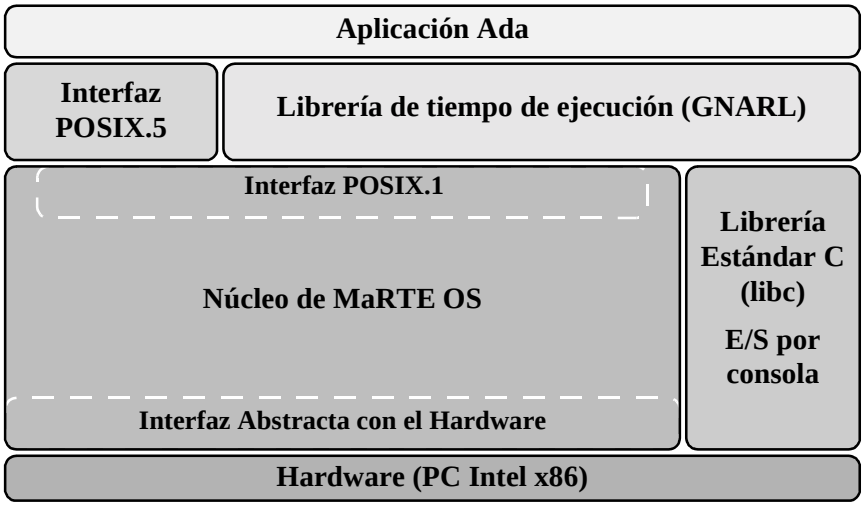

Figura 3.1: Arquitectura de una aplicación Ada ejecutando sobre MaRTE OS

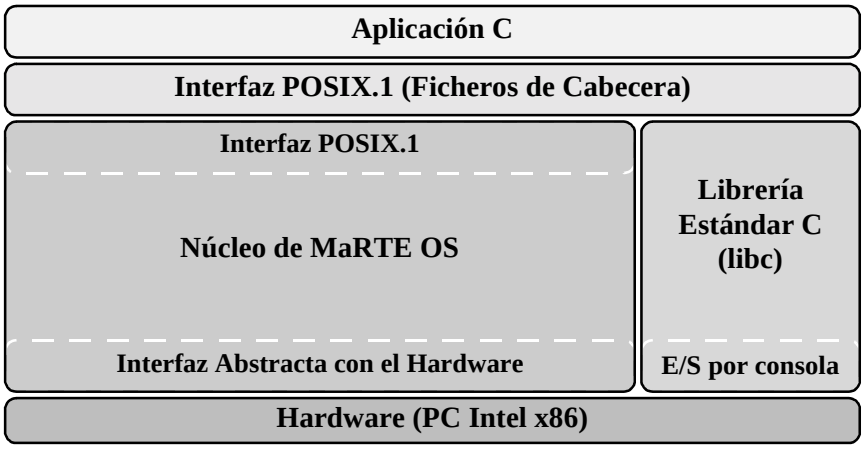

Figura 3.2: Arquitectura de una aplicación C ejecutando sobre MaRTE OS

Como puede apreciarse en ambas figuras, el núcleo incluye una interfaz abstracta de bajo nivel para acceder al hardware. En ella se define la visión que del hardware tienen las partes del núcleo que son independientes de la plataforma de ejecución. Esta interfaz constituye la única parte del núcleo que es dependiente del hardware, lo que facilita el portado de MaRTE OS a distintas plataformas. La interfaz se describe con más detalle en el apartado 3.4.2, “Interfaz abstracta con el hardware”.

Además de la dependencia de la plataforma encapsulada en la interfaz abstracta con el hardware, también la librería estándar C depende del hardware sobre el que se ejecuta en lo referente a las operaciones de entrada/salida por consola. La E/S por consola no está incluida dentro de la interfaz abstracta con el hardware porque no es común a todas las posibles arquitecturas (los sistemas empotrados normalmente no disponen de dispositivo de interfaz con el usuario). Lo normal es que esta funcionalidad sea aportada por el código encargado de manejar el dispositivo que se desea haga las veces de consola.

Como se vio en el apartado 2.1.2, el subconjunto mínimo de sistema de tiempo real definido en el estándar POSIX.13 incluye la librería estándar C (libc). La “libc” utilizada por MaRTE OS ha sido tomada del OSKit. Como muestra la figura 3.1, también es utilizada por la librería de tiempo de ejecución del compilador GNAT, pero únicamente en lo referente a las funciones básicas para entrada y salida por consola.

A pesar de estar escrito en Ada, la interfaz del núcleo ha sido desarrollada de acuerdo con la interfaz POSIX.1 (lenguaje C)[1], en lugar de seguir la interfaz POSIX.5b (lenguaje Ada). Hay varias razones detrás de esta decisión:

- La mayoría de las versiones de la librería de tiempo de ejecución GNARL ejecutan sobre un sistema operativo con interfaz POSIX.1, por lo que su adaptación resulta más sencilla y eficiente si la interfaz de MaRTE OS sigue este estándar.
- La eficiencia en las aplicaciones Ada que utilizan la librería de tiempo de ejecución, será mayor si las llamadas de bajo nivel realizadas por ésta, se hacen directamente sobre el núcleo sin necesidad de más capas de software intermedias.

1. En la versión de MaRTE OS que se encuentra públicamente disponible a fecha de escritura de esta memoria de tesis doctoral (V. 1.0), todavía hay partes del núcleo que presentan una interfaz similar a la POSIX-Ada aunque sin utilizar excepciones.

- La interfaz POSIX.5b hace un extenso uso de las excepciones, y puesto que el núcleo debe ofrecer tanto la interfaz en lenguaje Ada como la C, no es posible utilizar excepciones dentro del núcleo, ya que las aplicaciones C no sabrían como manejarlas.
- El uso de excepciones no esta permitido para algunas aplicaciones de seguridad crítica, por lo que su inclusión en el núcleo imposibilitaría su utilización en dichas aplicaciones.

Además de no utilizar excepciones, hay otras importantes características del lenguaje Ada, como son la multitarea o la programación orientada a objetos que no son utilizadas dentro del núcleo o que lo son de forma limitada. En el caso de la multitarea, es obvio que no puede ser utilizada en el núcleo puesto que, como se aprecia en la figura 3.1, a ese nivel no se dispone de la librería de tiempo de ejecución. Por su parte los tipos etiquetados son usados extensamente en el núcleo, aunque siempre evitando utilizar el enlazado dinámico por constituir una posible fuente de ineficiencia. A pesar de ello, el subconjunto de Ada utilizado presenta suficientes ventajas para justificar la elección de este lenguaje, siendo similar a los subconjuntos empleados en programas de seguridad crítica para aplicaciones aero-espaciales o médicas entre otras, como son el subconjunto SPARK [BAR97a] o el perfil de Ravenscar [BUR99].

La interfaz está formada por funciones Ada definidas de acuerdo a los prototipos propuestos en el estándar POSIX.1, para ello es necesario que los tipos utilizados por las aplicaciones C coincidan con los utilizados por el núcleo:

- Se definen tipos Ada equivalentes a los tipos C estándar (`char`, `int`, etc.).
- Los `access` de Ada mapean exactamente los punteros C (extensamente utilizados en la interfaz POSIX.1) según asegura el manual de referencia del compilador GNAT.
- También los registros Ada y las estructuras C son equivalentes como indica el manual de referencia del compilador GNAT.
- En el caso de los objetos opacos definidos por la interfaz POSIX (`pthread_attr_t`, `sigset_t`, `pthread_mutex_t`, `pthread_mutexattr_t`, `pthread_cond_t`, `pthread_condattr_t`, `pthread_key_t`, etc.) durante la compilación del núcleo de MaRTE OS se genera automáticamente un fichero de cabecera C en el que estos tipos se definen como formaciones de caracteres de la longitud apropiada. De esta forma cualquier cambio en la definición de uno de estos objetos se ve automáticamente reflejado en las cabeceras POSIX.

Las funciones de la interfaz son exportadas con el nombre propuesto por el estándar POSIX.1. De esta forma, el núcleo constituye una librería que puede enlazarse directamente con las aplicaciones C, con lo que la capa de interfaz POSIX.1 mostrada en la figura 3.2 únicamente consiste en el conjunto de ficheros de cabecera descrito por el estándar, no habiendo necesidad de ningún tipo de código intermedio. Por su parte, la interfaz POSIX-Ada, que aparece en la figura 3.1, constituye una capa adicional que convierte los códigos de error retornados por las funciones del núcleo en excepciones para ser elevadas a la aplicación Ada.

3.4.2. Interfaz abstracta con el hardware

Esta interfaz proporciona al resto del núcleo de MaRTE OS una visión abstracta de la plataforma que consta de los siguientes dispositivos y operaciones:

- *Temporizador*: dispositivo que permite ser programado para provocar una interrupción cuando transcurra el intervalo de tiempo elegido, siendo el intervalo máximo programable finito y conocido por MaRTE OS. Cuando el intervalo deseado supera el máximo, el núcleo se encargará de programar varias veces consecutivas el temporizador

hasta cubrir dicho intervalo. Tras la inicialización del sistema el temporizador no se encuentra programado.

- *Reloj monótono de alta resolución*: dispositivo que permite obtener el tiempo transcurrido desde un instante fijo en el pasado, posiblemente la hora de arranque del sistema. El valor retornado por este dispositivo crece monótonamente con el tiempo y no puede ser cambiado por la aplicación ni por el sistema operativo. Conviene que tenga una resolución igual o mayor que la del temporizador y no es necesario que exista como dispositivo físico. En el caso de que el sistema carezca de reloj, MaRTE OS realizará la programación periódica del temporizador incluso cuando no haya ningún evento temporal pendiente. De esta forma la capa de interfaz podrá llevar la cuenta del tiempo transcurrido desde la inicialización del sistema actualizando una variable global tras cada programación. El tiempo transcurrido en un instante dado se podrá obtener como la hora de la última programación más el estado de la cuenta del temporizador en ese instante.
- *Reloj de tiempo real*: se trata de un reloj al que se le requiere precisión de segundos. Permite obtener la hora actual con el formato indicado por el estándar POSIX, esto es, como el tiempo transcurrido desde las cero horas, cero minutos y cero segundos del día 1 de enero de 1970 (instante denominado “Epoch” por el estándar POSIX). Se lee una vez durante la inicialización del sistema para obtener la hora actual. En sistemas que no dispongan de este dispositivo su lectura retornará 0.
- *Operaciones de conversión de tiempos*: se proporcionan operaciones que permiten convertir tiempos medidos en cuentas del reloj a valores del tipo `Duration` y viceversa. MaRTE OS utiliza internamente el formato usado por el reloj para representar valores temporales. Se prefiere utilizar las unidades del reloj en lugar de las del temporizador puesto que las operaciones matemáticas en las que interviene el valor actual del reloj son muy frecuentes con lo que se evitan innecesarias conversiones de tipo.
- *Dispositivo controlador de las interrupciones*: permite la habilitación o deshabilitación de las interrupciones de forma individual. En el caso de que no exista un dispositivo que centralice el control de todas las interrupciones las operaciones de habilitación o deshabilitación actuarán, siempre que sea posible, directamente sobre los dispositivos generadores de interrupciones. El dispositivo comienza con todas las interrupciones deshabilitadas.
- *Interrupciones hardware*: existen una o más fuentes de interrupciones hardware identificadas en la interfaz, debiendo existir como mínimo la correspondiente al temporizador. Se facilita una operación que permite asociar un procedimiento con una interrupción de forma que sea ejecutado inmediatamente cada vez que ésta se produzca. Por omisión a todas las interrupciones les es asignado un manejador que muestra en la consola la interrupción producida y finaliza la aplicación.
- *Operaciones de gestión de los manejadores de interrupción*. Cuando las operaciones del núcleo son invocadas desde un manejador de interrupción debe posponerse la ejecución de los cambios de contexto hasta después de que haya sido completado correctamente el ciclo de reconocimiento de la interrupción, con la consiguiente rehabilitación del dispositivo controlador de interrupciones. La interfaz abstracta con el hardware proporciona una operación para saber cuando se está ejecutando dentro de un manejador y otra para indicar si se desea que sea llamada la función de planificación una vez rehabilitada la interrupción en el controlador y finalizada la ejecución del manejador.
- *Operación de cambio de contexto entre tareas*: toma como argumentos las cimas de las pilas de las tareas entre las que se realizará el cambio. Tras su ejecución el estado de los registros del procesador de la tarea saliente queda almacenado en su pila, de forma que el

valor almacenado de su contador de programa corresponde a una dirección interna de la propia rutina de cambio de contexto. Por otro lado, el estado de la tarea entrante ha sido restaurado.

- *Operaciones sobre los registros del procesador*: existen operaciones para lectura y escritura del registro de estado del procesador y para habilitar o deshabilitar todas las interrupciones. La habilitación o deshabilitación de las interrupciones se realiza mediante la modificación de uno o más bits de ese registro, por lo que estas operaciones resultan más rápidas que las proporcionadas por el dispositivo controlador de las interrupciones. Las operaciones sobre los registros son utilizadas por el núcleo para crear secciones críticas: la entrada a una de estas secciones se realiza almacenando el registro de estado y deshabilitando las interrupciones, mientras que la salida consiste en restaurar el antiguo valor del registro de estado.

Al igual que en otros sistemas operativos de tiempo real como LynxOS [LYN96], en MaRTE OS la ejecución de los manejadores de interrupción constituye una operación atómica, no permitiéndose su anidamiento. Las razones por las que se hace esto en lugar de deshabilitar únicamente la interrupción atendida son expuestas a continuación:

1. La posibilidad de deshabilitar una interrupción de forma independiente de las demás es una opción que no está disponible en todas las arquitecturas, mientras que la deshabilitación simultánea de todas ellas sí constituye una operación portable.
2. El enmascaramiento de las interrupciones de forma individual suele requerir de la existencia de un dispositivo manejador independiente del procesador. El acceso a dicho dispositivo normalmente constituye una operación lenta, que puede tardar más que la duración del código que se desea proteger. Por el contrario la deshabilitación de todas las interrupciones de forma simultánea constituye en la mayoría de las arquitecturas una operación muy rápida.
3. La principal razón para permitir el anidamiento de interrupciones sería evitar que la gestión de una interrupción de alta prioridad se viera retrasada por otra de baja. Este bloqueo es relevante en sistemas operativos en los que los manejadores deban ser largos, posiblemente debido a que no es posible desplazar parte del procesado de la interrupción fuera del manejador. En MaRTE OS la principal parte del trabajo de gestión de la interrupción puede ser realizada por una tarea ejecutando a la prioridad deseada por el programador. Dicha tarea “manejadora” se sincroniza con el procedimiento manejador utilizando las operaciones facilitadas por la interfaz para gestión de interrupciones a nivel de aplicación que será descrita en el apartado 3.5.

3.4.3. Arquitectura interna del núcleo

La figura 3.3 constituye una representación esquemática de la arquitectura interna del núcleo de MaRTE OS. Los paquetes que componen el núcleo están organizados jerárquicamente a partir del paquete raíz `Kernel`, en el que se definen los tipos, constantes y operaciones básicas que será utilizados por el resto del núcleo.

Los distintos servicios POSIX son implementados por módulos formados por paquetes de segundo nivel tales como: `Kernel.Signals`, `Kernel.Hardware_Interrupts`, `Kernel.Mutexes`, `Kernel.Condition_Variables`, `Kernel.Tasks_Operations` y `Kernel.Timers`, junto con los demás paquetes que componen la subjerarquia de cada uno de estos paquetes principales. En cada módulo el paquete principal define los tipos de la interfaz POSIX, junto con la parte de chequeo de errores y de estructura global de las operaciones. Por su parte los paquetes de tercer nivel pueden bien desempeñar el mismo papel que el principal

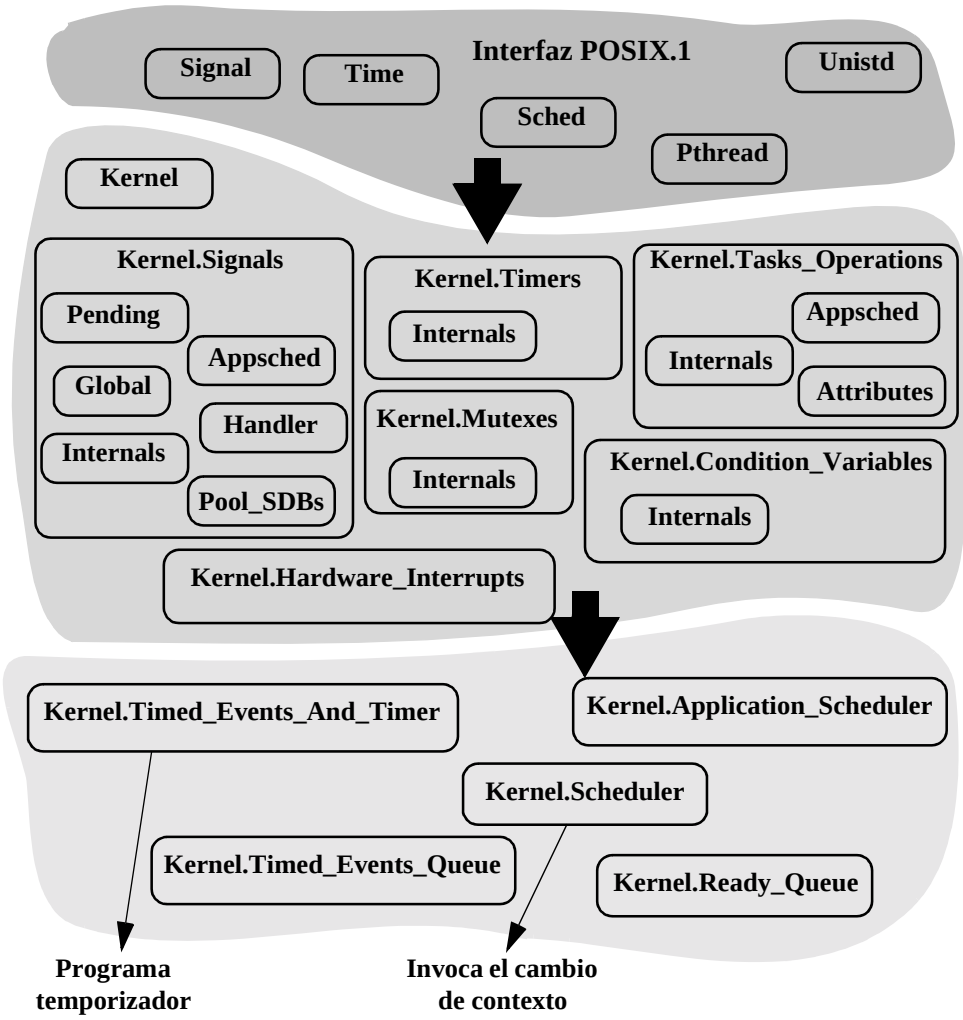

Figura 3.3: Paquetes del núcleo de MaRTE OS

para una parte diferenciada de la interfaz, o bien implementar las operaciones básicas en las que son divididos los servicios POSIX. De esta forma el acceso cruzado entre módulos se realiza utilizando las operaciones exportadas por alguno de los paquetes de tercer nivel (principalmente los denominados `Internals` dentro de cada módulo) y rara vez se accede a las del paquete principal.

La parte de la interfaz POSIX.1 aportada por el núcleo está formada por los paquetes `Signal`, `Time`, `Sched`, `Pthread` y `Unistd`, que se corresponden con los ficheros de cabecera C de sus mismos nombres. Existen otros ficheros de cabecera en la interfaz POSIX, cuyas funciones son aportadas por la librería estándar C. Los citados paquetes están constituidos en su mayor parte por renombrados de las funciones y subtipos de los tipos definidos en los paquetes del núcleo que implementan la interfaz POSIX, por lo que podría parecer que constituyen una capa redundante. Las razones para la existencia de esta capa extra son[1]:

- Existen operaciones y tipos que no se utilizan dentro del núcleo, representando servicios aislados sólo utilizados en la interfaz POSIX.1. Por claridad estos servicios se han separado del resto del núcleo. Ejemplos de estas interfaces son el tipo `timeval` y las

1. Otra razón para la existencia de esta capa es que en la versión de MaRTE OS que se encuentra disponible a fecha de escritura de esta memoria (V. 1.0), todavía hay partes del núcleo que presentan una interfaz similar a la POSIX-Ada , esta situación irá desapareciendo en próximas versiones.

funciones `sched_get_priority_max`, `sched_get_priority_min` y `sched_rr_get_interval`

- Algunos tipos definidos en el POSIX.1 no son equivalentes a los utilizados por MaRTE OS, por lo que son necesarias funciones que realicen la conversión antes de llamar a las operaciones del núcleo. Esta situación se da con el tipo `timespec` (una estructura con dos campos de tipo `int` para almacenar segundos y nanosegundos respectivamente), cuyo papel en el núcleo es desempeñado por un entero de 64 bits que permite realizar operaciones aritméticas de una forma mucho más eficiente.

La funcionalidad básica del núcleo, encargada de la gestión directa de las tareas y los eventos temporales es desempeñada por los paquetes que componen la capa inferior de la figura 3.3: `Kernel.Scheduler`, `Kernel.Timed_Events_Queue`, `Kernel.Ready_Queue`, `Kernel.Application_Scheduler` y `Kernel.Timed_Events_And_Timer`. Estos paquetes constituyen la parte del núcleo que se encuentra más estrechamente relacionada con la interfaz abstracta con el hardware, siendo desde ellos desde donde se realiza la programación del timer y las llamadas a la rutina de cambio de contexto.

3.4.4. Adaptación de la librería de tiempo de ejecución del compilador GNAT

El compilador GNAT [SCH94] implementa toda la semántica propia de las tareas Ada en su librería de tiempo de ejecución GNARL (GNU Ada Runtime Library) [GIE94]. Esta librería no facilita directamente el soporte para la ejecución concurrente, sino que éste se consigue mapeando cada tarea Ada sobre uno de los threads proporcionados por una implementación externa de threads, ya sea a nivel de librería o del propio sistema operativo. Lo normal en la mayoría de los sistemas, es que la implementación de threads presente una interfaz POSIX (Pthreads).

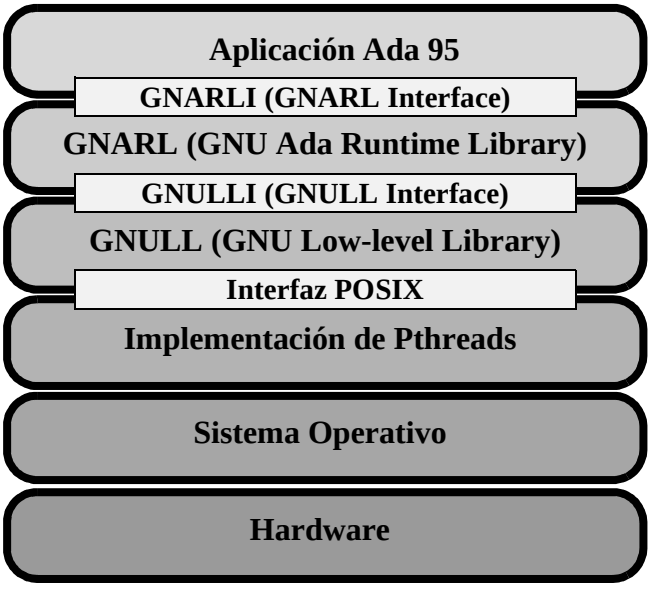

Figura 3.4: Aplicación Ada ejecutando sobre GNARL y Pthreads.

La figura 3.4 muestra el esquema de una aplicación Ada ejecutando sobre GNARL y sobre una implementación de Pthreads. El compilador GNAT genera el código correspondiente a la semántica de tareas mediante llamadas a GNARL. Existe una interfaz claramente definida, denominada GNARLI (GNARL Interface), que facilita la comunicación entre el código generado por el compilador y la librería de tiempo de ejecución.

La librería de tiempo de ejecución es en su mayor parte independiente de la plataforma sobre la que se ejecuta. Sólo una pequeña parte, denominada GNULL (GNU Low-level Library), debe ser modificada para adaptarse a las diferentes plataformas de ejecución. La interfaz entre las partes de la librería dependientes e independientes de la plataforma se denomina GNULLI (GNULL Interface). La modificación de GNULL permite ejecutar GNARL en sistemas operativos sin interfaz POSIX o incluso sobre máquina desnuda.

La adaptación de GNARL para MaRTE OS ha sido realizada a partir de una distribución estándar del compilador GNAT para el sistema operativo Linux, compilada para utilizar los Florida State University Threads (FSU-Threads) [MUE93], que es una implementación de Pthreads a nivel de librería. Dicha adaptación se muestra de forma esquemática en la figura 3.5

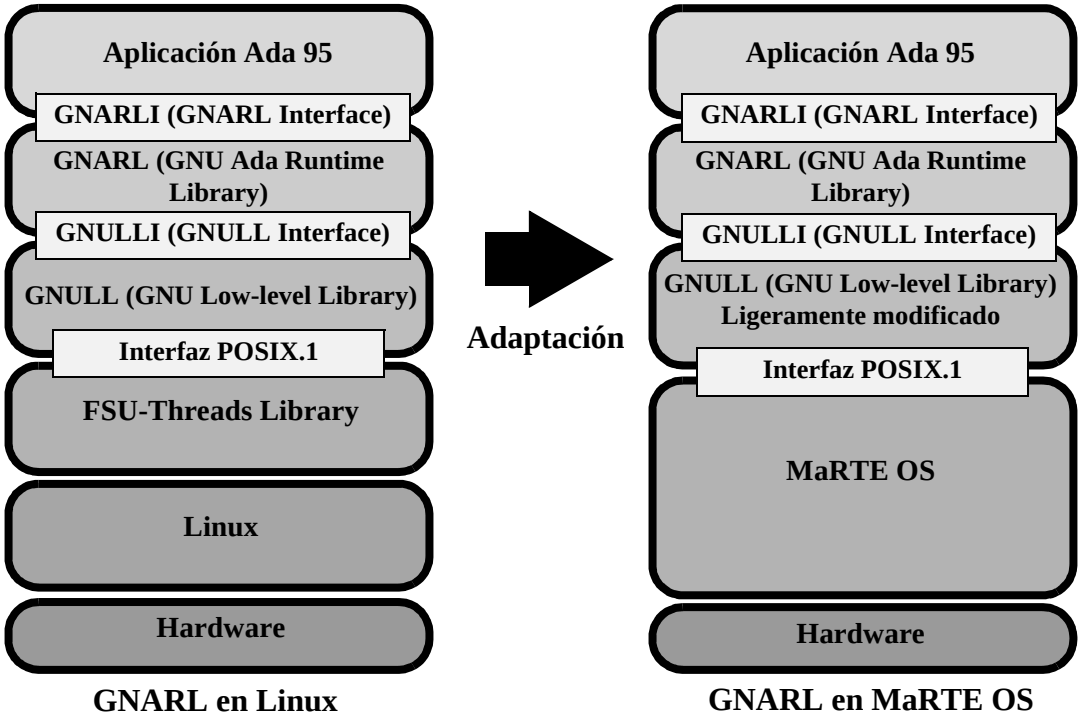

Figura 3.5: Adaptación de GNARL a MaRTE OS.

La adaptación es sencilla puesto que MaRTE OS también proporciona una interfaz POSIX, afectando básicamente a los paquetes que componen la capa de interfaz con el sistema operativo (GNULL) cuya estructura se muestra en la figura 3.6.

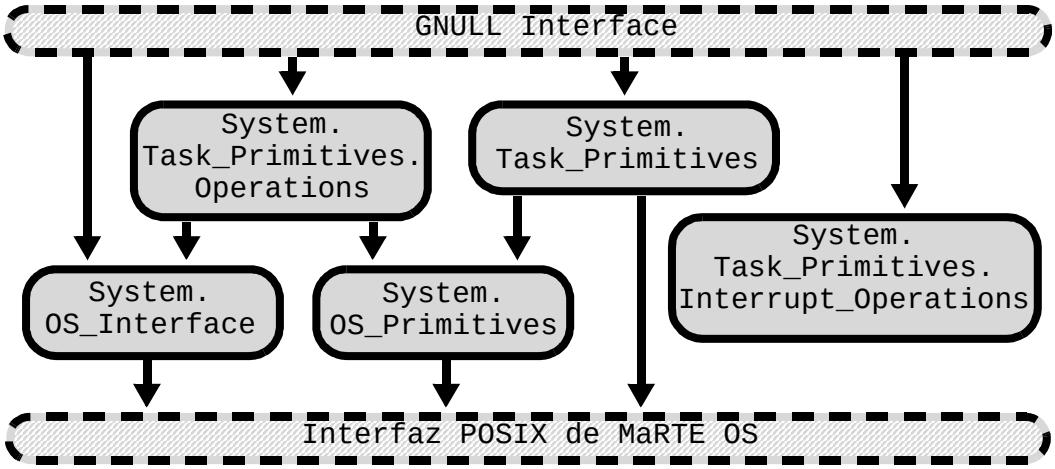

Figura 3.6: Estructura de GNULL

La adaptación a MaRTE OS de la librería de tiempo de ejecución del compilador GNAT ha requerido los cambios en los paquetes de GNULL que detallamos a continuación:

- `System.OS_Interface`: este paquete hace disponible la interfaz de Pthreads al resto de GNARL. Se compone principalmente de definiciones de tipos que mapean los definidos en la interfaz POSIX y de pragmas "Import" de las funciones C exportadas por la librería de Pthreads.

  Puesto que la interfaz POSIX C de MaRTE OS está escrita en Ada, en la adaptación de este paquete los pragmas "Import" desaparecen, invocándose directamente las funciones y procedimientos Ada de la interfaz POSIX de MaRTE OS. Por su parte, las definiciones completas de tipos se han reemplazado por subtipos de los tipos definidos en MaRTE OS.

  Aunque lo ideal hubiera sido reducir este paquete a un conjunto de funciones que renombraran las definidas en MaRTE OS, esto no es posible puesto que la mayoría de los tipos están definidos como privados limitados, por lo que es necesario realizar una conversión explícita.

- `System.OS_Primitives`: en este paquete se importan las funciones POSIX `nanosleep` y `gettimeofday`. En la adaptación a MaRTE OS se eliminan los pragmas "Import", invocándose directamente las funciones correspondientes definidas en el núcleo de MaRTE OS.

- `System.Task_Primitives.Operations`: este paquete implementa el manejador de la señal POSIX utilizada para abortar tareas asíncronamente. En su forma original, el manejador eleva una excepción `Abort_Signal`. Puesto que en las FSU-Threads el manejador se ejecuta en el contexto de la tarea a la que estaba dirigida la señal, GNARL no tiene problemas para saber en qué tarea ha sido elevada la excepción.

  Este mecanismo no puede ser utilizado en MaRTE OS ya que los manejadores de señal son ejecutados por un thread especialmente creado para ese fin. Una excepción elevada en el contexto de dicho thread no sería correctamente tratada por GNARL al no saber a que tarea Ada corresponde. En consecuencia, hemos optado por implementar una alternativa propuesta en las mismas fuentes de GNARL: en lugar de elevar la excepción, el manejador de señal cambia la dirección de retorno de la tarea cuya operación ha sido abortada. La nueva dirección de retorno corresponde a un procedimiento cuya única función consiste en elevar la excepción `Abort_Signal`. De esta forma, tan pronto como la tarea vuelva a tomar la CPU se elevará la excepción en el contexto adecuado.

- `System.Task_Primitives`: no es necesario modificarle.

- `System.Task_Primitives.Interrupt_Operations`: no es necesario modificarle.

Una alternativa a llamar directamente desde GNULL a las operaciones definidas en el núcleo de MaRTE OS hubiera sido importarlas mediante la utilización de pragmas "Import". Esta solución, que permitiría independizar la compilación de GNARL de la del núcleo de MaRTE OS, presenta dos inconvenientes que provocaron que fuera descartada:

- Deberían redefinirse en GNARL los tipos POSIX (pthread_mutex_t, pthread_cond_t, pthread_t, etc.) que ya se han definido en MaRTE OS, con el consiguiente riesgo de errores debido a la incorrecta sincronización de ambas versiones tras posibles cambios.

- Algunos errores que la solución elegida detecta en tiempo de compilación, como el error en el tipo de un parámetro, pasarían inadvertidos con esta segunda alternativa.

La única excepción en el uso de la interfaz POSIX por parte de la versión de GNARL adaptada la constituye la implementación de las funciones `nanosleep()`, `clock_gettime()` y `pthread_cond_timedwait()`. Todas ellas tienen en común que utilizan un parámetro de tipo `struct timespec`, por lo que si se utilizara su versión POSIX, habría que convertir los intervalos temporales manejados por GNARL mediante el tipo `Duration` a `struct timespec` para posteriormente, ya dentro del núcleo de MaRTE OS, transformarlos en cuentas del reloj hardware utilizado. Por razones de eficiencia se ha optado por evitar la transformación intermedia, para lo cual a la interfaz POSIX de MaRTE OS se ha añadido una versión de las citadas funciones que utiliza el tipo `Duration` en lugar del `struct timespec`.

Además de los citados paquetes de GNULL, también ha sido necesario modificar de forma muy ligera algunos paquetes de la librería Ada estándar:

- `Ada.Calendar`: es necesario añadir pragmas para forzar que este paquete sea elaborado después del paquete `System.OS_Primitives`.
- `Ada.Text_IO`: durante la creación de los descriptores de los dispositivos de entrada y salida estándar y de la salida de error no se llama a la función `is_regular_file` (no implementada en MaRTE OS). En lugar de eso, se asigna directamente el valor `FALSE` al campo `Is_Regular_File` de los descriptores mencionados. Esto se hace así puesto que en MaRTE OS no existe sistema de ficheros, por lo que los citados dispositivos no deben ser tratados como tales.

Como se ha expuesto, la adaptación de GNARL a MaRTE OS resulta sencilla, siendo casi todos los cambios relativos a la interfaz con la capa POSIX, y por tanto sin implicar ninguna modificación en el modo de operación de la librería. Además, los cambios a realizar se encuentran localizados en un conjunto muy reducido de paquetes. Esto permite que la adaptación de nuevas versiones de la librería resulte un proceso sencillo siempre y cuando no se produzcan cambios importantes en la estructura interna de GNARL. Así, la adaptación de la librería de tiempo de ejecución de la versión 3.13p del compilador realizada tomando como base la de la versión 3.12p resultó inmediata, requiriendo sólamente unas pocas horas de trabajo. Algo similar ocurrió con la posterior adaptación de la versión 3.14p.

Existe una estrategia de adaptación de GNARL distinta a la expuesta para MaRTE OS. Esta estrategia, seguida por el “Open Ravenscar Real-Time Kernel” (ORK) [PUE00] y su antecesor “Jose’s Tasking Kernel” (JTK) [RUI99], persigue la máxima integración posible entre el núcleo y la librería de tiempo de ejecución. Por este motivo, el núcleo presenta una interfaz pensada expresamente para cubrir las necesidades de GNARL, no pretendiendo ofrecer además una interfaz POSIX a las aplicaciones. En ORK se opta por modificar profundamente GNULL, de forma que se integre lo más íntimamente posible en el propio núcleo, con el objeto de reducir el tamaño y de mejorar las prestaciones del sistema. Esa solución tiene como inconvenientes que exige un profundo conocimiento de la estructura interna de GNARL y complica la adaptación de nuevas versiones de la librería. Por otra parte, si se desea proporcionar una interfaz POSIX, habría que construirla sobre la interfaz orientada a GNARL con lo que el núcleo en su conjunto resultaría más complejo.

3.5. Gestión de interrupciones a nivel de aplicación

Las aplicaciones empotradas se relacionan muy estrechamente con los dispositivos hardware del sistema, en particular con los sensores, actuadores o puertos de comunicaciones que las permiten interactuar con el entorno físico que las rodea. En muchas ocasiones estos dispositivos hardware utilizan las interrupciones como mecanismo de comunicación con la aplicación. Por

consiguiente, resulta fundamental que los sistemas operativos proporcionen a las aplicaciones una interfaz que, al menos, las permita instalar procedimientos para ser ejecutados directamente tras la generación de una interrupción (procedimientos manejadores) y operaciones para habilitar y deshabilitar las interrupciones.

Lo normal en la mayoría de los sistemas operativos es que los procedimientos manejadores se ejecuten a la más alta prioridad del sistema, expulsando de la CPU a cualquier tarea que pudiera encontrarse en ejecución en ese momento. Para minimizar los tiempos de bloqueo debidos a la ejecución de los manejadores, en la mayoría de los sistemas operativos se trata de minimizar el código que debe ser realizado por ellos, reduciéndolo únicamente a las operaciones directas sobre el dispositivo, como son la lectura o escritura del nuevo dato, la rehabilitación del dispositivo para una nueva interrupción, etc. Para ello, la mayoría de los sistemas operativos proporcionan mecanismos que permiten ejecutar, a una prioridad intermedia que en ocasiones puede ser especificada por el programador, la parte más costosa de la atención de la interrupción (procesado de la información, almacenamiento definitivo, etc.). Este es el caso de mecanismos como los “Bottom Halves” de Linux [RUB01], o las “Deferred Procedure Calls” (DPC) de Windows [RAM98]. Por lo general, en los sistemas operativos de tiempo real esta función se deja en manos de tareas que son activadas desde los manejadores de interrupción utilizando primitivas de sincronización específicas para ese fin.

Nuestro sistema operativo también proporciona un mecanismo con una funcionalidad similar a los nombrados anteriormente. La solución adoptada se basa en la posibilidad de bloquear tareas a la espera de que se produzca una interrupción hardware. Utilizando esta funcionalidad es posible desplazar parte o todo el procesado de la interrupción a una tarea con la prioridad deseada por el programador, lo que permite reducir al mínimo los tiempos de bloqueo sufridos por las tareas de mayor prioridad que la utilizada como tarea “manejadora”.

MaRTE OS proporciona una interfaz para la gestión de interrupciones a nivel de aplicación en la que, además de la operación anteriormente mencionada, se definen operaciones para la habilitación y deshabilitación de interrupciones, así como para la instalación de manejadores de interrupción tradicionales que son ejecutados a la más alta prioridad del sistema inmediatamente después de producirse la interrupción. Existen dos versiones equivalentes de la interfaz, una en lenguaje Ada y otra en C. La interfaz Ada se encuentra definida en el paquete `Hardware_Interrupts`, mientras que la versión C se define en el fichero de cabeceras `<hwinterrupts.h>`. Las operaciones proporcionadas por ambas versiones de la interfaz son equivalentes una a una, por lo que procederemos a explicarlas de forma simultánea.

3.5.1. Fuentes de interrupción

Se define un conjunto de constantes que permiten identificar cada una de las fuentes de interrupción existentes en el sistema. En el caso de la interfaz C se utilizan constantes simbólicas mientras que la interfaz Ada son constantes del tipo discreto `Hardware_Interrupt`.

Algunas de estas fuentes de interrupción son utilizadas por el sistema operativo MaRTE OS, considerándose reservadas, por lo que no podrán ser utilizadas por ninguna de las operaciones definidas en la interfaz. En la implementación actual de MaRTE para PCs, las interrupciones reservadas son las correspondientes al temporizador hardware y al teclado (`TIMER_INTERRUPT` y `KEYBOARD_INTERRUPT`).

3.5.2. Habilitación y deshabilitación de interrupciones

La ejecución de un procedimiento manejador de interrupción, y la posterior activación de la tarea bloqueada (si la hubiera), únicamente se producirá mientras la interrupción correspondiente se encuentre habilitada. En caso contrario, la interrupción permanecerá pendiente hasta su habilitación.

La interfaz proporciona operaciones que permiten habilitar o deshabilitar de una sola vez todas las interrupciones del sistema (incluidas las reservadas). La versión Ada de tales operaciones es:

```
procedure Enable_All;
procedure Disable_All;
```

El equivalente C de los procedimientos anteriores son las funciones:

```
void hwinterrupts_enable_all ();
void hwinterrupts_disable_all ();
```

También se proporcionan operaciones que permiten habilitar y deshabilitar de forma independiente cada interrupción. A continuación se muestra la interfaz de los procedimientos proporcionados por la interfaz Ada:

```
procedure Enable_Hardware_Interrupt
     (Interrupt : in Hardware_Interrupt);
procedure Disable_Hardware_Interrupt
     (Interrupt : in Hardware_Interrupt);
```

Las funciones equivalentes proporcionadas por la interfaz C son:

```
int hwinterrupts_enable (int hwint);
int hwinterrupts_disable (int hwint);
```

La semántica de estas operaciones asegura que tras su ejecución la interrupción elegida siempre será habilitada o deshabilitada, sin embargo no dice nada sobre si afectarán en el mismo sentido a otras interrupciones del sistema. Así, en algunas arquitecturas hardware podrá ocurrir que la habilitación de una interrupción suponga también la habilitación de todas las interrupciones más prioritarias que ella, y al revés, su deshabilitación también implique a las interrupciones de nivel inferior. Incluso pueden existir arquitecturas en las que sólo sea posible actuar sobre todas las interrupciones de forma simultánea, produciéndose en todos los casos la habilitación o deshabilitación de todas las interrupciones del sistema independientemente de la interrupción indicada en la operación.

3.5.3. Instalación de manejadores de interrupción

La interfaz proporciona una operación que permite instalar un procedimiento manejador de interrupción. Este manejador será ejecutado por el sistema operativo inmediatamente después de la generación de la interrupción y siempre que ésta se encuentre habilitada. El procedimiento Ada que permite instalar manejadores de interrupción es el siguiente:

```
type Interrupt_Handler_Procedure is access procedure;
procedure Install_Handler
     (Interrupt : in Hardware_Interrupt;
      Handler   : in Interrupt_Handler_Procedure);
```

Con ese mismo propósito, la interfaz C define la función `hwinterrupts_install_handler()`:

```
int hwinterrupts_install_handler (int hwint,
                                  void (handler) (void));
```

No está permitido asignar manejadores de interrupción a las interrupciones reservadas por el sistema operativo. Cuando `Install_Handler` es invocada para una interrupción reservada, se eleva la excepción `RESERVED_INTERRUPT`. Por su parte, en esa misma situación `hwinterrupts_install_handler()` retorna el código de error `EPERM`.

3.5.4. Bloqueo de una tarea a la espera de una interrupción hardware

Una tarea Ada puede bloquearse a la espera de la generación de una interrupción utilizando el procedimiento:

```
procedure Wait (Interrupt : in Hardware_Interrupt);
```

Los threads C pueden utilizar la función equivalente:

```
int hwinterrupts_wait (int hwint);
```

En el caso de que la aplicación haya instalado un procedimiento manejador de la interrupción, éste será ejecutado antes de que se produzca la activación de la tarea o el thread bloqueados.

En cada interrupción sólo puede encontrarse bloqueada una única tarea (o thread), que será la última que haya ejecutado la operación de espera para esa interrupción. En el caso de que una tarea ejecute el procedimiento `Wait` sobre una interrupción en la que ya se encontraba bloqueada otra tarea, esta última finalizará la espera de la interrupción elevándose para ella la excepción `INTERRUPT_WAITING_WAS_ABORTED`. En esta misma situación la función `hwinterrupts_wait()` finaliza su ejecución retornando el código de error `EINTR`.

No es posible bloquear una tarea o thread a la espera de una interrupción reservada. Cuando la operación `Wait` es invocada para una interrupción reservada, se eleva la excepción `RESERVED_INTERRUPT`. Por su parte, en esa misma situación `hwinterrupts_wait()` retorna el código de error `EPERM`.

3.5.5. Inhibición de la activación de la tarea bloqueada

En ocasiones, tras la generación de una interrupción, el procedimiento manejador puede decidir que no es necesario activar a la tarea manejadora. Esto puede ocurrir, entre otras muchas circunstancias, porque todavía no ha llegado el dato esperado o porque no se ha completado aún todo el bloque de información que necesita la tarea para llevar a cabo su función. Para inhibir la activación de la tarea bloqueada un procedimiento manejador escrito en Ada debe invocar el procedimiento:

```
procedure Do_Not_Activate_Waiting_Task;
```

En el caso de que el manejador haya sido escrito usando el lenguaje C, podemos utilizar la función:

```
void hwinterrupts_do_not_activate_waiting_thread ();
```

En el caso de que no exista ninguna tarea bloqueada, la ejecución de las operaciones anteriormente mencionadas no tendrá ningún efecto.

3.6. Implementación

3.6.1. Interfaz abstracta con el hardware en PCs

Como plataforma inicial de desarrollo de MaRTE OS se ha elegido el PC con procesador 80386 o superior. Su elección se ha debido a que constituye una arquitectura para la que es fácil disponer de herramientas de desarrollo (compiladores, depuradores, etc.). Otra razón es que para ella existe una extensa documentación y gran cantidad de código disponible (un ejemplo lo constituye el conjunto de librerías OSKit previamente mencionado). También ha pesado en su elección su amplísima utilización, lo que facilitará que MaRTE OS pueda ser utilizado como herramienta de docencia o investigación por otros grupos de trabajo.

La implementación de los relojes y de los temporizadores para esta arquitectura depende del procesador de que se disponga. En el caso de que el procesador sea un 80386 o un 80486, el dispositivo utilizado tanto para los relojes (medida del tiempo absoluto o de sistema) como para los temporizadores (generación de eventos temporales en el instante requerido) es el "Programmable Interval Timer" (PIT) [TRI92]. El PIT es un dispositivo estándar en la arquitectura PC que tiene 3 contadores de 16 bits cada uno, controlados por una señal de reloj de 883.1 nseg. de periodo, lo que permite un intervalo máximo programable de algo más de 50 mseg. El principal problema para la utilización del PIT es que el acceso a sus registros se realiza a través del bus de E/S del PC, por lo que constituye una operación muy lenta, típicamente de varios microsegundos.

La estrategia utilizada para la programación del PIT es la misma que la usada en el RT-Linux [RTLIN] [YOD99]. El contador 0 se utiliza para generar las interrupciones de los temporizadores y para llevar la cuenta del tiempo absoluto, por lo que siempre se reprograma después de su expiración. Así, salvo que se encuentre pendiente algún evento temporal más urgente, dicho contador se reprograma para producir una interrupción cada 50 mseg. Después de cada interrupción, el intervalo programado se suma al tiempo total. El tiempo utilizado en la reprogramación se mide utilizando el contador 2 (habitualmente empleado en el PC para generar sonidos de tonos diferentes a través del altavoz). Este intervalo también se suma al tiempo total para evitar que los tiempos de reprogramación produzcan un retraso acumulativo en el reloj del sistema.

Con esta estrategia, la obtención del tiempo transcurrido desde la inicialización del sistema consiste en sumar el tiempo total acumulado mas el valor actual del contador 0 del PIT. Si se desea obtener la hora actual, al valor anteriormente calculado habrá que sumarle la hora de arranque del sistema, la cual fue leída del "Real Time Clock" (RTC) del PC durante la inicialización de MaRTE OS. Cuando se utiliza el PIT para la implementación del reloj y del temporizador, la resolución en MaRTE OS de ambos dispositivos es de 838.1 ns (igual que los contadores del PIT).

Cuando se dispone de un procesador Pentium, la hora actual puede obtenerse utilizando el "Time-Stamp Counter" (TSC) [INTE3]. El TSC, tal y como está implementado en las familias de procesadores Pentium y P6, es un contador de 64 bits que toma el valor cero con la inicialización del procesador durante el arranque del sistema. Posteriormente, el contador se incrementa a cada ciclo del procesador, incluso mientras el procesador esta detenido por una instrucción `HLT` o por el estado del pin `STPCLK#`. La lectura del TSC constituye una operación

muy rápida (89 ns en un Pentium III a 550 MHz), debido a que requiere una única instrucción máquina y a que no precisa de la utilización del bus de E/S, puesto que este contador se encuentra dentro el propio procesador.

La hora actual se obtiene sumado la hora de inicialización de MaRTE OS (leída del RTC) más el número del ciclos que lleva contados el TSC en ese momento menos el número de ciclos contados por el TSC en el instante de inicialización. Con esta estrategia la resolución del reloj es excelente. Por ejemplo, la resolución en un Pentium a 550 MHz es 1.8 ns.

Cuando se dispone del TSC el PIT no se utiliza para la medida de tiempos, aunque su contador 0 continúa siendo utilizado como fuente de interrupciones para los temporizadores. Sin embargo, en este caso no es necesario utilizar el contador 2 para medir el tiempo de reprogramación, ya que el tiempo total del sistema se lleva de forma automática por el TSC. El tiempo máximo programable en el contador 0 es 50 ms, por lo que para intervalos mayores que ese valor deberán programarse una o más interrupciones intermedias. En consecuencia, con esta estrategia aún debemos pagar el precio de usar el bus de E/S del PC para acceder a un contador del PIT.

Para procesadores P6 (Pentium II) y superiores los tiempos de programación del temporizador pueden reducirse enormemente gracias a la utilización del temporizador incluido en el "Avanced Programmable Interrupt Controller" (Local APIC) [INTE3]. Aunque su función principal es la de gestionar las interrupciones, el Local APIC también cuenta con un temporizador programable de 32 bits, cuya base de tiempo se deriva del reloj del bus de procesador, y que puede programarse para interrumpir al procesador con el vector de interrupción que se desee. La frecuencia del bus del procesador en el Pentium III a 550 MHz utilizado para pruebas es de 100 MHz. Por lo tanto, los temporizadores en dicha máquina tendrán una resolución de 10 ns.

El Local APIC se encuentra deshabilitado después de la inicialización del procesador. Su habilitación se realiza en dos pasos: primero, debe ponerse a uno un bit en el registro específico del procesador ("machine-specific register") denominado `APIC_BASE_MSR`, lo que posibilita el acceso a sus registros; a continuación debe habilitarse mediante la escritura del valor apropiado en su registro "Spurious-Interrupt Vector".

El acceso a los registros del APIC se realiza leyendo y escribiendo en unas direcciones de memoria determinadas. Cuando el APIC se encuentra habilitado sus registros se encuentran mapeados sobre las citadas direcciones de memoria. Los tiempos de lectura y escritura sobre sus registros son muy cortos. Por ejemplo, la escritura sobre el registro "Inital Count" tarda únicamente 62 ns en un Pentium III a 550 MHz.

La tabla 3.1 muestra una comparación entre los tiempos que tardan los servicios de temporización en función de los dispositivos hardware utilizados. Los tiempos que aparecen en la tabla se han obtenido en un Pentium III a 550 MHz y están medidos en nanosegundos.

Tabla 3.1: Comparación de los servicios de temporización

Operación	***PIT***	***TSC + PIT***	***TSC + APIC***
Lee hora actual	3100	105	105
Programa temporizador	12900	3000	324

Como controlador de interrupciones se utiliza el 82C59A “Programmable Interrupt Controller” (PIC) [TRI92], un dispositivo capaz de gestionar 8 fuentes de interrupción diferentes. El PC cuenta con dos de estos dispositivos conectados en cascada (“PIC master” y “PIC slave”), con los que se pueden manejar 15 dispositivos generadores de interrupciones. La asignación estándar de las interrupciones en un PC y los vectores asignados en MaRTE OS se muestra en la tabla 3.2. En el caso de que se utilice el temporizador del APIC, durante la configuración del sistema MaRTE OS le asigna el vector de interrupción siguiente al último del PIC slave (vector 0x30).

Tabla 3.2: Interrupciones hardware en MaRTE OS

PIC Master	PIC Slave	Dispositivo Interruptor
IRQ 0 (Vector 0x20)		PIT (contador 0)
IRQ 1 (Vector 0x21)		Teclado
IRQ 2 (Vector 0x22)		PIC Slave
	IRQ 8 (Vector 0x28)	Real Time Clock (RTC)
	IRQ 9 (Vector 0x29)	Bus
	IRQ 10 (Vector 0x2A)	Disponible
	IRQ 11 (Vector 0x2B)	Disponible
	IRQ 12 (Vector 0x2C)	Disponible
	IRQ 13 (Vector 0x2D)	Coprocesador matemático
	IRQ 14 (Vector 0x2E)	Controlador de disco duro
	IRQ 15 (Vector 0x2F)	Disponible
IRQ 3 (Vector 0x23)		Puerto serie 2
IRQ 4 (Vector 0x24)		Puerto serie 1
IRQ 5 (Vector 0x25)		Puerto paralelo 2
IRQ 6 (Vector 0x26)		Controlador de disquete
IRQ 7 (Vector 0x27)		Puerto paralelo 1
Pseudo IRQ 16 (0x30)		Local APIC (temporizador)

Tras la inicialización del sistema los vectores de interrupción comienzan apuntado a una envoltura genérica para los manejadores del usuario, que está formada por una primera parte específica para cada nivel de interrupción y una segunda parte dependiente del dispositivo controlador o generador de la interrupción (“PIC master”, “PIC slave” o “Local APIC”). La primera parte permite identificar de forma automática el nivel de la interrupción, mientras que la segunda se encarga de las operaciones de rehabilitación del dispositivo controlador, así como de invocar el manejador de interrupción instalado por el usuario o por el propio núcleo de MaRTE OS. Los manejadores de interrupción de usuario se almacenan en un array de punteros a funciones, cuyo índice es el nivel de la interrupción.

3.6.2. Cola de tareas ejecutables

La cola de tareas ejecutables es una de las estructuras más importantes del núcleo. En ella se encuentran ordenadas por orden de prioridad todas las tareas que están en disposición de ejecutar en un momento dado. Su importancia radica en que todas las acciones de planificación que ocurren en el núcleo, tales como la inserción o extracción de tareas o la búsqueda de la tarea más prioritaria en disposición de tomar la CPU, incluyen operaciones sobre esta cola, con lo que sus prestaciones influyen en gran medida en las prestaciones globales del sistema.

Es fundamental, por tanto, elegir una implementación que permita operaciones muy rápidas con tiempos de ejecución acotados para las condiciones típicas en las que va a operar el núcleo:

- Un número de tareas entre 20 y 50, lo cual es considerado típico para el tipo de aplicaciones empotradas a las que está destinado.
- Muy pocas tareas en cada nivel de prioridad (normalmente una), ya que la teoría RMA [KLE93] [LIU73] recomienda que cada tarea tenga una prioridad diferente.
- Entre 32 y 64 niveles de prioridad diferentes. El estándar POSIX exige un mínimo de 32 prioridades, pero como se ha expuesto anteriormente, para un número mayor de tareas resultará deseable disponer de más niveles de prioridad.
- Las operaciones a realizar sobre la cola, y por tanto aquellas para las que habrá que estudiar su comportamiento, son: identificación de la tarea más prioritaria, encolado de una tarea después de todas las de su misma prioridad y extracción de la cabeza de la cola (tarea más prioritaria).

Bajo las condiciones descritas varias alternativas fueron analizadas:

- Listas simple y doblemente enlazadas (con puntero al próximo elemento y al anterior), ordenadas y sin ordenar. Es una implementación muy simple y eficiente para números muy pequeños de tareas. Sin embargo, sus tiempos de extracción y/o inserción crecen linealmente con el número de tareas, lo que hace que esta implementación deba ser descartada para el número de tareas considerado. Estas colas son utilizadas en algunos sistemas operativos. Así RT-Linux utiliza una lista simplemente enlazada sin ordenar y la librería FSU-Threads utiliza una lista simplemente enlazada ordenada con punteros a la cabeza y a la cola.
- Tablas de troceado (“hash tables”) [FEL93]. Es una implementación muy utilizada en otros entornos de programación, pero no es válida para aplicaciones de tiempo real, ya que presenta tiempos de respuesta no acotados.
- Montículo binario [ALL95]. Constituye una posibilidad muy eficiente. De hecho, es la opción elegida para la cola de eventos temporales (ver apartado 3.6.3, “Eventos temporales”). Su utilización para la cola de tareas ejecutables debe ser descartada ya que no permite conservar el orden FIFO entre tareas de igual prioridad.
- Array de colas de prioridad. Para la implementación de las colas existen varias alternativas, pudiendo ser simple o doblemente enlazadas (con puntero al elemento siguiente y al anterior), y con puntero a la cabeza o a la cabeza y a la cola. Se descarta la utilización de colas doblemente enlazadas, puesto que la posibilidad de conocer el elemento anterior a uno dado no supone ninguna ventaja para el uso en cuestión, mientras que los tiempos de inserción y extracción de este tipo de colas son mayores que para las simplemente enlazadas. Aunque el disponer de un puntero al último elemento puede parecer una ventaja, se comprobó que su utilización no compensa cuando el

número de tareas en la cola de prioridad es pequeño. El sistema operativo HP-RT [HPR94] utiliza esta solución.

- Array de colas de prioridad con mapa de bits. Constituye una implementación idéntica a la anterior, pero además se utiliza un mapa de bits que sirve para indicar las colas de prioridad no vacías. Esta es la implementación utilizada por el sistema operativo RTEMS.

El análisis anterior reduce mucho el número de implementaciones a estudiar, limitando a dos las posibles opciones: el array de colas de prioridad y el array de colas de prioridad con mapa de bits. Una comparativa de las prestaciones de ambas colas es mostrada en la tabla 3.3, las medidas fueron realizadas para 20 tareas y 32 niveles de prioridad en un Pentium a 133 MHz.

Tabla 3.3: Análisis de prestaciones de las colas de prioridad

Implementación de cola de prioridad	**Encola** (cualquier prioridad, única tarea en su prioridad)	**Busca la tarea más prioritaria** (sólo una tarea en la lista. Situada en la cola de menor prioridad)	**Elimina más prioritaria** (que se supone conocida)
Array de colas de prioridad	107 ns	1033 ns	80 ns
Array de colas de prioridad con mapa de bits	128 ns	479 ns	113 ns

La gran diferencia existente en la operación de búsqueda de la tarea más prioritaria, diferencia que crece al utilizar más niveles de prioridad, junto con el hecho de tratarse de una operación que se va a ser invocada muy frecuentemente, provocó que se optara por el array de colas de prioridad con mapa de bits.

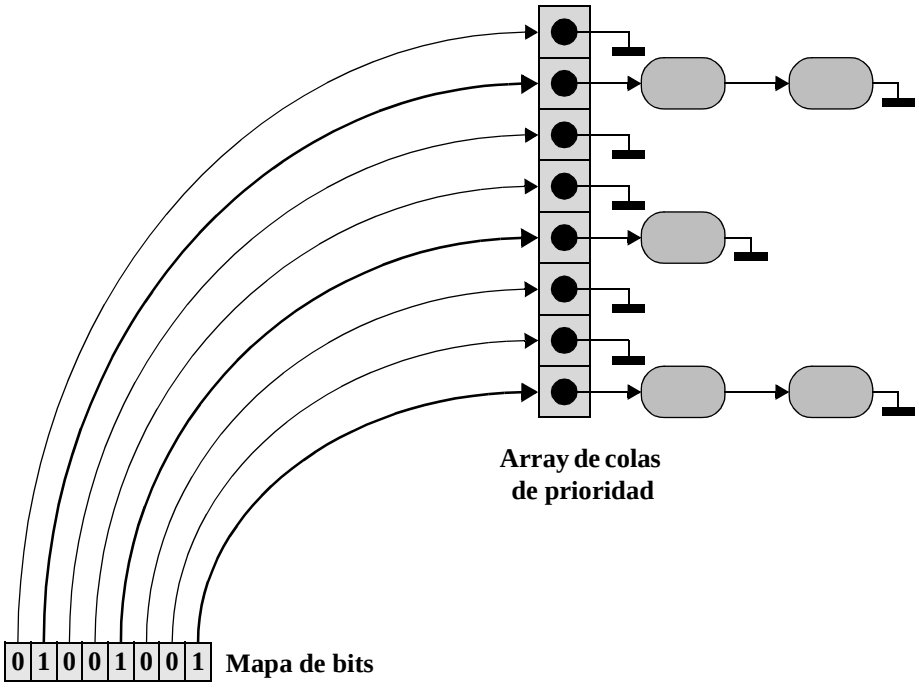

Figura 3.7: Esquema de la cola de tareas ejecutables

El esquema de la implementación elegida se muestra en la figura 3.7. Cada bit a uno en el mapa de bits indica que la cola correspondiente del array de colas de prioridad tiene algún elemento, o lo que es lo mismo, que existe alguna tarea activa para ese nivel de prioridad. A diferencia de lo mostrado en la figura en la que el mapa de bits es de longitud 8, en la implementación real se utilizan mapas de longitud 32, en el caso de que el sistema se configure para más de 32 prioridades, se utiliza un array de mapas de bits

La búsqueda de la tarea más prioritaria consiste en buscar el bit más significativo a 1 del mapa de bits y obtener la tarea que se encuentre a la cabeza de la cola correspondiente. Los procesadores Intel X86 cuentan con una instrucción máquina que realiza la búsqueda del primer bit a 1 en un registro de 32 bits (instrucciones `BSF` y `BSR` [TRI92]), por lo que esta operación resulta muy rápida en la citada arquitectura. En el caso de portar MaRTE OS a otra arquitectura que no cuente con una instrucción equivalente, sería preciso comprobar si la implementación de cola de tareas ejecutable basada en mapas de bits continúa siendo la más eficiente.

3.6.3. Eventos temporales

Los eventos temporales se utilizan dentro del núcleo de MaRTE OS para representar acciones programadas para ser ejecutadas en instantes de tiempo determinados. Dependiendo de la acción programada, los eventos temporales pueden dividirse en:

- *Suspensión temporizada*: representa el instante de tiempo en el que debe activarse una tarea suspendida como consecuencia de la ejecución de las instrucciones `delay` o `delay until` de Ada o de las funciones `sleep()`, `nanosleep()` o `clock_nanosleep()` del estándar POSIX.
- *Espera temporizada en una variable condicional*: representa el instante en el que debe abortarse una operación `pthread_cond_timedwait()`.
- *Espera temporizada de eventos de planificación*: representa el instante en el que debe abortarse una operación de espera de eventos de planificación con tiempo límite. La semántica de esta operación se describe en el capítulo 4, “Interfaz de usuario para la definición de algoritmos de planificación”.
- *Planificación*: representa el instante de tiempo en el que debe invocarse el planificador debido, bien a que una tarea de política “round-robin” ha finalizado su rodaja temporal, o bien a que una tarea de política “servidor esporádico” ha agotado su capacidad de ejecución.
- *Recarga de un servidor esporádico*: representa el instante en el que debe recargarse parte o toda la capacidad de ejecución de un servidor esporádico. La implementación de las políticas de planificación, y entre ellas el servidor esporádico, se expone en el apartado 3.6.4, “Planificación de tareas”.
- *Temporizador*: representa el instante en el que debe producirse la expiración de un temporizador. Los temporizadores pueden estar basados en el reloj de tiempo real, en el monótono o en el de tiempo de ejecución de alguna de las tareas del sistema.

Todos los eventos temporales, salvo los correspondientes a los temporizadores de tiempo de ejecución, se encuentran ordenados de mayor a menor urgencia en la cola de eventos temporales. La ordenación en dicha cola se realiza en base a su instante de expiración, expresado como un tiempo absoluto referido al reloj monótono utilizado internamente por el núcleo de MaRTE OS.

Esta cola es otra de las estructuras que tienen un gran impacto sobre las prestaciones globales del núcleo, ya que algunas de las operaciones más frecuentemente utilizadas por las aplicaciones de tiempo real, como son las de suspensión temporizada, espera temporizada y uso de temporizadores, involucran operaciones de inserción o extracción de elementos en dicha cola. La mayoría de las extracciones corresponderán a la eliminación del primer evento de la cola, una vez que éste haya expirado, aunque también en ocasiones será necesario eliminar un elemento que no ocupe la cabeza, como en el caso de una espera temporizada que finalice antes del tiempo límite. Por su parte las inserciones consistirán en el encolado en orden de un evento.

La implementación elegida deberá permitir realizar las operaciones anteriormente citadas lo más eficientemente posible para un número de eventos similar al de tareas (entre 20 y 50). Debe buscarse una solución distinta a la empleada para la cola de tareas ejecutables, ya que en este caso, la llave de ordenación es el tiempo, el cual está descrito por un rango continuo de valores.

Se procedió a evaluar distintas implementaciones: cola ordenada simplemente enlazada (utilizada por GNAT y FSU-Threads), cola ordenada doblemente enlazada (utilizada por RTEMS) y montículo binario [ALL95]. Esta última constituye una estructura extensamente utilizada para la implementación de colas de prioridad, teniendo la forma de un árbol binario completamente lleno, con la posible excepción del nivel más bajo, que se llena de izquierda a derecha. En él se cumple que el elemento más prioritario se encuentra en la raíz, y además, puesto que cualquier subárbol debe ser también un montículo, cada nodo es más prioritario que todos sus descendientes. Para esta implementación, cuya estructura se muestra en la figura 3.8, las operaciones de extracción e inserción dependen linealmente del número de niveles del árbol, siendo por tanto `O(log n)`.

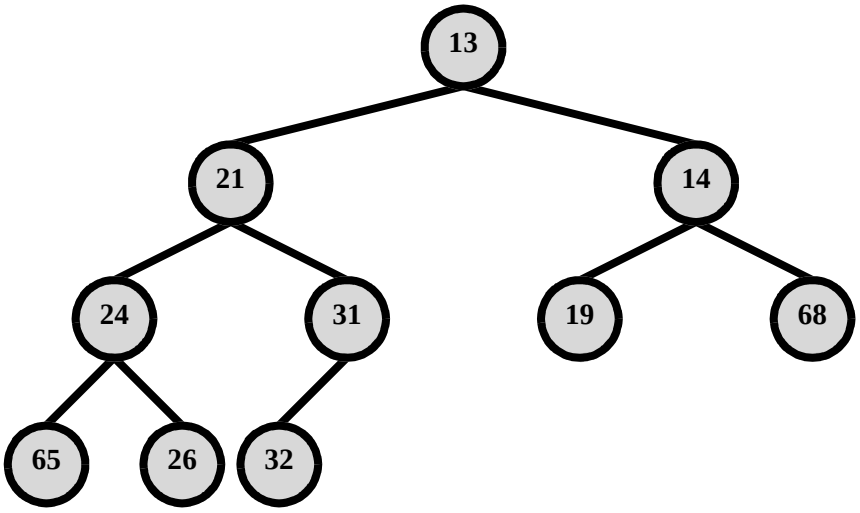

Figura 3.8: Montículo binario

Para las operaciones a realizar sobre la cola de eventos temporales no supone ninguna ventaja disponer del puntero al elemento anterior a uno dado, por lo que la cola doblemente enlazada es descartada frente a la de enlace simple. En la tabla 3.4 se comparan las prestaciones de la cola simplemente enlazada frente al montículo binario. Los tiempos, medidos en un Pentium a 133 MHz, representan los valores de peor caso de las operaciones. En el caso de la cola simplemente enlazada, los tiempos mostrados para la operación de encolado corresponden a la inserción del elemento que va a ocupar la última posición de la cola, mientras que para el montículo binario, la situación de peor caso se da al encolar el elemento que ocupará la primera posición de la cola. En cuanto a la extracción, la situación de peor caso para el montículo se da al eliminar la cabeza de la cola, mientras que para la lista enlazada ocurre al eliminar el último elemento. Los tiempos

de la extracción de un elemento para la cola simplemente enlazada no se muestran en la tabla ya que son similares a los de inserción.

Tabla 3.4: Análisis de prestaciones de las colas de eventos

Implementación	Encola en orden			Elimina cabeza		
	15 eventos	20 eventos	50 eventos	15 eventos	20 eventos	50 eventos
Montículo binario	790 ns	910 ns	1060 ns	930 ns	1040 ns	1400 ns
Cola simplemente enlazada	1720 ns	2620 ns	7200 ns	100 ns		

Para cualquier evento que se programe en el sistema deberá producirse su encolado y posteriormente su extracción de la cola, por tanto, la implementación más apropiada será la que minimice la suma de ambos tiempos. Aplicando este criterio el montículo supera a la cola simplemente enlazada incluso para números de eventos inferiores a los considerados como normales en las aplicaciones a las que está destinado MaRTE OS. La ventaja del montículo es mayor aún en el caso de que el evento sea eliminado de la cola antes de llegar a la cabeza.

Además de los tipos de eventos temporales anteriormente citados, en MaRTE OS se utiliza otro tipo de evento que permite representar acciones programadas para ser ejecutadas cuando el tiempo de ejecución de una tarea alcance un determinado valor. Estos eventos son utilizados para implementar los temporizadores de tiempo de ejecución y también en la planificación de tareas de políticas “round-robin” y servidor esporádico, como se expondrá en el apartado 3.6.4, “Planificación de tareas”.

A diferencia de los demás eventos, los de tiempo de ejecución no se encolan en la cola de eventos temporales. Esto se hace así para evitar las inserciones y extracciones en la cola que se producirían cada vez que cambiara la tarea en ejecución. Puesto que lo normal es que el número de eventos de tiempo de ejecución por tarea no sea muy grande, se ha optado por implementar una cola simplemente enlazada para cada tarea.

En todo momento, el temporizador hardware se programa en función del evento más urgente del sistema, siendo éste el más urgente de entre los que se encuentran en las cabezas de la cola global de eventos temporales y de la cola de eventos de tiempo de ejecución de la tarea ejecutando en ese momento.

3.6.4. Planificación de tareas

MaRTE OS soporta los tres algoritmos de planificación definidos en el estándar POSIX: FIFO con prioridades, “round-robin” con prioridades y servidor esporádico. Además de las tres políticas estándar, en MaRTE OS se define una cuarta política para permitir que las aplicaciones definan sus propios algoritmos de planificación. La presentación de esta última política, junto con la descripción de su implementación se hace en el capítulo 4, por lo que no será tratada en este apartado.

Independientemente de su política de planificación, todas las tareas conviven en la cola de tareas ejecutables, de forma que en todo momento la tarea a la cabeza de dicha cola es la que se encuentra posesión de la CPU. Resultará necesario por tanto, realizar un cambio de contexto cada vez que cambie la cabeza de la cola debido, bien a que la tarea en ejecución se ha suspendido o bloqueado, o bien a que una nueva tarea de mayor prioridad ha sido activada. Lo

mencionado hasta ahora constituye el soporte básico de la planificación de tareas con política FIFO con prioridades, pero para tareas de políticas “round-robin” o servidor esporádico también debe tenerse en cuenta el tiempo durante el que éstas se encuentran en ejecución, para de esta forma, tomar las decisiones de planificación definidas en sus políticas.

Ambas políticas imponen a sus tareas una limitación en el tiempo que éstas pueden mantenerse en ejecución de forma continuada. En el caso de las tareas “round-robin” esta limitación está marcada por la rodaja temporal o “quantum”, mientras para los servidores esporádicos el límite lo impone su capacidad de ejecución. Se ha optado por implementar dicha limitación mediante la utilización de un evento de tiempo de ejecución como los descritos en el apartado 3.6.3, “Eventos temporales”. De esta forma se simplifica la implementación del núcleo al unificar en gran medida el tratamiento de ambas políticas con la gestión de los eventos de tiempo de ejecución.

Una solución alternativa consistiría en utilizar un evento temporal normal que fuera incluido en la cola cada vez que una tarea de estas políticas tomara la CPU y eliminado de ella cuando la dejara. Aunque podría aplicarse la optimización de sólo añadir el evento cuando éste vaya a ocupar la cabeza de la cola, la expulsión de la tarea antes de concluir su plazo supondría el tener que eliminar su evento, con el consiguiente aumento del tiempo de peor caso de la operación de cambio de contexto. Por otra parte, la inclusión en el núcleo de estos eventos temporales de comportamiento especial complicaría su implementación.

Para las tareas planificadas bajo la política de servidor esporádico, es necesario programar también acciones de recarga de parte o toda su capacidad de ejecución. Puesto que estas acciones no están referidas al tiempo de ejecución de la tarea, sino a instantes absolutos de tiempo, pueden ser directamente implementadas como un tipo especial de evento temporal.

3.6.5. Señales

Como se expuso en el apartado 3.3, “Funcionalidad soportada”, se ha realizado una implementación limitada de las señales POSIX, en la que no se permite la suspensión dentro de los manejadores de señal. Con esta restricción, en lugar de ejecutar los manejadores de señal en el contexto de la tarea a la que la señal fue entregada, es posible ejecutarlos utilizando una única tarea especial a la máxima prioridad del sistema. Esto supone una importante simplificación, ya que, por ejemplo, cuando se entrega una señal a una tarea bloqueada en un mutex, no es necesario eliminarla temporalmente de la cola del mutex mientras que dure la ejecución del manejador.

Con la implementación elegida, cuando la acción asociada a una señal es ejecutar un manejador, ésta se encola en la cola ordenada de manejadores pendientes y se comprueba si la tarea manejadora se encuentra en ejecución. En caso de que no lo esté, la tarea es activada y, sin respetar el orden FIFO, pasa a ocupar la primera posición de la cola de tareas ejecutables. Su ejecución sólo puede ser interrumpida por los manejadores de interrupción. De ahí la necesidad de la cola de manejadores pendientes, ya que la gestión de una interrupción podría generar una nueva señal cuya acción asociada fuera ejecutar un manejador. Cuando la tarea manejadora finaliza la ejecución del manejador actual, comienza inmediatamente la ejecución del más prioritario de los que se encuentran encolados (en caso de haber alguno). En el estándar POSIX cada señal está representada por un número entero de 32 bits diferente, la ordenación de la cola de manejadores se realiza en base a dicho número, siendo un manejador tanto más prioritario cuanto menor sea el número de la señal que provocó su ejecución. El estándar POSIX únicamente requiere la ordenación por número de señal para las señales de tiempo real, no especificando ningún orden para el resto de las señales. Por consiguiente, no es contrario al

estándar la opción tomada en MaRTE OS de aplicar la citada ordenación a todas las señales, incluidas las que no son de tiempo real.

Con la implementación utilizada, la suspensión dentro de un manejador de señal implicaría la suspensión de la tarea manejadora, con la consiguiente imposibilidad de ejecutar otros manejadores. Esta situación no deseada puede ser detectada si el núcleo ha sido compilado con las opciones de depuración activas, como se discutirá en el apartado 3.6.8, “Depuración del núcleo”.

El estándar POSIX define un tipo especial de señales, denominadas señales de tiempo real. A diferencia de las señales tradicionales, las de este nuevo tipo pueden tener asociado un campo de información adicional, sucesivas ocurrencias de una señal pendiente no se pierden sino que se encolan y además el orden en que se entregan depende del número de la señal, siendo más prioritarias las de menor número. Puesto que MaRTE OS debe soportar este tipo de señales, por simplicidad se ha optado por que todas las señales, estén o no en el rango correspondiente a las de tiempo real, se comporten como ellas. Esto no es contrario al comportamiento descrito por el estándar POSIX para las señales tradicionales, ya que el estándar no especifica si deben encolarse ni determina ningún orden para su entrega, dejando ambas características a la elección de cada implementación particular.

Existe una excepción al comportamiento homogéneo anteriormente descrito que afecta a las señales generadas utilizando las funciones `kill()` y `pthread_kill()`. Ambas funciones fueron definidas en el estándar POSIX antes de la inclusión de las señales de tiempo real, por lo que, al contrario que su equivalente para señales de tiempo real `sigqueue()`, no contemplan la situación en que una señal no pueda ser enviada debido a que el sistema no dispone de suficientes recursos. Puesto que MaRTE OS está pensado para aplicaciones estáticas en las que el número de recursos, y en particular de instancias de señales, es limitado, se ha optado por prealojar una instancia para cada número de señal. De esta forma si la instancia correspondiente a la señal enviada está sin utilizar se pondrá como pendiente, y en caso de que ya lo esté las funciones `kill()` y `pthread_kill()` no harán nada. Este comportamiento es correcto ya que el estándar POSIX sólo exige que deban encolarse las señales generadas con `sigqueue()`.

3.6.6. Mutexes

Cuando se crea un mutex, se le asocia un bloque de control con campos que permiten identificar la tarea propietaria, el techo de prioridad del mutex y la lista de tareas en él bloqueadas. De igual manera, cada tarea mantiene una lista con todos los mutexes que se encuentran en su poder. El campo de techo de prioridad del mutex se utiliza tanto con los mutexes de protocolo de protección de prioridad, como con los de herencia. La diferencia es que para los primeros constituye un valor constante fijado en el momento de la creación de mutex, mientras que para los segundos, este campo toma en cada momento el valor correspondiente a la mayor prioridad de las tareas bloqueadas. En el caso de los mutexes de protocolo `PTHREAD_PRIO_NONE`, el citado campo toma el valor correspondiente a la menor prioridad del sistema. Con este comportamiento el cálculo la prioridad heredada por una tarea únicamente consiste en encontrar el mayor de los techos de los mutexes en la lista de mutexes en su poder.

El mecanismo descrito permite unificar en parte la gestión de los mutexes. Así, la toma de un mutex se realiza de igual manera independientemente de su protocolo: basta con apuntar la tarea propietaria, añadir el mutex a la lista de mutexes tomados por la tarea y subir la prioridad activa de ésta cuando sea preciso. También la liberación de un mutex constituye una operación independiente de su protocolo, que consiste en eliminar el mutex de la lista de mutexes en posesión de la tarea, actualizar la prioridad de ésta cuando sea necesario y encontrar la tarea más

prioritaria de las que se encuentran bloqueadas en el mutex para que lo tome, quedando libre en caso de que no haya ninguna.

Por su parte, las operaciones a realizar cuando una tarea se bloquea en un mutex sí que dependen del protocolo de éste. En todos los casos la tarea es añadida a la lista de tareas bloqueadas pero también, en el caso de que se trate de un mutex de herencia de prioridad, puede ser necesario elevar la prioridad activa de la tarea propietaria, que a su vez puede encontrarse bloqueada en otro mutex de herencia de prioridad y así sucesivamente hasta llegar a una tarea que no se encuentre bloqueada en uno de estos mutexes. Además, cuando una tarea se bloquea en un mutex de herencia de prioridad, este mutex se apunta en un campo de su bloque de control, lo que facilita la operación de herencia de prioridad en cadena descrita anteriormente.

A pesar de lo que podría parecer natural, ni la lista de tareas bloqueadas en un mutex ni la de mutexes en propiedad de una tarea han sido implementadas como colas ordenadas, aunque por supuesto el orden de prioridad es respetado, tal como exige el estándar POSIX, a la hora de elegir la próxima tarea a tomar un mutex.

Una de las razones para esta elección es que, cuando se utiliza el protocolo de protección de prioridad, y siempre que las tareas no se bloqueen mientras están en posesión de un mutex, no puede darse la circunstancia de que una tarea que desee tomar un mutex le encuentre tomado. Lo habitual es que se cumpla la citada condición de no suspensión, por lo que con este protocolo lo normal es que las colas en los mutexes no sean utilizadas, y en consecuencia, la utilización de colas no ordenadas no supone ninguna desventaja para el funcionamiento normal del sistema, mientras que simplifica la implementación. Sin embargo, hay que tener en cuenta que, aún en el caso de que sólo se utilizara este protocolo, las colas deben existir para la inusual circunstancia de que una tarea en posesión de un mutex se bloquee.

Las colas no ordenadas también presentan ventajas cuando se utiliza el protocolo de herencia de prioridad. Con este protocolo puede darse la situación de que el bloqueo de una tarea en un mutex suponga que varias tareas bloqueadas en cadena hereden su prioridad. Si las colas fueran ordenadas, esta situación supondría el reordenamiento de varias colas de mutexes con el correspondiente aumento del tiempo de ejecución de la operación. Si se asume que lo normal es que el número de tareas encoladas en un mutex no sea muy grande, el tiempo extra precisado en una cola no ordenada para encontrar la tarea más prioritaria se ve compensado por el ganado al evitar los reordenamientos en cadena antes mencionados.

Se ha optimizado el protocolo de protección de prioridad mediante la utilización del “cambio de prioridad diferido”. Con este mecanismo, cuando una tarea toma un mutex y por tanto hereda su techo de prioridad, no se eleva su prioridad inmediatamente, si no que simplemente se anota que el cambio está pendiente. En la mayoría de los casos, la duración de la sección crítica durante la cual la tarea tiene el mutex es muy corta, y por tanto la prioridad debe retornar a su nivel normal muy pronto, sin que ninguna otra tarea haya debido ser planificada en ese intervalo. En ese caso, lo único que hay que hacer es anular la anotación de cambio de prioridad pendiente. Si por el contrario fuera necesario hacer alguna acción de planificación mientras la tarea tiene el mutex, el planificador comprobaría si hay un cambio de prioridad pendiente para llevarle a cabo antes de realizar la citada acción.

El efecto de esta optimización es que, en la mayoría de los casos, se evita tener que realizar dos cambios de prioridad. Con ello se mejora muchísimo el tiempo promedio de toma y liberación de mutexes de protocolo de protección de prioridad, siendo similar a los tiempos obtenidos para los mutexes de herencia de prioridad.

3.6.7. Variables condicionales

Cada variable condicional necesita tener una cola con las tareas que se encuentran bloqueadas en ella. Si el número de tareas bloqueadas fuera inferior a cuatro, la implementación más eficiente sería una cola simplemente ordenada. Sin embargo, para un número mayor de tareas, una implementación como la elegida para la cola de tareas ejecutables es la más eficiente. Como se considera que el número de tareas bloqueadas en una variable condicional puede ser grande, se ha optado por esta última alternativa.

Una variable condicional se utiliza en asociación con un único mutex. El estándar POSIX dice que cuando una tarea se bloquea en una variable condicional, ésta pasa a estar ligada con el mutex utilizado en la llamada, de forma que constituye un error el que otra tarea trate, de forma concurrente, de bloquearse en la misma variable condicional usando un mutex distinto. Para detectar este error, cada variable condicional almacena un puntero al mutex con el que se encuentra ligada. A su vez, para cada mutex se lleva la cuenta de las variables condicionales que se encuentran ligadas con él, de forma que sólo se permite su eliminación si dicho contador es cero.

3.6.8. Depuración del núcleo

El núcleo de un sistema operativo debe constituir una pieza de código intensamente probada y altamente fiable. El objetivo es que el núcleo pueda ser considerado virtualmente libre de fallos por los programadores, de forma que cualquier error o funcionamiento incorrecto deba ser achacado al código de la aplicación. De no existir esta confianza en el código del núcleo, el proceso de desarrollo de aplicaciones se complicaría enormemente.

Este requisito de fiabilidad constituye una de las principales razones para la elección del Ada 95 como principal lenguaje de programación del núcleo. Como se expuso en el apartado 2.2, su tipificación estricta y otras características relacionadas permiten detectar muchos errores en tiempo de compilación, lo que aumenta enormemente la robustez del código generado con respecto al obtenido utilizando otros lenguajes de programación. Por otro lado, gran parte de los errores no detectados durante la fase de compilación, pueden ser detectados en tiempo de ejecución antes de que produzcan resultados catastróficos o impredecibles. Esto es posible gracias a los chequeos introducidos automáticamente por el compilador que provocan la generación de excepciones.

Por las razones expuestas en el apartado 3.4.1, en el núcleo no se utilizan las excepciones como mecanismo de notificación de errores, lo que supone que dentro del núcleo no existen manejadores para ellas ni se emplea la instrucción `raise`. Esto no significa que deba ser compilado forzosamente con los chequeos deshabilitados, pidiendo éstos ser habilitados durante la fase de desarrollo y prueba del núcleo, de forma que la información asociada con las excepciones (tipo de error y línea de código en la que se produjo) pueda ser utilizada en el proceso de depuración. Los chequeos podrán ser eliminados en las aplicaciones finales para mejorar las prestaciones del núcleo y reducir su tamaño.

Además de la posibilidad de habilitar las excepciones, en MaRTE OS puede activarse un conjunto extra de comprobaciones que permiten detectar la utilización de objetos inválidos o inactivos. Para ello, los objetos más importantes manejados por el núcleo (bloques de control de tareas, mutexes, variables condicionales, temporizadores, etc.) contienen como campo adicional un número entero de 32 bits. Este campo toma valores especiales (valores “mágicos”) dependiendo del estado del objeto: no utilizado, activo, finalizado, etc. De esta forma, resulta sencillo detectar la utilización de objetos con valor “mágico” inválido o válido pero inapropiado

para una determinada operación del núcleo. También mediante este conjunto de chequeos es posible detectar si una tarea ha sobrepasado el espacio dedicado a su pila, lo que constituye un error catastrófico y de muy difícil detección. Con este fin se escribe un valor especial al final de cada pila, de forma que su modificación implica que la tarea ha superado su límite. Las citadas comprobaciones han sido codificadas mediante la utilización de las directivas al compilador `Assert` y `Debug`, las cuales permiten la inclusión condicional de código en tiempo de compilación en función de que se utilice o no una opción del compilador.

Utilizando este mismo mecanismo de inclusión condicional de código, puede hacerse que MaRTE OS vuelque una traza de las operaciones que realiza sobre la consola del sistema o a través del puerto serie. También utilizando el puerto serie puede enviarse información más detallada de la actividad del núcleo (cambios de contexto, cambios de prioridad de las tareas, acciones de toma y liberación de mutexes, etc.) junto con los instantes en los que ocurren las acciones citadas. En base a esta información es posible representar en forma de cronograma la actividad del núcleo y de las tareas por él controladas.

Además de los mecanismos anteriormente descritos, que permiten la detección de errores internos y el análisis del funcionamiento del núcleo, también es posible depurar el código del núcleo y de las librerías utilizando un depurador igual al que se utilizaría para el código de las aplicaciones. El proceso de depuración se describe en el apartado 3.9, “Entorno de desarrollo de aplicaciones”.

3.7. Evaluación de la inclusión de nuevos servicios

Como comentamos en el apartado 2.1 de esta memoria, en la actualidad se encuentra en marcha el proceso de revisión del estándar POSIX.13. En dicho proceso se propone la incorporación de nuevos servicios al subconjunto mínimo de sistema de tiempo real. La implementación de algunos de ellos en MaRTE OS nos puede permitir obtener información real sobre el aumento de complejidad que puede suponer su inclusión en un sistema operativo conforme con el subconjunto mínimo.

Por considerar que serían los más útiles para las aplicaciones empotradas de tiempo real, los servicios elegidos para su implementación han sido el reloj monótono, la operación para suspensión absoluta de alta resolución, la política de planificación de servidor esporádico y los relojes y temporizadores de tiempo de ejecución. A continuación se procede a evaluar la complejidad añadida a MaRTE OS con la incorporación de los citados servicios.

Para la realización de todas las medidas mostradas en este apartado hemos utilizado un Pentium III a 1.1GHz.

3.7.1. Reloj monótono

Como mencionábamos en el apartado 3.6.3, anteriormente a la inclusión de esta funcionalidad POSIX, MaRTE OS ya utilizaba internamente un reloj de este tipo para la gestión de los eventos temporales. La forma de implementar este reloj varía dependiendo del procesador sobre el que ejecuta MaRTE OS. Como se expuso en el apartado 3.6.1, si el procesador es un Pentium se utiliza el “Time Stamp Counter” (TSC) como reloj del sistema. Este contador, que comienza a cero tras la inicialización del computador, constituye directamente el reloj monótono del sistema.

En procesadores 80386 y 80486, la hora del sistema se mantiene actualizada mediante la programación periódica del contador 0 del “Programmable Interval Timer” (PIT), de forma que

tras cada expiración del contador, el último intervalo programado se suma al tiempo total acumulado desde la inicialización del sistema. Con esta estrategia, el valor del reloj monótono en un instante dado se obtiene sumando el tiempo transcurrido desde la inicialización del sistema mas el valor actual del contador 0 del PIT.

En lo referente al reloj de tiempo real, su valor se obtiene de igual manera sea cual sea la arquitectura sobre la que ejecuta MaRTE OS: sumando al valor obtenido para el reloj monótono la hora de arranque del sistema (quizás modificada por los cambios de hora efectuados por la aplicación).

La posibilidad de utilizar el reloj monótono directamente desde las aplicaciones ha supuesto una modificación mínima en el núcleo, que sólamente afecta a las funciones que permiten operar con diferentes relojes, como son `clock_gettime()`, `clock_nanosleep()` y `pthread_cond_timedwait()`. Ahora en estas funciones se realiza la detección el reloj utilizado y, cuando se trata del reloj de tiempo real, se resta del valor temporal proporcionado la hora de arranque del sistema. De esta forma se obtiene un tiempo referido al reloj monótono que es el utilizado internamente para la gestión de los eventos temporales. El número de instrucciones introducidas es por tanto mínimo, teniendo un efecto muy poco importante sobre las prestaciones de las funciones afectadas. En todo caso la diferencia es siempre favorable a la utilización del reloj monótono por no requerirse entonces la adaptación del valor temporal facilitado a la función.

3.7.2. Operación de suspensión temporizada absoluta de alta resolución

La implementación de esta funcionalidad únicamente implica la incorporación en el núcleo de la función `clock_nanosleep()`, lo que no supone ningún impacto en las prestaciones o el tamaño de ningún otro servicio proporcionado por el sistema.

Tabla 3.5: Prestaciones de la función `clock_nanosleep()`.

Descripción medida	Tiempo (μs)
Suspensión mediante `nanosleep()`	0.89
Suspensión mediante `clock_nanosleep(CLOCK_REALTIME)` relativo	0.93
Suspensión mediante `clock_nanosleep(CLOCK_MONOTONIC)` absoluto	0.93
Suspensión mediante `clock_nanosleep(CLOCK_REALTIME)` absoluto	0.94
Activación tras la finalización de una operación de suspensión (`nanosleep()` o `clock_nanosleep()`)	0.75
Resolución de la función `nanosleep()`	1.0
Resolución de `clock_nanosleep(CLOCK_REALTIME)` relativo	1.0
Resolución de `clock_nanosleep()` absoluto	0.9

En la tabla 3.5 se evalúan las prestaciones de la función `clock_nanosleep()` (en sus diferentes modos de operación) y se comparan con las correspondientes a la función `nanosleep()`. Las medidas de las cuatro primeras filas corresponden al tiempo transcurrido desde que un thread de alta prioridad se suspende hasta que otro thread de baja prioridad, que se encontraba activo, comienza a ejecutar. Este es el único caso en el que existen diferencias

apreciables, aunque pequeñas, entre ambas funciones, debido al tiempo empleado en el tratamiento del mayor número de parámetros y modos de operación de la función `clock_nanosleep()`. La siguiente fila corresponde a la situación opuesta, en la que un thread de baja prioridad es expulsado de la CPU por un thread de alta prioridad que finaliza su operación de suspensión. Los tiempos de esta operación son iguales, independientemente de la operación de suspensión utilizada. Por su parte, en las tres últimas filas mostramos la diferencia entre el instante teórico en el que debería haberse producido la activación del thread y el instante real en que éste se encuentra en disposición de ejecutar. Como puede apreciarse las diferencias con las prestaciones de la función `nanosleep()` son mínimas.

3.7.3. Política de planificación de servidor esporádico

La política de planificación de servidor esporádico es relativamente compleja, aunque su dificultad de implementación se ha visto reducida en gran medida gracias a estar parcialmente integrada con los temporizadores de tiempo de ejecución y con la política de planificación "round robin". Así, para su implementación en el planificador de MaRTE OS se utilizan aproximadamente 60 líneas de código, de las cuales 20 son también utilizadas para la política "round robin" y/o los temporizadores de tiempo de ejecución. Además también es necesario incluir otras 30 líneas para la gestión de los parámetros de planificación propios de los threads planificados bajo está política.

Existe un parámetro de configuración que permite compilar el núcleo con el soporte para la política de servidor esporádico habilitado o deshabilitado. Esto ha permitido evaluar el impacto que supone la existencia de dicha política sobre los cambios de contexto entre threads de otras políticas. En la tabla 3.6 mostramos los tiempos de cambio de contexto entre dos tareas de política FIFO, en ella puede apreciarse que la habilitación de la política de planificación de servidor esporádico apenas afecta a los tiempos de cambio de contexto en los que no interviene ningún thread esporádico.

Tabla 3.6: Impacto de la política `SCHED_SPORADIC` en los tiempos de cambio de contexto de threads de política `SCHED_FIFO`

¿Habilitada la política de servidor esporádico?	Tiempo (μs)
NO	0.42
SI	0.44

Los cambios de contexto debidos a las reglas de la política de servidor esporádico que requieren la gestión de eventos temporales, como son los provocados por recargas o agotamientos de la capacidad de ejecución, son más costosos que los realizados entre threads FIFO. Sin embargo, como puede apreciarse en la tabla 3.7, el aumento en el tiempo de ejecución no resulta excesivo, resultando totalmente asumible, más si se tiene en cuenta que para un thread periódico de 1KHz de frecuencia, el tiempo de cambio de contexto nunca superaría el 0.13% de su periodo de ejecución.

En vista de la poca complejidad añadida y del impacto casi nulo sobre las prestaciones globales del sistema cuando no se utilizan threads de esta política, puede concluirse que merece la pena su implementación, más teniendo en cuenta las importantes ventajas que proporciona para la planificación de tareas aperiódicas.

Tabla 3.7: Tiempos de cambio de contexto en los que intervienen threads de política `SCHED_SPORADIC`

Thread que abandona el procesador	Thread que toma el procesador	Tiempo (μs)
Thread `SCHED_FIFO` invoca `sched_yield()`	Thread `SCHED_SPORADIC`	0.5
Thread `SCHED_SPORADIC` invoca `sched_yield()`	Thread `SCHED_FIFO`	0.45
Thread `SCHED_SPORADIC` invoca `sched_yield()`	Thread `SCHED_SPORADIC`	0.47
Thread `SCHED_SPORADIC` ha agotado su capacidad de ejecución	Thread `SCHED_FIFO`	1.2
Thread `SCHED_FIFO`	Thread `SCHED_SPORADIC` recarga su capacidad de ejecución	1.3

3.7.4. Relojes y temporizadores de tiempo de ejecución

La implementación de los relojes y temporizadores de tiempo de ejecución en el núcleo de MaRTE OS ha supuesto la inclusión de unas 10 líneas de código en la rutina de cambio de contexto (para medir el tiempo de ejecución consumido por las tareas) y de otras 50 líneas añadidas a la implementación de los relojes y temporizadores que utilizaban la hora global de sistema. Además también ha sido necesario añadir una lista de eventos temporales asociada a cada tarea, la cual debe ser tenida en cuenta cuando se desea encontrar el evento temporal más urgente con el que programar el temporizador hardware del sistema.

En MaRTE OS es posible, mediante un parámetro de configuración, compilar el núcleo de forma que soporte o no los temporizadores de tiempo de ejecución. En la tabla 3.8 se muestra el aumento que supone la habilitación de los temporizadores de tiempo de ejecución en los tiempos de cambio de contexto entre tareas. Cuando estos temporizadores se encuentran habilitados, el tiempo consumido aumenta ya que es necesario conocer la hora en que se produce el cambio de contexto para anotar el instante de activación de la tarea que toma la CPU y actualizar el tiempo de ejecución consumido por la tarea que la deja.

Tabla 3.8: Impacto de los temporizadores de tiempo de ejecución en los tiempos de cambio de contexto

¿Temporizadores de tiempo de ejecución habilitados?	Tiempo (ns)
NO	0.42
SI	0.44

La implementación en MaRTE OS de los relojes y temporizadores de tiempo de ejecución no ha supuesto un aumento excesivo en la complejidad del núcleo. Tampoco supone un impacto resaltable sobre los tiempos de cambio de contexto, al menos en arquitecturas modernas, en las que la lectura del reloj es muy rápida (como ocurre en los procesadores Pentium cuando se utiliza el “Time Stamp Counter”, tal y como se expuso en el apartado 3.6.1, “Interfaz abstracta con el hardware en PCs”). La utilización de estos relojes y temporizadores permite detectar

sobrepasos en los tiempos de ejecución de peor caso estimados para las tareas, lo que constituye un error de fatales consecuencias y de dificil detección por otros medios. Esta importante ventaja compensa de sobra el pequeño aumento en los tiempos de cambio de contexto que sufrirían las aplicaciones que requiriesen su utilización.

3.8. Prestaciones y tamaño de las aplicaciones

3.8.1. Prestaciones

Un factor importante en un sistema operativo que pretende ser utilizado en aplicaciones empotradas de tiempo real son sus prestaciones. Cuanto menos tiempo emplee el sistema operativo en realizar su función, de más tiempo dispondrá la aplicación, con lo que será posible la implementación de algoritmos más complejos. La minimización de la influencia de los tiempos de ejecución del sistema operativo, aunque buscada en todos los entornos computacionales, resulta más importante si cabe en los sistemas empotrados, puesto que muchos de ellos disponen de una capacidad de ejecución limitada por razones de tamaño, peso, consumo o coste.

Para proporcionar una idea general de la sobrecarga impuesta por MaRTE OS a las aplicaciones, en este apartado mostraremos los tiempos de ejecución correspondientes a las operaciones del núcleo que hemos considerado más representativas. Los valores mostrados han sido obtenidos con la versión 1.1[1] de MaRTE OS y utilizando un computador con procesador Pentium III a 1.1GHz.

En la tabla 3.9 se muestran los tiempos empleados por MaRTE OS para la realización de cambios de contexto, tanto entre threads POSIX de política FIFO con prioridades como entre tareas Ada. Los valores de las dos primeras filas corresponden a la situación más sencilla en la que el thread o la tarea ceden el procesador de forma voluntaria a otro thread o tarea de su misma prioridad. La diferencia entre ambos tiempos se debe a la librería de tiempo de ejecución del compilador GNAT. Esta capa extra de software, que proporciona el elegante y potente soporte multitarea del lenguaje Ada, presenta como contrapartida un aumento de los tiempos de ejecución que se verá reflejado en todas las medidas presentadas en este apartado.

Tabla 3.9: Tiempos de cambio de contexto.

Descripción medida	Tiempo (μs)
Cambio de contexto entre threads C después de una operación `sched_yield()`	0.44
Cambio de contexto entre tareas Ada después de una instrucción `delay 0.0`	1.5
Rendezvous entre tareas Ada (incluye dos cambios de contexto)	3.9
Paso de un entero desde una tarea productora a otra consumidora a través de una tarea intermedia (incluye dos “rendezvous”)	9.3

1. Se trata de una versión interna similar a la última versión públicamente disponible a fecha de escritura de esta memoria (Versión 1.0) salvo por que incluye la implementación de servicios como el reloj monótono y la suspensión temporizada absoluta de alta resolución.

Los últimos dos valores de la tabla 3.9 corresponden a cambios de contexto en los que entran en juego otros elementos de la semántica de tareas Ada. Así, el tercer dato corresponde a un "rendezvous" mínimo entre tareas Ada en el que no se transmite ningún mensaje. Esta operación supone la realización de dos cambios de contexto. El cuarto de los valores corresponde al paso de un dato entero desde una tarea productora hasta otra consumidora, utilizando como mecanismo de almacenamiento intermedio una tercera tarea. Por consiguiente, esta medida incluye el tiempo correspondiente a dos "rendezvous" en los que se transmite información y a las operaciones de sincronización realizadas en la tarea intermedia, que permiten bloquear la tarea productora hasta que el dato haya sido consumido y la tarea consumidora hasta que haya un dato disponible.

En la tabla 3.10 aparecen algunos tiempos relativos a la utilización de mutexes y variables condicionales. Los dos primeros valores corresponden al tiempo de toma y liberación de un mutex de protocolo de protección de prioridad dependiendo de si se aplica o no el cambio de prioridad diferido que fue comentado en el apartado 3.6.6. Se puede apreciar como esta optimización mejora de forma sustancial el tiempo de toma y liberación del recurso al ahorrar el cambio de prioridad del thread. El tercero de los valores corresponde al tiempo de activación de un thread que se encuentra bloqueado en una variable condicional, esto es, el tiempo transcurrido entre que un thread de baja prioridad señaliza la variable y el thread de alta prioridad que estaba bloqueado en ella comienza a ejecutar con el mutex en su poder.

Tabla 3.10: Mutexes y Variables Condicionales.

Descripción medida	Tiempo (μs)
Toma y liberación de mutex (con cambio de prioridad diferido)	0.34
Toma y liberación de mutex (realizando el cambio de prioridad)	0.53
Activación de thread esperando en `pthread_cond_wait()`	0.75

Los tiempos mostrados en la tabla 3.11 corresponden a la duración de una operación de lectura del reloj tanto desde programas C, utilizando la función POSIX `clock_gettime()`, como desde programas Ada, mediante la función `Clock` del paquete estándar `Calendar`. Como se aprecia en los tiempos de las dos primeras filas, no existe diferencia apreciable entre la operación de lectura del reloj de tiempo real o del monótono utilizando `clock_gettime()`.

Tabla 3.11: Lectura del reloj.

Descripción medida	Tiempo (μs)
`clock_gettime (CLOCK_REALTIME)`	0.24
`clock_gettime (CLOCK_MONOTONIC)`	0.24
`Calendar.Clock`	0.6

La tabla 3.12 muestra algunas medidas relativas a la utilización de señales y temporizadores POSIX. Se proporcionan valores para las dos funciones definidas en el estándar para la espera de señales: `sigwait()` y `sigwaitinfo()`, esta última permite esperar a una señal y obtener la información adicional que acarrea. Contrariamente a lo que podría pensarse, los tiempos correspondientes a `sigwaitinfo()` son iguales o incluso menores que los obtenidos para `sigwait()` a pesar de su teórica mayor complejidad. Esto es debido a que, con el objeto de

simplificar la implementación y pensando que la utilización de señales con información asociada será muy frecuente, el núcleo de MaRTE OS únicamente implementa `sigwaitinfo()`. Por su parte, `sigwait()` se implementa como una función de envoltura de la anterior, en la que se descarta el campo de información asociado con la señal.

Tabla 3.12: Señales y temporizadores.

Descripción medida	Tiempo (μs)
Suspensión de un thread a la espera de señales con `sigwait()`	0.93
Suspensión de un thread a la espera de señales con `sigwaitinfo()`	0.82
Activación de un thread esperando una señal mediante `sigwait()`	0.54
Activación de un thread esperando una señal mediante `sigwaitinfo()`	0.54
Activación de un thread esperando una señal mediante `sigwaitinfo()` como consecuencia de la expiración de un temporizador de tiempo de ejecución	0.9

Los dos primeros valores de la tabla 3.12 corresponden al tiempo transcurrido entre que un thread de alta prioridad se suspende a la espera de un conjunto de señales y un thread de baja prioridad que se encontraba activo retoma su ejecución. Por su parte, las dos filas siguientes representan la situación opuesta, en la que un thread de baja prioridad envía una señal mediante la función `pthread_kill()` y un thread de alta prioridad que está esperando esa señal se pone en ejecución. El último de los valores corresponde a un cambio de contexto provocado por la generación de una señal como consecuencia de la expiración de un temporizador de tiempo de ejecución, esto es, el tiempo empleado en la activación del thread de alta prioridad que se encontraba esperando la señal mediante la función `sigwaitinfo()`. Es importante resaltar que todas las medidas mostradas en la tabla incluyen un cambio de contexto.

Tabla 3.13: Suspensión temporizada de threads C.

Descripción medida	Tiempo (μs)
Suspensión mediante `nanosleep()`	0.89
Suspensión mediante `clock_nanosleep(CLOCK_REALTIME)` relativo	0.93
Suspensión mediante `clock_nanosleep(CLOCK_MONOTONIC)` absoluto	0.93
Suspensión mediante `clock_nanosleep(CLOCK_REALTIME)` absoluto	0.94
Activación tras la finalización de una operación de suspensión (`nanosleep()` o `clock_nanosleep()`)	0.75
Resolución de la función `nanosleep()` y de `clock_nanosleep(CLOCK_REALTIME)` relativo	1.0
Resolución de `clock_nanosleep(CLOCK_MONOTONIC)` absoluto y de `clock_nanosleep(CLOCK_REALTIME)` absoluto	0.9

En la tabla 3.13 mostramos las medidas relacionadas con la suspensión temporizada de threads utilizando las funciones `nanosleep()` y `clock_nanosleep()`. En el caso de esta última

función hemos realizado medidas utilizado los relojes monótono y de tiempo real tanto para suspensiones absolutas como relativas. Las medidas de las cuatro primeras filas corresponden al tiempo transcurrido desde que un thread de alta prioridad se suspende hasta que otro thread de baja prioridad, que se encontraba activo, comienza a ejecutarse. La siguiente fila corresponde a la situación opuesta, en la que un thread de baja prioridad es expulsado de la CPU por un thread de alta prioridad que finaliza su operación de suspensión. Los tiempos de esta operación son iguales, independientemente de la función utilizada. Por su parte, en las dos últimas filas mostramos la diferencia entre el instante teórico en el que debería haberse producido la activación del thread y el instante real en que éste se encuentra en disposición de ejecutar.

Tabla 3.14: Suspensión temporizada de tareas Ada.

Descripción medida	Tiempo (μs)
Instrucción `delay` y cambio de contexto	2.1
Instrucción `delay until` y cambio de contexto	2.2
Activación tras la finalización de una instrucción `delay` (incluye un cambio de contexto)	1.9
Activación tras la finalización de una instrucción `delay` (incluye un cambio de contexto)	1.9
Resolución de la instrucción `delay`	2.2
Resolución de la instrucción `delay until`	2.1

La tabla 3.14 muestra medidas equivalentes a las que aparecían en la tabla 3.13, pero en este caso para las tareas Ada y sus instrucciones propias de suspensión relativa (`delay`) y absoluta (`delay until`).

Tabla 3.15: Comparación de MaRTE OS con sistemas operativos de tiempo real comerciales.

Descripción medida	Sistema Operativo			
	QNX Neutrino V6.2	VxWorks AE 1.1	Windows CE .NET	MaRTE OS V1.1
Cambio de contexto después de una operación de cesión voluntaria de la CPU	2.4 μs	3.4 μs	3.3 μs	2.4 μs
Liberación y posterior toma de un mutex de herencia de prioridad	11.9 μs	9.3 μs	19.4 μs	3.5 μs
Cambio de contexto provocado por una interrupción	3.9 μs	4.9 μs	5.6 μs	5.5 μs

Finalmente, la tabla 3.15 permite comparar las prestaciones de MaRTE OS con las de algunos de los sistemas operativos de tiempo real para sistemas empotrados más utilizados en la

actualidad, como son QNX Neutrino V6.2, Windows CE .NET y VxWorks AE 1.1. Los datos han sido obtenidos de evaluaciones de esos sistemas realizadas por “Dedicated Systems Experts” [DSE02A] [DSE02B] [DSE02C] sobre un computador con procesador Pentium MMX a 200MHz. Por su parte, los datos obtenidos para MaRTE y mostrados en la última columna de la tabla, han sido medidos sobre un computador de similares características y tratando de repetir los tests descritos en los citados documentos de evaluación.

Puesto que el entorno de test no es el mismo para MaRTE OS que para los demás sistemas operativos, los resultados no pueden utilizarse más que de una forma orientativa. Aún así, creemos que permiten llegar a la conclusión de que las prestaciones de MaRTE OS son al menos comparables a las de esos sistemas, lo que desde nuestro punto de vista constituye ya un hecho más que satisfactorio, sobre todo si se tiene en cuenta los años de desarrollo y el nivel de refinamiento que pueden haber alcanzado estos sistemas operativos comerciales.

3.8.2. Tamaño de las aplicaciones

Además de las prestaciones, otro factor muy importante en los sistemas operativos para aplicaciones empotradas es su tamaño. Lo normal es que los computadores empotrados dispongan de algún dispositivo de memoria no volátil para almacenar de forma permanente el programa de usuario junto con el sistema operativo. La cantidad disponible de este tipo de memoria se ve limitada en estos sistemas por razones de tamaño, peso, consumo o coste, lo que obliga a reducir al máximo el tamaño de la aplicación empotrada.

Tabla 3.16: Tamaño de las aplicaciones.

Descripción de la aplicación	**Opciones de compilación de MaRTE OS**	
	Optimización para máxima velocidad (`-gnatn -gnatp -O3`)	**Optimización para mínimo tamaño (`-gnatp -O2`)**
Dos threads comparten datos mediante un mutex y una variable condicional	311 Kb	180 Kb
Nueve threads. Utiliza señales, suspensión temporizada, mutexes y variables condicionales.	318 Kb	188 Kb
Dos tareas Ada comparten datos mediante un objeto protegido	379 Kb	235 Kb
Cuatro tareas Ada y dos objetos protegidos. Utiliza las señales y los temporizadores del POSIX.5b.	489 Kb	357 Kb

En la tabla 3.16 mostramos el tamaño de algunas aplicaciones para MaRTE OS. Los valores mostrados corresponden a la aplicación final, que incluye tanto el sistema operativo como el programa de usuario. Los tamaños corresponden a la suma de los valores indicados por la orden `size` para los segmentos de texto y datos El tamaño de las aplicaciones depende de las opciones con las que ha sido compilado el núcleo de MaRTE OS. En la tabla se muestran dos tamaños

para cada aplicación, uno obtenido después de haber compilado el núcleo con las opciones que permiten optimizar su velocidad: `-gnatn` (inclusión en línea de subprogramas), `-gnatp` (no excepciones) y `-O3` (máximo nivel de optimización) y otro cuando el núcleo ha sido compilado para ocupar el mínimo tamaño: no se permite la inclusión en línea de subprogramas y se utiliza un nivel de optimización menor (`-O2`).

Como se aprecia en la tabla, el tamaños de las aplicaciones Ada son mayores que los de las aplicaciones C. Esto es debido a la inclusión en las primeras de parte de la librería de tiempo de ejecución del compilador GNAT. La última fila de la tabla corresponde a una aplicación relativamente compleja que utiliza una parte importante de la librería de tiempo de ejecución y, además, gran parte de la interfaz POSIX.5b. Esa es la razón de que su tamaño sea bastante mayor que el de las demás.

Los valores obtenidos son grandes si se comparan con sistemas operativos para sistemas empotrados pequeños, cuyos tamaños suelen ser de unas decenas de kilobytes. Sin embargo, esta comparación no resulta apropiada puesto que esos núcleos proporcionan una funcionalidad muy inferior a la soportada por MaRTE OS. Los resultados se encuentran más cercanos a los tamaños de sistemas operativos de tiempo real con interfaz POSIX, en los que sólo el núcleo suele variar entre los 100 y los 200 kilobytes.

Pensamos que son varios los factores por los que el tamaño de MaRTE OS es algo mayor que el de otros sistemas operativos de similares características. Por un lado, el tamaño del código generado utilizando el lenguaje de programación Ada es algo mayor que el generado utilizando el lenguaje C. Otro factor se deriva de la filosofía de organización del código en paquetes, ya que la utilización de una sola entidad de las definidas en un paquete implica la inclusión del paquete completo en la aplicación final. Esta circunstancia limita la escalabilidad de las aplicaciones, de forma que programas que hacen uso de un pequeño subconjunto de la funcionalidad definida en MaRTE OS no ven reducido su tamaño en proporción a los servicios utilizados. Por este mismo motivo en cualquier aplicación por sencilla que sea, siempre se incluye una parte importante de la librería de tiempo de ejecución del compilador GNAT, más si se tiene en cuenta que la escalabilidad no fue considerada un objetivo fundamental en el diseño de la citada librería. Por otra parte, hay que tener en cuenta que la interfaz POSIX-Ada supone un aumento extra de la complejidad y del tamaño de MaRTE OS frente a otros sistemas operativos que no proporcionan dicha interfaz.

3.8.3. Complejidad del núcleo

Por último, para dar una idea de la complejidad del núcleo, la tabla 3.17 muestra el número de líneas de código (ocurrencias del carácter “punto y coma”), paquetes Ada, ficheros C y ficheros ensamblador que componen MaRTE OS. Como se aprecia en la primera fila, el núcleo de MaRTE OS está formado por 89 paquetes Ada (unos 160 ficheros entre especificaciones y cuerpos), 68 ficheros de código C y 9 de ensamblador. La primacía del lenguaje Ada es más clara si nos fijamos en el número de líneas de código: la parte del núcleo escrita en Ada contiene unas 9141 líneas de código, aproximadamente el 86% de las 10636 líneas de código que componen el núcleo. En las líneas siguientes se muestran los datos relativos a otras partes del sistema operativo, como son las interfaces POSIX en sus versiones Ada y C y la librería estándar C.

Tabla 3.17: Número de ficheros y líneas de código de MaRTE OS.

	Paquetes Ada	Ficheros C	Ficheros Asm	TOTAL
Núcleo	89 paquetes 9141 líneas	68 ficheros 1426 líneas	9 ficheros 69 líneas	166 paquetes y fich. 10636 líneas
Interfaz POSIX-Ada	18 paquetes 2092 líneas	-	-	18 paquetes 2092 líneas
Interfaz POSIX-C (ficheros de cabecera)	-	28 ficheros 615 líneas	-	28 ficheros 615 líneas
Librería C estándar (`libc`)	-	75 ficheros 2443 líneas	-	75 ficheros 2443 líneas
TOTAL	107 paquetes 11233 líneas	171 ficheros 4484 líneas	9 ficheros 69 líneas	287 paquetes y fich. 15786 líneas

3.9. Entorno de desarrollo de aplicaciones

Lo habitual en los sistemas empotrados es que el desarrollo de las aplicaciones se realice en un entorno cruzado formado por dos computadores: el computador empotrado sobre el que se ejecutarán las aplicaciones y el equipo de desarrollo, generalmente más potente y con un sistema operativo de propósito general, en el que se dispone de un conjunto de herramientas que posibilitan la creación, carga en el equipo de ejecución y depuración de las aplicaciones.

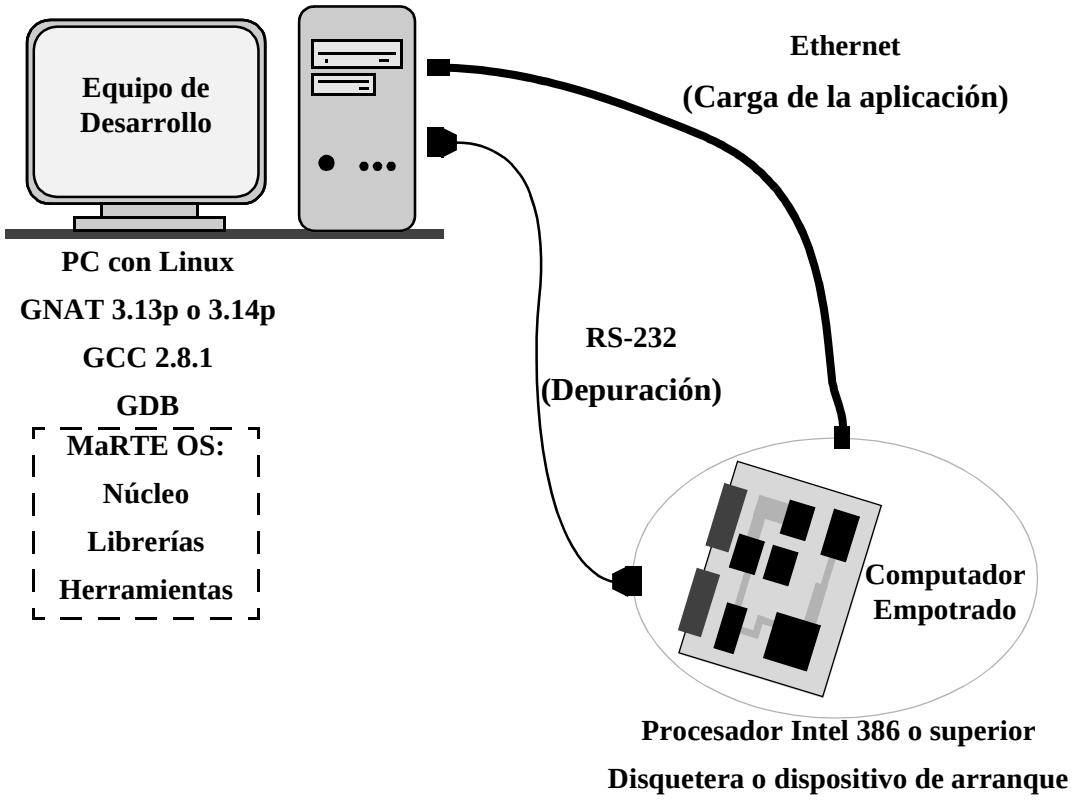

Figura 3.9: Entorno de desarrollo de MaRTE OS

En la figura 3.9 se muestra el entorno de desarrollo de MaRTE OS cuando se utiliza como computador empotrado un PC. El equipo de desarrollo es también un PC con sistema operativo Linux en el que se encuentran instalados los compiladores GNAT (versiones 3.13p o 3.14p) y GCC (versión 2.8.1). Lo normal en un entorno de desarrollo cruzado sería que se utilizasen las versiones cruzadas de ambos compiladores, aunque en este caso particular, puesto que el computador empotrado y el de desarrollo tienen la misma arquitectura, se utiliza la versión nativa. También en el equipo de desarrollo se encuentra el núcleo y el resto de librerías que componen el sistema operativo MaRTE OS, junto con un conjunto de archivos de órdenes escritos en lenguaje Perl [PER96] que automatizan la creación, carga y depuración de las aplicaciones.

El proceso de creación de una aplicación consiste en la compilación de su código y su posterior enlazado con las librerías que componen MaRTE OS. Los archivos de órdenes `mgcc` y `mgnatmake` se encargan de automatizar este proceso mediante la invocación de los compiladores GNAT y GCC con las opciones apropiadas. Además realizan la copia de la aplicación generada al directorio desde el que será cargada en el computador empotrado. Su modo de utilización es similar al de los comandos `gcc` y `gnatmake`, aceptando prácticamente las mismas opciones que éstos.

Así por ejemplo, la compilación del programa C `hola_mundo.c`, su enlazado con las librerías de MaRTE OS y la generación de la aplicación final lista para ser ejecutada en el computador empotrado, se realizaría con la orden:

```
$ mgcc hola_mundo.c
```

Como hemos dicho, también es posible utilizar opciones de compilación. Por ejemplo, la compilación del programa C `mi_programa.c` con las opciones de depuración (`-g`) y optimización de nivel dos (`-O2`) y su posterior enlazado con las librerías de MaRTE OS y con el fichero de código objeto `extra_obj.o` se realizaría mediante la orden:

```
$ mgcc -g -O2 extra_obj.o mi_programa.c
```

La compilación y enlazado de un programa Ada cuyo procedimiento principal esta en el fichero `mi_programa.adb`, utilizando la opción `-gnato` (detección de errores de sobrepaso de rango), se realiza mediante la orden:

```
$ mgnatmake -gnato mi_programa.adb
```

Una opción que no puede ser utilizada con `mgcc` ni con `mgnatmake` es "`-o`". La utilización de esta opción con los comandos `gcc` o `gnatmake` permite indicar el nombre del ejecutable. Esto no es posible en nuestro entorno, ya que la aplicación debe ser generada con un nombre y en un directorio prefijados desde donde será cargada en el computador empotrado tal como explicaremos a continuación. Otra pequeña limitación es que, a diferencia de la orden `gnatmake`, con `mgnatmake` no es posible omitir la extensión (`.adb`) del fichero que contiene el procedimiento principal.

La aplicación generada en el equipo de desarrollo se descarga desde el computador empotrado utilizando un programa que permite la carga de aplicaciones a través de la red. Dicho programa está contenido en un disquete automáticamente generado durante la instalación de MaRTE OS (aunque de igual manera podría estar contenido en otro tipo de dispositivo de arranque como un disco duro o una "Flash RAM"). Para el entorno de la figura 3.9, el ciclo de desarrollo de una aplicación es el siguiente:

1. El computador empotrado se arranca con el disquete de carga de aplicaciones por red.
2. La aplicación se compila y enlaza en el equipo de desarrollo.
3. El programa de carga por red descarga la aplicación desde equipo de desarrollo y la pone en ejecución en el computador empotrado.
4. La aplicación ejecuta libremente o es depurada remotamente desde el equipo de desarrollo.
5. Tan pronto como la aplicación finaliza, el programa de carga toma de nuevo el control sobre el computador empotrado y un nuevo ciclo puede comenzar en el paso 2.

El método de carga de la aplicación utilizando la red es el apropiado para la etapa de desarrollo de la aplicación, pero si se desea, la aplicación puede ser cargada directamente desde un disquete u otro medio similar sin la necesidad del computador de desarrollo. Esta opción puede ser utilizada para realizar la carga de la aplicación en el computador empotrado cuando se halle en su ubicación final, en la cual es muy probable que se encuentre totalmente desconectado de cualquier otro equipo

La depuración de aplicaciones y del núcleo se realiza desde el equipo de desarrollo utilizando el modo de depuración remoto a través del puerto serie del depurador GDB de GNU [GDB98]. Para que la comunicación entre el depurador y la aplicación sea posible, es necesario configurar correctamente el puerto mediante una llamada a la operación correspondiente por parte de la aplicación. Además, con el objeto de que el GDB pueda tomar el control de la ejecución, es necesario también incluir en el código de la aplicación un punto de ruptura. MaRTE OS proporciona ambas operaciones en el paquete `Debug_Marte` y en el fichero de cabecera `<debug_marte.h>`. A continuación mostramos un ejemplo de un programa Ada preparado para depuración:

```
with ...
with Debug_Marte;
...
procedure Mi_Programa is
begin
  ...
  Debug_Marte.Init_Serial_Communication_With_Gdb
                                        (Debug_Marte.Serial_Port_1);
  Debug_Marte.Set_Break_Point_Here;

  Código de la aplicación ...;
end Mi_Programa;
```

Por su parte, un programa C preparado para depuración tendría la estructura siguiente:

```
#include ...
#include <debug_marte.h>
...
int main()
{
  ...
  init_serial_communication_with_gdb (SERIAL_PORT_1);
  set_break_point_here;

  Código de la aplicación;
}
```

Una vez que la aplicación alcanza un punto de ruptura (instalado mediante `Debug_Marte.Set_Break_Point_Here` o `set_break_point_here`), en el computador de desarrollo se puede comenzar una sesión de depuración remota mediante el siguiente comando del GDB:

```
(gdb) target remote /dev/ttyS0
```

Donde `/dev/ttyS0` indica el puerto serie utilizado para la conexión (en este caso el `com1`). A partir de este momento es posible llevar a cabo una sesión de depuración normal, pudiéndose instalar nuevos puntos de ruptura, ejecutar instrucción a instrucción, observar el valor de las variables y todas las operaciones habituales de los depuradores de alto nivel.

Además de las herramientas de compilación de aplicaciones, con MaRTE OS se proporcionan archivos de comandos que automatizan la compilación del núcleo y demás librerías que componen el sistema operativo. Mediante dichas herramientas es posible recompilar el sistema tras haber cambiado alguno de sus parámetros configurables o después de haber realizado alguna modificación o añadido nueva funcionalidad. Las herramientas facilitadas son:

- `mkkernel`: compila el núcleo de MaRTE OS. Acepta opciones de `gnatmake` y/o `gcc`. Un ejemplo de utilización sería:
  ```
  $ mkkernel -gnatn -O3 -gnatp
  ```
- `mklibmc`: compila la librería estándar C aceptando opciones de `gcc`. Así por ejemplo, la compilación de la librería con el máximo nivel de optimización se realizaría mediante la orden
  ```
  $ mklibmc -O3
  ```
- `mkall`: realiza las mismas operaciones `mkkernel` y `mklibmc` juntos pero forzando la recompilación de todos los paquetes Ada. La compilación total del sistema optimizado para obtener la máximas prestaciones se realizaría mediante la orden:
  ```
  $ mkall -gnatn -O3 -gnatp
  ```

3.10. Conclusiones

Se ha desarrollado el sistema operativo MaRTE OS, un núcleo de tiempo real para sistemas empotrados conforme con el perfil de sistema de tiempo real mínimo definido en el estándar POSIX.13. El núcleo proporciona las interfaces POSIX para los lenguajes Ada y C, y además, en el caso del lenguaje Ada se ha adaptado la librería de tiempo de ejecución de compilador GNAT, lo que permite a las aplicaciones Ada ejecutando sobre MaRTE OS utilizar toda la semántica de tareas proporcionada por este lenguaje.

Con la funcionalidad proporcionada, sobre MaRTE OS es posible ejecutar aplicaciones empotradas concurrentes escritas en Ada o C (así como aquellas en las que conviven tareas Ada y threads C). Las restricciones impuestas a dichas aplicaciones son las descritas para el perfil de sistema de tiempo real mínimo, siendo la más importante de ellas la inexistencia de sistema de ficheros.

Junto con el sistema operativo se ha desarrollado un entorno de desarrollo cruzado basado en Linux, en los compiladores GCC y GNAT y en un conjunto de archivos de órdenes que automatizan el proceso de creación, carga y depuración de las aplicaciones.

A fecha de escritura de esta memoria, se encuentra disponible en Internet (`http://marte.unican.es`) la versión 1.0 de MaRTE OS para computadores personales (PC) con

procesador Intel 80386 y superiores. Dicha versión se distribuye como software libre bajo licencia “GNU General Public License” (GPL).

A la vista de los resultados mostrados en el apartado 3.8 referentes a prestaciones y tamaño de los ejecutables, puede concluirse que es posible implementar un núcleo de sistema operativo conforme al perfil de sistema de tiempo real mínimo definido en el estándar POSIX.13 que sea apropiado para sistemas empotrados, lo que constituía el principal objetivo de este perfil. De esos mismos resultados se deduce que es posible implementar un sistema operativo de esas características utilizando el lenguaje de programación Ada 95, lo que supone un hecho relevante si se tiene en cuenta que este lenguaje supera en muchos aspectos (mantenibilidad, fiabilidad, etc.) al C, que es el lenguaje más ampliamente utilizado para la escritura de sistemas operativos.

Otro importante resultado obtenido tiene que ver con la implementación en MaRTE OS de algunos de los nuevos servicios que, dentro del actual proceso de revisión del estándar POSIX.13, han sido propuestos para su incorporación en el perfil mínimo de sistema de tiempo real. Los servicios elegidos han sido el reloj monótono, la operación para suspensión absoluta de alta resolución, la política de planificación de servidor esporádico y los relojes y temporizadores de tiempo de ejecución. Esto nos ha permitido obtener información sobre el impacto que su incorporación supone en el tamaño, complejidad y prestaciones de un núcleo conforme con el subconjunto mínimo.

Tanto en el caso de la implementación del reloj monótono, como en el de la operación de suspensión absoluta de alta resolución, el impacto es mínimo. Por su parte la política de planificación de servidor esporádico y los relojes y temporizadores de tiempo de ejecución suponen un aumento apreciable aunque no excesivo de la complejidad del núcleo, el cual creemos totalmente asumible comparado con el importante aumento de funcionalidad que supone su incorporación.

A fecha de escritura de esta memoria, MaRTE OS constituye un producto terminado y totalmente operativo que puede ser utilizado:

- Para desarrollar aplicaciones industriales de tiempo real.
- Como herramienta para la docencia en sistemas operativos de tiempo real y sistemas empotrados.
- Como plataforma de investigación sobre la que probar nuevos servicios facilitados por los sistemas operativos.

Como herramienta docente, MaRTE OS está siendo utilizado en el grupo de Computadores y Tiempo Real de la Universidad de Cantabria y en otras universidades de España y del resto del mundo como base para laboratorios de tiempo real y desarrollo de aplicaciones empotradas. También está siendo utilizado en ésta y otras universidades para la realización de proyectos fin de carrera. Los proyectos en los que el autor de esta tesis colabora en su dirección y que a fecha de escritura de esta memoria han sido finalizados o se encuentran próximos a su finalización son:

- Diseño e implementación del control de un péndulo invertido sobre MaRTE OS.
- Desarrollo del entorno de programación e instalación de manejadores de dispositivos (drivers) en MaRTE OS.
- Portado del núcleo de MaRTE OS a un robot basado en el microcontrolador MC68332.

Como herramienta de investigación, la arquitectura modular de MaRTE OS junto con las características del lenguaje Ada hacen que sea sencilla la incorporación de nuevas funcionalidades. Esto se ha comprobado con la implementación de las interfaces presentadas en

los capítulos 4, “Interfaz de usuario para la definición de algoritmos de planificación” y 5, “Interfaz de usuario para la definición de protocolos de sincronización”.

Con el mismo objetivo de implementar nuevas funcionalidades proporcionadas por los sistemas operativos, MaRTE OS ha sido utilizado en el Departamento de Sistemas Informáticos y Computación de la Universidad Politécnica de Valencia para implementar el recién aprobado estándar POSIX.1q (“Trace”) [PSX00q] y desarrollar un “binding” para Ada de ese estándar [ESP02].

Son numerosos los departamentos de universidades de España y del resto del mundo que están utilizando MaRTE OS en proyectos de docencia, investigación y desarrollo. Entre ellos cabría citar: “Departamento Sistemas Informáticos y Computación”, “Departamento Informática de sistemas y computadores” y “Departamento Ingeniería de sistemas y automática” (Universidad Politécnica de Valencia), “Departamento de Ciencias Matemáticas e Informática” (Universidad de las Islas Baleares), “Departamento de Lenguajes y Ciencias de la Computación” (Universidad de Málaga), “Departamento de Ingeniería Telemática” (Universidad de Vigo), “Instituto Universitario de Microelectrónica Aplicada” (Universidad de Las Palmas de Gran Canaria), “Department of Computer Science” (University of York), “Department of Computer Science” (University of Northern Iowa) y “École d'ingénieurs de Genève” (Université de Genève).

4. Interfaz de usuario para la definición de algoritmos de planificación

4.1. Introducción

En la introducción de esta memoria se planteó que los algoritmos de planificación actualmente soportados por sistemas operativos, generalmente basados en prioridades fijas, no son suficientes para satisfacer los requerimientos temporales de todo tipo de aplicaciones. Ante la gran variedad de algoritmos de planificación existentes, cada uno de ellos apropiado para satisfacer un tipo de requisitos temporales, se llegaba a la conclusión de que no sería razonable buscar la solución del problema por la vía de la estandarización e inclusión de todos ellos en los sistemas operativos. En lugar de eso, se proponía que fueran las propias aplicaciones las que definieran los algoritmos bajo los cuales querían planificar sus tareas, debiendo el sistema operativo aportar los mecanismos necesarios para que esto fuera posible.

En el apartado 2.3, “Soluciones para la planificación a nivel de aplicación”, se revisaron algunas de las soluciones existentes en la bibliografía para el problema de la planificación flexible, encontrándose que todas ellas adolecían de uno o varios de los siguientes problemas:

- Son incompletas puesto que únicamente permiten implementar un conjunto limitado de políticas de planificación.
- No abordan (o no resuelven completamente) la implementación de políticas de planificación en sistemas multiprocesadores.
- No permiten el aislamiento de los distintos algoritmos entre sí ni con el sistema operativo.
- No abordan la gestión de los recursos compartidos (o lo hacen de forma incompleta)[1].
- Son soluciones particulares que en ningún caso persiguen la compatibilidad mediante su integración en algún estándar.

En este capítulo se presenta una interfaz para la definición de algoritmos de planificación con la que se pretende superar las limitaciones anteriormente citadas. A continuación se procede a enumerar y justificar los principales requerimientos establecidos para la interfaz:

- La interfaz constituirá una ampliación de las interfaces POSIX (en sus versiones Ada y C). Allí donde sea posible habrá que utilizar operaciones ya definidas por el estándar POSIX (posiblemente extendiendo su funcionalidad) en lugar de definir otras nuevas.

  Justificación: La integración con POSIX facilitará el uso de la interfaz por programadores familiarizados con este estándar, en número creciente debido a su alto grado de aceptación entre los fabricantes de sistemas operativos.

1. Esta limitación será tratada en el capítulo 5, “Interfaz de usuario para la definición de protocolos de sincronización”.

- Las nuevas políticas de planificación definidas por la aplicación deberán ser compatibles con los algoritmos estándar definidos en el POSIX y en el Ada, de forma que puedan coexistir en el sistema tareas planificadas bajo ambos tipos de políticas.

  Justificación: Es importante mantener la compatibilidad con las aplicaciones existentes. Además, el cumplimiento de este requerimiento conjuntamente con el anterior, podría permitir la incorporación de la interfaz a una futura revisión del estándar POSIX.

- Debe ser posible aislar las partes críticas del sistema frente a fallos en los algoritmos de planificación definidos por la aplicación.

  Justificación: Lo normal será que los algoritmos de planificación definidos por la aplicación no sean tan fiables como las políticas estándar implementadas en el sistema operativo. En muchas aplicaciones puede resultar importante que tareas críticas planificadas por las políticas estándar no se vean afectadas por posibles fallos en los algoritmos de planificación definidos por el usuario.

- Los algoritmos de planificación definidos por la aplicación podrán actuar sobre tareas de un mismo proceso no pudiendo hacerlo sobre tareas de varios procesos o sobre los procesos mismos.

  Justificación: Un algoritmo de planificación definido por la aplicación que actuase sobre tareas de distintos procesos resultaría muy complejo y no parece justificado por los requerimientos de las aplicaciones. Los procesos se utilizan para establecer barreras de protección entre distintas partes de la aplicación y en ese sentido parece que únicamente deberían ser planificados utilizando las políticas predefinidas en el sistema operativo.

- Debe ser posible definir varios algoritmos de planificación de forma que convivan en el sistema simultáneamente.

  Justificación: En distintas partes de la aplicación podrían requerirse algoritmos de planificación diferentes y esto resultaría más fácil de implementar si fuera posible definir varios algoritmos de planificación que operasen de forma simultánea.

- La posibilidad de anidar algoritmos de planificación definidos por la aplicación, de forma que el de bajo nivel (junto con las tareas por él planificadas) sea visto por el de alto nivel como una más de sus tareas planificadas, se considera como una funcionalidad opcional.

  Justificación: Parece que el soporte para esta funcionalidad podría resultar complejo, no compensando sus posibles ventajas.

- No debe presuponerse el entorno en el que ejecutarán los algoritmos de planificación definidos por la aplicación. La implementación podrá elegir si desea que sean ejecutados dentro del núcleo del sistema operativo o, por el contrario, prefiere que lo sean en el entorno de las tareas de usuario.

  Justificación: Dentro del núcleo de algunos sistemas operativos únicamente puede ejecutarse software fiable, por lo que los planificadores deberán utilizar el espacio de direcciones de usuario, incluso si ello implica menor eficiencia. En otros sistemas podría permitirse la ejecución de los planificadores dentro del núcleo por razones de eficiencia.

- Los algoritmos de planificación definidos por la aplicación deberán ser capaces de conocer y limitar el tiempo durante el cual ejecutan las tareas por ellos planificadas.

  Justificación: Algunos algoritmos basan sus decisiones de planificación en el tiempo de CPU consumido por sus tareas.

- Los algoritmos de planificación definidos por la aplicación tendrán la capacidad de indicar el siguiente instante en el pretenden tomar una decisión de planificación. Además

podrán indicar el reloj, de entre los proporcionados por el sistema, en base al que desean que se determine dicho instante.

Justificación: La posibilidad de tomar decisiones de planificación en instantes determinados puede ser esencial para muchos algoritmos de planificación. Por otra parte diferentes planificadores pueden necesitar distintos relojes. Así por ejemplo, considerando los relojes definidos por el estándar POSIX, un planificador podría necesitar un reloj de alta resolución, sincronizado o monótono.

- Deberá ser posible asignar a las tareas parámetros de planificación específicos del algoritmo definido por la aplicación bajo el que van a ser planificadas. Esos parámetros específicos podrán ser obtenidos por el planificador en cualquier momento, en particular cuando la tarea requiera ser planificada por él.

  Justificación: En el caso general un algoritmo definido por la aplicación requerirá parámetros de planificación diferentes de los definidos para las tareas planificadas bajo las políticas estándar.

- Existirá un mecanismo que permita a los algoritmos de planificación definidos por la aplicación aceptar o rechazar una tarea cuando ésta requiera ser planificada (ya sea en el momento de su creación o posteriormente). El rechazo de una tarea deberá ser notificado tanto a la aplicación como al propio sistema operativo.

  Justificación: En ocasiones un algoritmo de planificación podrá decidir no planificar una tarea, bien porque sus parámetros de planificación son incorrectos o porque considera que el sistema no dispone de suficientes recursos para hacerlo. El sistema operativo deberá tener noticia de este hecho para no vincular la tarea con el planificador; por su parte, la notificación a la aplicación será necesaria para que ésta pueda tomar las acciones que considere oportunas.

- Los algoritmos de planificación definidos por la aplicación deberán ser capaces de tomar las decisiones de planificación en los momentos precisos. Para ello serán informados por el sistema operativo de todos los eventos, relevantes para el algoritmo de planificación, que sufran sus tareas planificadas.

  Justificación: Un algoritmo debe ser capaz de indicar la nueva tarea o tareas a ejecutar tan pronto como ocurra cualquier circunstancia que pueda afectar a la situación de planificación de sus tareas.

- Un algoritmo de planificación definido por la aplicación podrá filtrar los eventos que le son notificados por el sistema.

  Justificación: Por razones de eficiencia, los tipos de eventos que no sean relevantes para un determinado planificador podrán ser enmascarados por éste. De esta forma el sistema operativo podrá descartarlos, no invirtiendo tiempo en su procesado y notificación.

- Las tareas planificadas deberán ser capaces de pasar información específica a su algoritmo de planificación en tiempo de ejecución.

  Justificación: Esto podría ser necesario para aquellos algoritmos de planificación que necesiten información sobre el estado de sus tareas que sea más detallada que la facilitada por el sistema operativo.

- Deberá ser posible implementar algoritmos de planificación definidos por la aplicación aptos para sistemas multiprocesadores y que puedan sacar partido de este tipo de arquitecturas. No se requiere que los algoritmos sean independientes de la arquitectura de cada sistema particular: la planificación eficiente en multiprocesadores requerirá

conocer el sistema, principalmente el número de procesadores capaces de ejecutar tareas planificadas por la aplicación de forma concurrente.

Justificación: El soporte para multiprocesadores es un requerimiento general del estándar POSIX. Es muy difícil construir un algoritmo de planificación que sea válido y eficiente en cualquier arquitectura, por lo que se supone que será necesario conocer información sobre ella, en particular el número de procesadores disponibles.

- Un algoritmo de planificación definido por la aplicación deberá ser capaz de activar y/o suspender varias de sus tareas planificadas utilizando una sola operación y de forma simultánea.

  Justificación: Esto puede resultar útil en muchas ocasiones, en particular en aquellas arquitecturas multiprocesadoras en las que existe un único planificador, quizás ejecutando en un procesador de propósito general, que planifica tareas que pueden ser ejecutadas en un conjunto de procesadores de propósito particular (p.e. DSPs).

4.2. Descripción del modelo

En la figura 4.1 se muestra el modelo elegido para la planificación definida por la aplicación. Cada planificador de aplicación se implementa mediante una tarea especial, la cual es responsable de planificar un conjunto de tareas. En base a esta distinción, podemos considerar dos tipos de tareas:

- *Tareas planificadoras de aplicación*: tareas especiales utilizadas para implementar los algoritmos de planificación.
- *Tareas normales*: el resto de las tareas, encargadas de ejecutar del código de la aplicación.

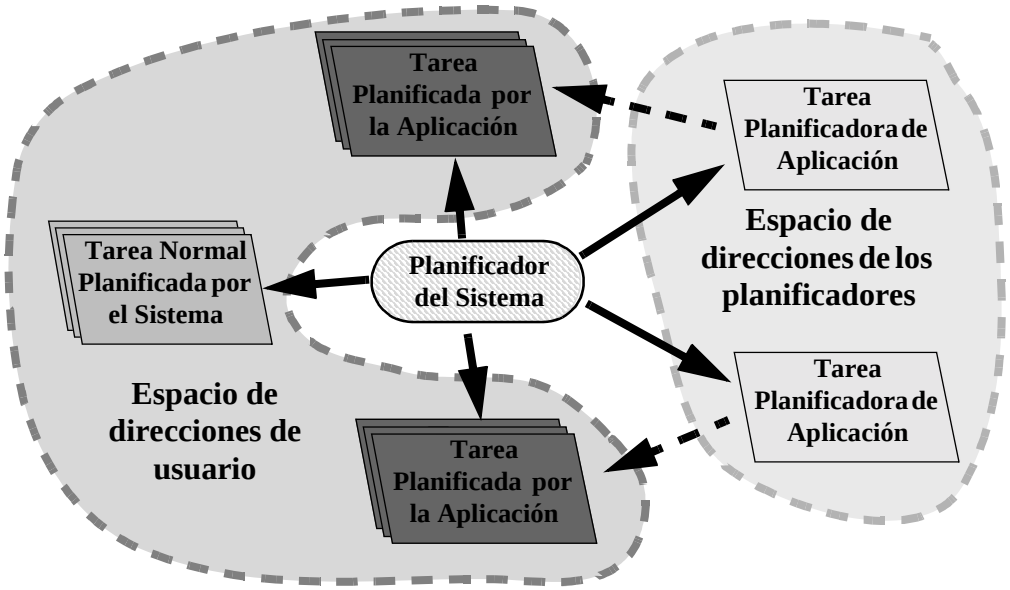

Figura 4.1: Modelo para la planificación definida por la aplicación

En la figura los planificadores de aplicación aparecen en un área de memoria diferente al del resto de las tareas. Esto es así puesto que el modelo permite tanto que los planificadores ejecuten en el espacio de direcciones del núcleo como que lo hagan en el de las tareas de aplicación. El que se use una u otra alternativa depende de la implementación elegida por el sistema operativo. De esta forma, en implementaciones en las que se pretenda mejorar la eficiencia, los

planificadores podrán ejecutar en el espacio de direcciones del núcleo, mientras que en aquellas en las que la clave sea la seguridad, los planificadores ejecutarán en el mismo espacio que las demás tareas de la aplicación.

La principal implicación de que en el modelo se dejen abiertas ambas posibilidades es que, por razones de portabilidad, los planificadores no deben compartir información de forma directa con el núcleo del sistema operativo ni con tareas normales. Los planificadores de aplicación pertenecientes a un mismo proceso sí pueden compartir información entre ellos. Esto puede resultar útil a la hora de construir planificadores multitarea para plataformas multiprocesadoras.

Considerando la forma en la que las tareas son planificadas, pueden clasificarse en:

- *Tareas planificadas por el sistema*: estas tareas son planificadas directamente por el sistema operativo sin que intervenga ninguna tarea planificadora. En esta clasificación entrarían tanto las tareas normales planificadas por el sistema como los planificadores de aplicación.
- *Tareas planificadas por un algoritmo definido por la aplicación*: estas tareas también son planificadas por el sistema operativo, pero su tarea planificadora puede decidir cuándo quiere que lo sean, ya que como se expondrá más adelante, tiene el poder de activarlas y suspenderlas cuando desee.

El modelo permite que una tarea planificadora sea a su vez planificada por otra planificadora, formándose así una jerarquía de planificadores. Sin embargo, esta posibilidad no se considera clave en el modelo, por lo que se deja a la elección de cada implementación particular el soportarla o no.

4.2.1. Planificación de tareas por el sistema operativo

Como se ha comentado anteriormente todas las tareas, independientemente de su tipo, son planificadas por el planificador del sistema como si se trataran de tareas normales, esto es, para el planificador del sistema cada tarea tiene una prioridad y una política de planificación estándar: “FIFO con prioridades”, “round-robin con prioridades” o “servidor esporádico” en el caso de tratarse de un thread POSIX o “FIFO con prioridades” si es una tarea Ada. La forma en que los citados atributos de planificación globales se ven afectados por el tipo de tarea de que se trate se detalla a continuación:

- Las tareas normales planificadas por el sistema se comportan de la forma habitual: su prioridad base y política son las que les han sido asignadas en su creación o posteriormente mediante la utilización de una operación para tal fin, pudiendo su política ser cualquiera de entre las estándar. Por otra parte, su prioridad activa es la base o una prioridad heredada debido al uso de recursos compartidos (mutexes u objetos protegidos).
- En el caso de las tareas planificadas por la aplicación, se comportan de forma similar a las anteriores, salvo que su prioridad siempre deberá ser menor o igual que la de su planificador y que siempre serán tratadas por el sistema operativo como tareas de política “FIFO con prioridades”. Además de sus atributos de planificación de sistema (política y prioridad), estas tareas tienen un conjunto de parámetros de planificación dependientes de la política definida por la aplicación y que únicamente son interpretados por su planificador. Éste tomará las acciones de planificación propias de la política que implementa en base a dichos parámetros, aunque a la hora de competir con las demás tareas del sistema, las tareas planificadas lo harán únicamente en base a su prioridad activa y política (“FIFO con prioridades”).

- Para las tareas planificadoras su prioridad base es la asignada en su creación o posteriormente de forma dinámica, mientras que de cara al sistema operativo su política es siempre “FIFO con prioridades”. Al igual que las demás tareas, un planificador también puede heredar prioridades debido al uso de recursos compartidos, pero además, también hereda todas las prioridades heredadas por esa misma causa por cualquiera de sus tareas planificadas.

En resumen, las tareas planificadoras o planificadas por la aplicación se comportan como tareas normales, teniendo preferencia sobre tareas de prioridad menor y resultando expulsadas por tareas de mayor prioridad. Cuando comparten nivel de prioridad con tareas normales se comportan como si de tareas de política “FIFO con prioridades” se trataran, ejecutando hasta que se bloqueen o sean expulsadas.

La única excepción a esta regla la constituye el hecho de que un planificador siempre tiene preferencia sobre sus tareas planificadas, esto es, cuando una tarea planificada se encuentra en disposición de comenzar a ejecutar y su tarea planificadora se encuentra activa, el planificador del sistema cederá la CPU a dicha tarea planificadora. Esto es necesario para permitir que ésta pueda tomar decisiones de planificación en los momentos oportunos y de esta forma decidir cuál, de entre todas sus tareas planificadas, debe ser la siguiente en ejecutar.

Como se ha comentado, en presencia de herencia de prioridad, el planificador hereda las mismas prioridades que sus tareas planificadas. Si esto no ocurriera, un planificador que prentendiese activar una de sus tareas planificadas que ha heredado una prioridad alta, tendría que esperar por causa de otras tareas de prioridad intermedia. El retraso en la activación de la tarea en posesión del recurso, provocaría que las tareas bloqueadas en él sufrieran un efecto de inversión de prioridad no acotada. Esto implica que para tareas de alta prioridad que compartan recursos con tareas de baja prioridad planificadas por la aplicación, deberá ser tenida en cuenta la ejecución del planificador a la hora de calcular sus tiempos de bloqueo.

Por lo tanto, con las reglas anteriormente expuestas se logra integrar el modelo para planificación definida por la aplicación con las políticas definidas en el estándar POSIX o el lenguaje Ada. Por supuesto, esta compatibilidad no significa que las interacciones entre tareas de distintas políticas que compartan los mismos niveles de prioridades sean fáciles de analizar teóricamente, por lo que lo normal será que cada planificador y sus tareas planificadas ejecuten en un rango de prioridades exclusivo.

4.2.2. Relación entre el planificador y sus tareas planificadas

La relación entre una tarea planificadora y las tareas por ella planificadas se muestra de forma esquemática en la figura 4.2. Los planificadores tienen el poder de ejecutar determinadas acciones de planificación sobre sus tareas planificadas, y por otra parte, son informados de las circunstancias sufridas por cualquiera de sus tareas planificadas que puedan resultar relevantes para el algoritmo de planificación que implementan.

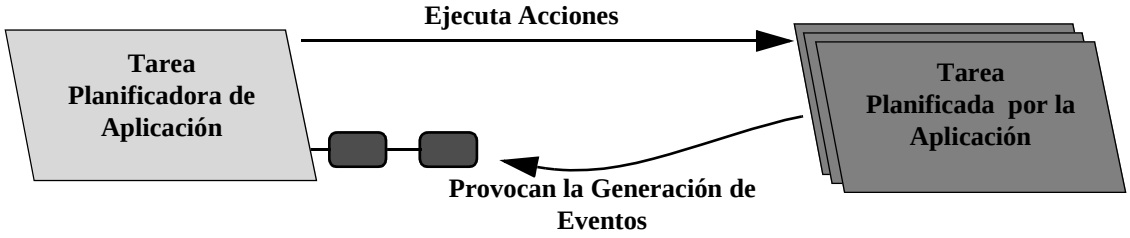

Figura 4.2: Relación entre una tarea planificadora y sus tareas planificadas.

Una tarea planificadora puede ejecutar sobre sus tareas planificadas las siguientes acciones:

- *Aceptación*: una tarea que ha requerido ser planificada por un planificador es aceptada. Para tomar esa decisión el planificador puede basarse en los atributos de planificación de la tarea, así como en otros factores conocidos por él como pueden ser el número de tareas que planifica en ese momento, la carga total de dichas tareas, etc. A partir de este momento la tarea podrá ser activada por el planificador cuando éste lo desee.
- *Rechazo*: una tarea que ha requerido ser planificada por un planificador es rechazada. El rechazo de una tarea implica que la función de creación retorne un código de error cuando se trata de un thread POSIX o que se eleve una excepción en el caso de tareas Ada. Ambos casos se tratarán en detalle en los apartados 4.3, “Descripción de la interfaz C” y 4.4, “Descripción de la interfaz Ada”.
- *Activación*: el planificador puede activar cualquiera de sus tareas planificadas siempre que hayan sido previamente suspendidas por él. Se considera un error pretender activar una tarea suspendida por el sistema operativo, ya sea temporalmente por haber invocado una operación de suspensión temporizada o al bloquearse tratando de tomar un recurso del sistema. La activación de una tarea no significa su inmediata puesta en ejecución, para ello primero deberá competir con las demás tareas del sistema en base a su prioridad tal y como se explicó en el apartado 4.2.1.
- *Suspensión*: al igual que en el caso de la activación, el planificador puede decidir en cualquier momento la suspensión de cualquiera de sus tareas planificadas siempre que ésta se encuentre en ejecución[1] o en disposición de estarlo.

Además existen otras acciones relacionadas con los mutexes de protocolo definido por la aplicación, como son la aceptación, rechazo o concesión de uno de estos mutexes a una tarea. Estas acciones serán tratadas en detalle en el capítulo 5.

Es importante resaltar que el planificador es capaz de ejecutar varias de estas acciones de planificación a la vez, lo que simplifica la implementación de planificadores para sistemas multiprocesador.

Junto al de “acción de planificación”, otro de los conceptos fundamentales de nuestro modelo es el de “evento de planificación”. Como mostrábamos en la figura 4.2, el planificador es notificado sobre las circunstancias en la vida de sus tareas planificadas y otros eventos que puedan ser relevantes para el algoritmo de planificación. Esta notificación se realiza en forma de eventos de planificación que son encolados en una cola FIFO asociada a cada planificador. La información incluida en cada uno de esos eventos es:

- *Tipo de evento*: indica la circunstancia que provocó la generación del evento.
- *Tarea causante del evento*: la mayoría de los eventos son causados por una tarea, la cual se indica en este campo de información.
- *Información adicional*: depende del tipo del evento y puede ser:
  - *Información asociada a una señal*: el evento puede consistir en la aceptación de una señal POSIX por parte del planificador. En este campo se proporciona la información asociada a dicha señal.
  - *Información específica*: las tareas planificadas pueden invocar directamente a su planificador enviándole además una información adicional. El planificador sabrá

1. Una tarea puede encontrarse en ejecución en un sistema multiprocesador en el que el planificador ejecuta en un procesador mientras que alguna de sus tareas planificadas lo hace en otro.

interpretar dicha información específica del algoritmo de planificación que implementa.

- *Código de evento definido por la aplicación*: las tareas planificadas pueden invocar directamente a su planificador indicando la causa de la invocación mediante un valor numérico que será interpretado por el planificador.
- *Prioridad heredada o perdida* como consecuencia del uso de recursos del sistema (mutexes o objetos protegidos). Tratado en detalle en el capítulo 5, “Interfaz de usuario para la definición de protocolos de sincronización”.
- *Mutex de protocolo de aplicación* sobre el que la tarea realizó alguna acción. Tratado en detalle en el capítulo 5, “Interfaz de usuario para la definición de protocolos de sincronización”.

Los eventos que pueden ser notificados al planificador junto con su descripción y campo de información asociada se muestran en la tabla 4.1. Conviene notar que no se incluye un evento para notificar la expulsión de la CPU de una tarea planificada. Aunque este evento parecería útil para medir los tiempos de ejecución de las tareas desde su planificador, se presenta el problema de que a éste le resultaría muy difícil saber el instante real en que una tarea comienza a ejecutar. Así por ejemplo, podría plantearse la situación en que un planificador activase una tarea cuya prioridad de sistema fuera inferior a la suya. En ese caso, entre la activación y el comienzo de la ejecución de la tarea podría pasar un tiempo indeterminado debido a la ejecución de tareas de prioridad intermedia. Por tanto, cuando se desee medir el tiempo de ejecución de una tarea, es mucho más sencillo utilizar para ese propósito los relojes POSIX de tiempo de ejecución.

Tabla 4.1: Eventos de planificación

Tipo de Evento	Descripción	Información Adicional
New Task	Una tarea requiere ser planificada por el planificador.	ninguna
Terminate Task	Una tarea ha terminado o deja de ser planificada por el planificador.	ninguna
Ready	Una tarea ha sido activada por el sistema operativo.	ninguna
Block	Una tarea ha sido bloqueada por el sistema operativo.	ninguna
Yield	Una tarea desea ceder la CPU (invocando `sched_yield()` o ejecutado `delay 0.0`).	ninguna
Change Scheduling Parameters	Una tarea ha cambiado sus parámetros de planificación específicos del algoritmo definido por aplicación.	ninguna
Explicit Call	Una tarea ha invocado explícitamente a su planificador.	Código de evento
Explicit Call with Data	Una tarea ha invocado explícitamente a su planificador con información asociada.	Información específica

Tabla 4.1: Eventos de planificación (cont.)

Tipo de Evento	Descripción	Información Adicional
Signal	Una señal perteneciente al conjunto de señales esperadas ha sido aceptada por el planificador.	Información asociada a la señal
Timeout	El tiempo límite de espera de eventos ha expirado.	ninguna
En el capítulo 5, “Interfaz de usuario para la definición de protocolos de sincronización” se añadirán a esta lista otros eventos relacionados con los mutexes de protocolo de sistema y de protocolo definido por la aplicación.		

Los dos últimos eventos de la tabla 4.1 (*Signal* y *Timeout*) no tienen una correspondencia directa con situaciones en la vida de las tareas planificadas; sin embargo, son fundamentales para lograr algunos de los requisitos marcados para la interfaz. El evento *Timeout* se produce cuando se ha cumplido el tiempo límite programado por la tarea planificadora para la operación de espera de eventos. Este evento puede ser utilizado por el planificador como un mecanismo de control del correcto funcionamiento del sistema (“watchdog”) o también para ejecutar una determinada acción que debe realizarse en un instante de tiempo determinado. Por su parte el evento *Signal* amplía enormemente las posibilidades del planificador ya que le permite esperar a un conjunto de señales POSIX a la vez que espera al resto de los eventos. Las señales son un mecanismo de comunicación inter/intra-procesos extensamente utilizado en los sistemas POSIX, y en particular son el mecanismo de notificación utilizado por los temporizadores (incluidos los de tiempo de ejecución).

La tabla 4.2 muestra el estado en el que se queda una tarea planificada tras generar un evento para su planificador. En la mayoría de los casos, el sistema operativo suspende a la tarea generadora hasta que su planificador decida de nuevo su activación (salvo que el evento se encuentre enmascarado). Este comportamiento es fundamental para los sistemas multiprocesadores, ya que de no hacerse así, una tarea planificada que provocara la generación de un evento, continuaría ejecutando en paralelo con su planificador antes de que éste tuviera oportunidad de decidir cuál debe ser su nuevo estado.

Tabla 4.2: Estado de la tarea asociada tras un evento de planificación.

Tipo de Evento	Estado de la tarea asociada tras el evento
New Task	Suspendida a la espera de ser activada por el planificador
Terminate Task	No cambia
Ready	Suspendida a la espera de ser activada por el planificador
Block	No cambia
Yield	Suspendida a la espera de ser activada por el planificador
Change Scheduling Parameters	No cambia
Explicit Call	Suspendida a la espera de ser activada por el planificador
Explicit Call with Data	Suspendida a la espera de ser activada por el planificador

Tabla 4.2: Estado de la tarea asociada tras un evento de planificación. (cont.)

Tipo de Evento	Estado de la tarea asociada tras el evento
Signal	No hay tarea asociada
Timeout	No hay tarea asociada

El número máximo de eventos de planificación que pueden encolarse en las colas FIFO asociadas a los planificadores podrá estar limitado por la implementación[1], en ese caso, los eventos generados para un planificador cuya cola se encuentre llena serán descartados. Lo normal será que este número máximo no sea excesivamente alto. En particular en el caso de que no se cambien dinámicamente los parámetros de planificación de las tareas planificadas, el número máximo de eventos que como máximo pueden encontrarse pendientes para un planificador es igual al número de sus tareas planificadas más uno. Este límite se alcanzaría en la situación en que, mientras una tarea de alta prioridad planificada por el sistema se encuentra en ejecución, el sistema operativo activara todas las tareas planificadas por el planificador y además se produjera un evento de expiración de tiempo límite o de llegada de una señal.

Cuando se incluye el cambio dinámico de los parámetros de las tareas planificadas no puede hablarse de un límite en el número de eventos pendientes, ya que, una tarea de alta prioridad planificada por el sistema podría cambiar los parámetros de las tareas planificadas cuantas veces quisiera antes de abandonar el procesador y permitir ejecutar al planificador de aplicación. Sin embargo, conociendo la forma en que la aplicación realiza estas acciones, es posible acotar el número de eventos pendientes.

Con en modelo presentado, una tarea planificadora será una tarea especial cuyo código normalmente consistirá en un lazo en el que espera los eventos de planificación que le son notificados por el sistema y, en función del evento recibido y del estado de sus tareas planificadas, determina la acción o acciones de planificación a ejecutar.

Por supuesto, el lazo de espera y procesado de eventos no tiene por que ser infinito, pudiendo el planificador finalizar su ejecución en el momento que estime oportuno. Se plantea entonces el problema de definir qué ocurre con las tareas planificadas una vez que su planificador haya finalizado. Podría plantearse que en dicha situación el sistema operativo debería cancelar o abortar todas las tareas planificadas por el planificador que finaliza, pero esto complicaría en gran medida el modelo. Por tanto optamos por la solución más sencilla consistente en no modificar el estado de las tareas planificadas, salvo por el hecho de que los eventos que puedan generar serán descartados inmediatamente por el sistema operativo. Quedará por tanto a la responsabilidad de la aplicación el asegurarse de que el planificador no finalice mientras tenga alguna tarea planificada. En caso de que se desee forzar la finalización de toda la aplicación, la tarea planificadora podría cancelar o abortar sus tareas planificadas y esperar la llegada de los eventos notificando la terminación de éstas antes de finalizar su ejecución.

4.3. Descripción de la interfaz C

Como se expuso en la introducción de este capítulo, la interfaz en lenguaje C para planificación definida por la aplicación ha sido diseñada de forma que se integre con el conjunto de interfaces POSIX. Se ha tratado de introducir el menor número posible de nuevas operaciones, de forma

1. Una posible mejora podría consistir en permitir especificar este límite para cada tarea planificadora en el momento de su creación.

que allí donde ha sido factible, se ha optado por ampliar servicios ya existentes en lugar de definir interfaces totalmente nuevas. Aún así, lógicamente hay aspectos de la interfaz que han requerido la definición de nuevas operaciones. En esos casos se ha perseguido que éstas sean lo más parecidas posibles a otras con una funcionalidad análoga ya existentes en el estándar, de forma que se facilite su uso por programadores acostumbrados al estándar POSIX.

También con el mismo objetivo de la integración con POSIX, los nombres de los nuevos identificadores introducidos siguen el criterio utilizado en las nuevas incorporaciones a este estándar, consistente en comenzar los identificadores con la palabra "posix" seguida de una palabra clave de la nueva funcionalidad añadida. Así los identificadores introducidos comenzarán por el prefijo "`posix_appsched_`", salvo en los casos en los que se amplie un servicio ya existente en los cuales se seguirá la nomeclatura POSIX tradicional, comenzando los identificadores por los prefijos "`pthread_`", "`sched_`", etc.

Los nuevos servicios han sido añadidos a los ficheros estándar de cabeceras `<pthread.h>` y `<sched.h>`. La nueva funcionalidad introducida permite a las aplicaciones:

- *Crear threads planificados y planificadores*: la interfaz define un conjunto de nuevos atributos que permiten identificar los planificadores de aplicación, indicar cuál es el planificador de un thread planificado por la aplicación y asignar los parámetros de planificación específicos del algoritmo. Además se define una nueva política (`SCHED_APP`) para los threads planificados.
- *Gestionar y ejecutar acciones de planificación*: se proporcionan operaciones que permiten al planificador crear una lista de acciones de planificación y posteriormente ejecutar dichas acciones.
- *Asignar las propiedades de un thread planificador de aplicación*. La propiedades que se pueden modificar son: el tipo de tiempo límite de espera de eventos (relativo o absoluto), el reloj utilizado para medir dicho tiempo límite, la información de retorno para los threads planificados y la máscara de eventos que serán descartados por el sistema.
- *Obtener la información asociada a los eventos de planificación*: la interfaz define símbolos para indentificar los tipos de eventos y estructuras que representan los propios eventos y que permiten acceder a su información asociada.
- *Invocar explícitamente al thread planificador desde un thread planificado*.
- *Asignar y obtener los datos específicos de un thread desde otro diferente*. Para ello se extiende la funcionalidad POSIX que permite asociar datos específicos con los threads.

Cada uno de los aspectos de la interfaz se describe en detalle en los siguientes apartados. Además, en el anexo A se presenta la descripción completa de la interfaz tal y como se propone para su integración en el estándar POSIX.

4.3.1. Creación de threads planificadores y planificados

En el estándar POSIX, la forma de definir las características de un thread es utilizando un objeto de tipo `pthread_attr_t`. Entre otros atributos, en este objeto se especifica la política y parámetros de planificación del thread, consistiendo estos últimos en una estructura de tipo `sched_param`. Nuestra interfaz extiende el conjunto de atributos estándar añadiendo tres nuevos: `appscheduler_state`, `appscheduler` y `appsched_param`. Otra posibilidad habría sido incluir estos parámetros como nuevos campos de la estructura `sched_param`. Esta opción presenta el grave inconveniente de que esta estructura es utilizada por la librería estándar C (`libc`), por lo que su modificación obligaría a la recompilación de la citada librería.

La forma elegida para indicar que un thread es un planificador de aplicación es mediante el nuevo atributo `appscheduler_state`, el cual puede tomar los valores `PTHREAD_REGULAR` o `PTHREAD_APPSCHEDULER`, siendo el valor por omisión `PTHREAD_REGULAR`. Ambos símbolos son declarados en el fichero de cabeceras `<pthread.h>`. La forma de asignar y obtener el valor del citado atributo de un objeto de atributos es mediante las funciones mostradas a continuación:

```
#include <pthread.h>
int pthread_attr_setappschedulerstate (pthread_attr_t *attr,
                                       int appschedstate);
int pthread_attr_getappschedulerstate (pthread_attr_t *attr,
                                       int *appschedstate);
```

La interfaz no permite la modificación del atributo `appscheduler_state` de forma dinámica tras la creación del thread, lo que significa que los threads planificadores deberán serlo desde el momento de su creación, no pudiendo cambiar de tipo posteriormente. Esto no supone ninguna limitación importante, ya que la decisión de si un thread va a ser un planificador de aplicación deberá tomarse al proceder a la escritura de su cuerpo. Una vez creado el thread, el valor de su atributo `appscheduler_state` puede obtenerse mediante la función:

```
#include <pthread.h>
int pthread_getappschedulerstate (pthread_t thread,
                                  int *appschedstate);
```

Por su parte, la forma de indicar que una tarea va a ser planificada por un planificador de aplicación consiste en asignar el valor `SCHED_APP` a su política de planificación (atributo `schedpolicy`) mediante la función estándar `pthread_attr_setschedpolicy()`. El símbolo `SCHED_APP` se define en `<sched.h>` junto con los símbolos correspondientes a las políticas POSIX estándar: `SCHED_FIFO`, `SCHED_RR`, `SCHED_SPORADIC`, y `SCHED_OTHER`.

Para los threads de política `SCHED_APP` es necesario indicar, mediante su atributo `appscheduler`, cuál va ha ser su thread planificador de aplicación. El valor de dicho atributo puede asignarse y obtenerse de un objeto de atributos mediante las funciones:

```
#include <pthread.h>
int pthread_attr_setappscheduler (pthread_attr_t *attr,
                                  pthread_t scheduler);
int pthread_attr_getappscheduler (const pthread_attr_t *attr,
                                  pthread_t *scheduler);
```

Además, como ya comentamos en la descripción del modelo, los threads planificados por la aplicación pueden ser creados con unos parámetros de planificación específicos de su política que serán interpretados por su planificador. Esta información es facilitada al sistema operativo mediante el nuevo atributo `appsched_param`, gestionado en un objeto de atributos mediante las funciones:

```
#include <pthread.h>
int pthread_attr_setappschedparam (pthread_attr_t *attr,
                                   const void * param,
                                   size_t paramsize);
int pthread_attr_getappschedparam (const pthread_attr_t *attr,
                                   void * param,
                                   size_t *paramsize);
```

El parámetro `paramsize` indica el tamaño de los parámetros de planificación específicos del algoritmo apuntados por `param`. Es necesario gestionar de forma explícita este parámetro puesto que el tamaño de los parámetros de planificación específicos del algoritmo no es conocido de antemano por el sistema operativo al ser dependiente de cada aplicación particular.

También puede accederse a los atributos `appscheduler` y `appsched_param` de forma dinámica después de la creación del thread mediante las funciones:

```
#include <pthread.h>
int pthread_setappscheduler (pthread_t thread,
                             pthread_t scheduler);
int pthread_getappscheduler (pthread_t thread,
                             pthread_t *scheduler);
int pthread_setappschedparam (pthread_t thread,
                              const void * param,
                              size_t paramsize);
int pthread_getappschedparam (pthread_t thread,
                              void * param,
                              size_t *paramsize);
```

Estas funciones permiten hacer que un thread planificado bajo alguna de las políticas estándar pase a ser planificado por un planificador de aplicación. Para ello, primero será preciso dar valor a su atributo `appscheduler` (y opcionalmente también a su atributo `appsched_param`) para posteriormente cambiar su política a `SCHED_APP` mediante la función estándar `pthread_setschedparam()`. En el caso de que el planificador rechace planificar el thread, la función `pthread_setschedparam()` retornará un código de error y el thread se mantendrá con su antigua política.

La interfaz no permite el cambio dinámico de planificador, por lo que la invocación de la función `pthread_setappscheduler()` para cambiar el planificador de un thread de política `SCHED_APP` constituye un error que será detectado por la implementación. Para cambiar el atributo `appscheduler` de un thread, será preciso primero cambiar su política a una de las estándar mediante la función `pthread_setschedparam()`, lo que provocará la generación de un evento de terminación para el thread planificador.

La interfaz anteriormente expuesta permitiría definir threads planificadores que a su vez sean de política `SCHED_APP`, creándose de este modo una jerarquía de planificadores. Sin embargo, el que se soporte o no esta posibilidad se deja como un comportamiento dependiente de la implementación, ya que no se considera un punto clave del modelo y, por otro lado, podría complicar de forma notable la implementación de la interfaz.

4.3.2. Gestión y ejecución de las acciones de planificación

Las acciones de planificación son, junto con los eventos de planificación tratados en el apartado 4.3.4, uno de los puntos claves de la interfaz. Constituyen el mecanismo mediante el cual un thread planificador comunica al sistema operativo las operaciones que desea realizar sobre sus threads planificados. Como expusimos al principio del presente capítulo, el planificador debe ser capaz de ordenar la ejecución de un conjunto de acciones de planificación mediante una única operación. La interfaz define el objeto opaco `posix_appsched_actions_t` que permite agrupar varias acciones de planificación para que puedan posteriormente ser ejecutadas de forma conjunta. El número máximo de acciones que puede contener uno de estos objetos está limitado por la aplicación, debiendo ser como mínimo igual al número máximo de threads

permitido en el sistema. Como se explicó en el apartado 4.2.2, las acciones que es posible insertar en una de estas listas son[1]:

- Aceptación de un thread que pretende vincularse con el planificador. Tras la ejecución de esta acción el thread permanecerá suspendido hasta que el planificador ejecute una acción de activación sobre él.
- Rechazo de un thread que pretende vincularse con el planificador.
- Activación de un thread recién aceptado o que fue previamente suspendido por el planificador.
- Suspensión de un thread vinculado con el planificador que se encuentra en estado ejecutable.

Los prototipos de las funciones proporcionadas por la interfaz para la gestión de las listas de acciones `posix_appsched_actions_t` anteriormente citadas son:

```
#include <sched.h>
int posix_appsched_actions_init(
              posix_appsched_actions_t *sched_actions);
int posix_appsched_actions_destroy(
              posix_appsched_actions_t *sched_actions);
int posix_appsched_actions_addaccept(
              posix_appsched_actions_t *sched_actions,
              pthread_t thread);
int posix_appsched_actions_addreject(
              posix_appsched_actions_t *sched_actions,
              pthread_t thread);
int posix_appsched_actions_addactivate(
              posix_appsched_actions_t *sched_actions,
              pthread_t thread);
int posix_appsched_actions_addsuspend(
              posix_appsched_actions_t *sched_actions,
              pthread_t thread);
```

La principal operación de la interfaz es `posix_appsched_execute_actions()`, cuyo prototipo se muestra a continuación:

```
#include <sched.h>
int posix_appsched_execute_actions(
              const posix_appsched_actions_t *sched_actions,
              const sigset_t *set,
              const struct timespec *timeout,
              struct timespec *current_time,
              struct posix_appsched_event *event);
```

Esta operación permite al planificador ejecutar la lista de acciones apuntada por `sched_actions` y suspenderse hasta la llegada del siguiente evento de planificación. Las acciones de planificación serán ejecutadas por el sistema operativo en el mismo orden en el que fueron añadidas al objeto de acciones `sched_actions`. Tras la llegada del evento, el planificador se pone de nuevo en estado ejecutable con el parámetro `event` indicando el evento

1. En el capítulo 5, “Interfaz de usuario para la definición de protocolos de sincronización” se añadirán a estas acciones otras nuevas que permiten aceptar o rechazar la creación de un mutex de protocolo definido por la aplicación así como conceder a un thread uno de estos mutexes.

recibido. En el caso de que en el momento ejecutar la operación `posix_appsched_execute_actions()` ya exista algún evento pendiente la función retornará inmediatamente.

En los casos que sea necesario puede utilizarse el parámetro `timeout` para indicar el tiempo límite de espera, de forma que la función retorne si se alcanza dicho tiempo antes de recibir un evento. Mediante las funciones presentadas en el apartado 4.3.3, es posible indicar el reloj con el que se desea medir dicho tiempo límite y también si éste debe ser interpretado de forma relativa o absoluta, esto es, como un intervalo de tiempo transcurrido el cual la función deberá retornar, o como el instante en el que debe abortarse la espera de eventos. Para indicar que no se desea programar ningún tiempo límite ha de asignarse el valor puntero nulo al parámetro `timeout`.

La función puede también programarse para que retorne en el caso de que una señal de las incluidas en el conjunto apuntado por `set` sea generada para el planificador. Esta posibilidad simplifica el uso de los temporizadores POSIX, incluidos los de tiempo de ejecución, como fuentes de eventos de planificación. La forma de indicar que no se desea que la función espere ninguna señal es pasando un puntero nulo en el lugar del parámetro `set`.

El instante de tiempo en el que se reciben los eventos puede constituir un dato relevante a la hora de decidir las acciones de planificación a ejecutar. Pensando en los algoritmos de planificación en los que eso ocurre, es posible requerir a través del parámetro `current_time` la hora medida inmediatamente antes de que la función `posix_appsched_execute_actions()` retorne. El uso de este parámetro permite mejorar la eficiencia, puesto que con ello se evita que la tarea planificadora tenga que obtener ese dato mediante una nueva llamada al sistema. La forma de indicar que no se requiere esa información es asignando el valor puntero nulo al parámetro `current_time`. El reloj con el que se desea medir el instante de retorno puede configurarse mediante la función `posix_appschedattr_setclock()` definida en el apartado 4.3.3.

4.3.3. Asignación de las propiedades de un thread planificador

Existe un conjunto de propiedades específicas de los threads planificadores no incluidas en los atributos y parámetros de planificación expuestos en el apartado 4.3.1, “Creación de threads planificadores y planificados”. Estas propiedades deben ser configuradas desde el mismo thread planificador y permiten especificar:

- El reloj utilizado en la función `posix_appsched_execute_actions()` tanto para la medida del instante actual como para el tiempo límite.
- El tipo de tiempo límite (absoluto o relativo) utilizado por la función `posix_appsched_execute_actions()`.
- La información de respuesta enviada a un thread planificado que ha invocado explícitamente a su planificador mediante la función `posix_appsched_invoke_withdata()`. Esta función será expuesta en el apartado 4.3.5, “Invocación explícita del planificador”.
- El conjunto de eventos de planificación que no resultan relevantes para el algoritmo implementado por el planificador y deben ser descartados por el sistema operativo.

El reloj utilizado por la función `posix_appsched_execute_actions()` tanto para la medida del instante actual como para el tiempo límite es asignado y obtenido mediante las funciones `posix_appschedattr_setclock()` y `posix_appschedattr_getclock()`, cuyos prototipos se muestran a continuación:

```
#include <sched.h>
int posix_appschedattr_setclock (clockid_t clockid);
int posix_appschedattr_getclock (clockid_t *clockid);
```

Los valores de `clockid` que es posible especificar son los correspondientes a todos los relojes descritos en el estándar POSIX a excepción de los correspondientes a los de tiempo de ejecución, esto es:

- `CLOCK_REALTIME`: representa el reloj que mide la hora global de sistema, o lo que es lo mismo, los segundos y nanosegundos transcurridos desde la "Época" (1 de enero de 1970). Las aplicaciones pueden cambiar el valor de este reloj mediante la función estándar `clock_settime()`.
- `CLOCK_MONOTONIC`: representa el reloj monótono del sistema. Este reloj mide el tiempo transcurrido desde un instante arbitrario pero fijo en el pasado, como puede ser la hora de arranque del sistema o la "Época". Se diferencia del anterior en que su valor no puede ser cambiado por la aplicación, con lo que siempre crece monótonamente con el transcurso del tiempo.

Por omisión el reloj asignado a un planificador en el momento de su creación es `CLOCK_REALTIME` por constituir `CLOCK_MONOTONIC` una funcionalidad opcional y por tanto no disponible en todos los sistemas POSIX.

La interfaz también define las funciones `posix_appschedattr_setflags()` y `posix_appschedattr_getflags()`. Estas funciones constituyen una forma general de cambiar las propiedades del planificador, ya que utilizan como parámetro una máscara de bits que podría ser utilizada para configurar varias propiedades de forma simultánea. En la actualidad, la única característica de los threads planificadores que es posible cambiar mediante su uso, es el tipo de tiempo límite de espera de eventos de la función `posix_appsched_execute_actions()`. Para ello se define en `<sched.h>` el símbolo `POSIX_APPSCHED_ABSTIMEOUT` que sirve para indicar que se desea utilizar un tiempo límite absoluto. Si el bit asociado a ese símbolo está a cero, el tiempo límite será relativo. Los prototipos de las funciones `posix_appschedattr_setflags()` y `posix_appschedattr_getflags()`se muestran a continuación:

```
#include <sched.h>
int posix_appschedattr_setflags (int flags);
int posix_appschedattr_getflags (int *flags);
```

Por omisión los planificadores se crean con tiempo límite de espera de eventos relativo.

Las funciones `posix_appschedattr_setreplyinfo()` y `posix_appschedattr_getreplyinfo()` permiten acceder al atributo `replyinfo` del planificador que las invoca. Sus prototipos se muestran a continuación:

```
#include <sched.h>
int posix_appschedattr_setreplyinfo(const void *reply,
                                    int reply_size);
int posix_appschedattr_getreplyinfo(void *reply,
                                    int *reply_size)
```

El atributo `replyinfo` representa un área de memoria de la longitud indicada por `reply_size`. El contenido de este atributo constituye la información enviada como respuesta a las tareas planificadas que hayan ejecutado la función `posix_appsched_invoke_withdata()` (presentada en el apartado 4.3.5).

Como dijimos anteriormente, otra de las propiedades que es posible configurar para un planificador es su máscara de eventos de planificación. Mediante está máscara el planificador comunica al sistema operativo los eventos que no quiere que le sean enviados por ser irrelevantes para el algoritmo de planificación que implementa. Esta funcionalidad permite mejorar la eficiencia del sistema ya que evita el consumo de tiempo innecesario debido a la entrega de eventos que no van a ser utilizados.

La interfaz proporciona el tipo opaco `posix_appsched_eventset_t` (definido en `<sched.h>`) para representar conjuntos (o máscaras) de eventos. También se facilitan funciones para gestionar objetos de ese tipo, las cuales permiten añadir y eliminar eventos así como saber si un evento se encuentra incluido en uno de estos conjuntos. Los prototipos de estas funciones son:

```
#include <sched.h>
int posix_appsched_emptyset (posix_appsched_eventset_t *set);
int posix_appsched_fillset (posix_appsched_eventset_t *set);
int posix_appsched_addset (posix_appsched_eventset_t *set,
                           int appsched_event);
int posix_appsched_delset (posix_appsched_eventset_t *set,
                           int appsched_event);
int posix_appsched_ismember (const posix_appsched_eventset_t *set,
                             int appsched_event);
```

Una vez configurada la máscara haciendo uso de las funciones anteriores, ésta se puede activar mediante la ejecución por parte del planificador de la función `posix_appschedattr_seteventmask()`. La interfaz también facilita la función `posix_appschedattr_geteventmask()` para que un planificador pueda obtener su máscara de eventos actual. Los prototipos de ambas funciones son:

```
#include <sched.h>
int posix_appschedattr_seteventmask (
              const posix_appsched_eventset_t *set);
int posix_appschedattr_geteventmask (
              posix_appsched_eventset_t *set);
```

En el momento de la creación de un thread planificador su máscara de eventos se encuentra vacía, lo que significa que el sistema operativo le enviará todos los eventos para él generados. Deberá invocarse la función `posix_appschedattr_seteventmask()` con la máscara deseada si se pretende cambiar este comportamiento por omisión.

4.3.4. Eventos de planificación

Los eventos de planificación constituyen el mecanismo utilizado por el sistema operativo para notificar a los threads planificadores que ha ocurrido alguna circunstancia potencialmente relevante para la política de planificación que implementan. Pueden ser causados por circunstancias en la vida de sus threads planificados como:

- Un thread requiere ser planificado por el planificador.
- Un thread planificado finaliza su ejecución o se desvincula de su planificador.
- El sistema operativo bloquea o pone en estado ejecutable a un thread planificado.
- Se ha producido un cambio en los parámetros de planificación de un thread planificado que son específicos del algoritmo de planificación definido por la aplicación.

- Un thread planificado invoca la operación estándar `sched_yield()`.
- Un thread planificado invoca explícitamente su planificador.

Además existen otras circunstancias que pueden provocar la generación de un evento de planificación que no están directamente relacionadas con los threads planificados como son:

- Se ha cumplido el tiempo límite de espera de eventos en una operación `posix_appsched_execute_actions()`.
- El planificador ha aceptado una señal de las esperadas por la función `posix_appsched_execute_actions()`.

En el fichero de cabeceras `<sched.h>` se define un conjunto de constantes simbólicas que permiten identificar los distintos tipos de eventos. La tabla 4.3 muestra dichas constantes junto con la descripción de la causa que provoca la generación del evento.

Tabla 4.3: Eventos de planificación

Tipo de Evento	Causa del Evento
`POSIX_APPSCHED_NEW`	Un thread solicita ser planificado por el planificador bien en el momento de su creación o posteriormente al cambiar de forma dinámica su política a `SCHED_APP`.
`POSIX_APPSCHED_TERMINATE`	Un thread planificado finaliza su ejecución o se desvincula de su planificador.
`POSIX_APPSCHED_READY`	El sistema operativo pone en estado ejecutable a un thread planificado por una razón distinta a que el planificador haya ejecutado una acción de activación sobre él.
`POSIX_APPSCHED_BLOCK`	El sistema operativo bloquea a un thread planificado por una razón distinta a que el planificador haya ejecutado una acción de suspensión sobre él.
`POSIX_APPSCHED_YIELD`	Un thread planificado invoca la operación estándar `sched_yield()`.
`POSIX_APPSCHED_-CHANGE_SCHED_PARAM`	Se ha producido un cambio en los parámetros de planificación de un thread planificado que son específicos del algoritmo de planificación de usuario.
`POSIX_APPSCHED_-EXPLICIT_CALL`	Un thread planificado ha invocado explícitamente su planificador utilizando la función `posix_appsched_invoke_scheduler()`.
`POSIX_APPSCHED_-EXPLICIT_CALL_WITH_DATA`	Un thread planificado ha invocado explícitamente su planificador utilizando la función `posix_appsched_invoke_withdata()`.
`POSIX_APPSCHED_SIGNAL`	El planificador ha aceptado una señal de las esperadas por `posix_appsched_execute_actions()`.

Tabla 4.3: Eventos de planificación (cont.)

Tipo de Evento	Causa del Evento
`POSIX_APPSCHED_TIMEOUT`	Se ha cumplido el tiempo límite de espera de eventos en la función `posix_appsched_execute_actions()`.
En el capítulo 5, “Interfaz de usuario para la definición de protocolos de sincronización” se añadirán a esta lista otros eventos relacionados con los mutexes de protocolo de sistema y de protocolo definido por la aplicación.	

Como se comentó en el apartado 4.3.3, un planificador tiene la capacidad de enmascarar algunos eventos mediante la función `posix_appschedattr_seteventmask()`. La generación de un evento no enmascarado provoca el encolado de una estructura del tipo `struct posix_appsched_event` (definida en `<sched.h>`) en la cola FIFO de eventos de planificación asociada al planificador. Además, en el caso de que el planificador se encuentre bloqueado en una operación `posix_appsched_execute_actions()` será puesto en estado ejecutable con el parámetro `event` apuntando al evento recién generado. Los campos que forman la estructura `posix_appsched_event` se describen en la tabla 4.4.

Tabla 4.4: Campos de la estructura `posix_appsched_event`.

Tipo	Nombre	Descripción
`int`	`event_code`	Tipo del evento. Identifica la causa que provocó su generación.
`pthread_t`	`thread`	Identificador del thread que causó el evento.
`union posix_appsched_eventinfo`	`event_info`	Información adicional asociada con el evento y dependiente de su tipo.
`size_t`	`info_size`	Tamaño de la información enviada por un thread planificado usando `posix_appsched_invoke_withdata()`.

El campo `event_code` permite identificar el tipo del evento; este campo tomará uno de los valores indicados en la tabla 4.3. Por su parte, el campo `thread` permite identificar el thread causante del evento. En los eventos no causados por un thread (`POSIX_APPSCHED_SIGNAL` y `POSIX_APPSCHED_TIMEOUT`) este campo tomará un valor indefinido. El campo `event_info` es a su vez una unión `posix_appsched_eventinfo` formada por los campos mostrados en la tabla 4.5.

El campo de la unión `posix_appsched_eventinfo` utilizado depende del tipo del evento. Así cuando éste es `POSIX_APPSCHED_SIGNAL` el campo utilizado es `siginfo`, una estructura `siginfo_t` definida por el estándar POSIX en el fichero de cabeceras `<signal.h>`. Además del número y la causa de generación, la estructura `siginfo_t` permite obtener la información asociada con la señal (en el caso de que la hubiera). La posibilidad de asociar información a una señal fue incluida en el estándar POSIX como una característica de las denominadas “Señales de Tiempo Real”. La inclusión de este campo en la unión `posix_appsched_eventinfo` permite utilizar esta funcionalidad como mecanismo para proporcionar información adicional

Tabla 4.5: Campos de la unión `posix_appsched_eventinfo`.

Tipo	Nombre	Descripción
`siginfo_t`	`siginfo`	Información relativa a una señal aceptada (número, causa e información asociada con la señal).
`void *`	`info`	Información específica para el planificador enviada por un thread planificado usando `posix_appsched_invoke_withdata()`.
`int`	`user_event_code`	Información específica para el planificador enviada por un thread planificado usando `posix_appsched_invoke_scheduler()`.
En el capítulo 5, "Interfaz de usuario para la definición de protocolos de sincronización" se añadirán a esta unión los campos `sched_priority` y `mutex`.		

al planificador por parte del elemento generador de la señal (un temporizador, otra tarea, un manejador de dispositivo, etc.).

Por su parte, el campo `info` es utilizado cuando el evento generado es `POSIX_APPSCHED_EXPLICIT_CALL_WITH_DATA`. Como expondremos en el apartado 4.3.5, la función `posix_appsched_invoke_withdata()` permite a un thread planificado invocar de forma explícita a su planificador enviándole una información consistente en un conjunto de bytes de longitud variable. Dicha información puede ser obtenida por el thread planificador mediante el campo `info` de la unión `posix_appsched_eventinfo`. Para este tipo de evento el campo `info_size` de la estructura `posix_appsched_event` indica la longitud del mensaje transmitido, mientras que para todos los demás eventos este campo no se utiliza y vale cero.

El campo restante de la unión `posix_appsched_eventinfo` (`user_event_code`) se utiliza con los eventos de tipo `POSIX_APPSCHED_EXPLICIT_CALL`. Este tipo de evento, al igual que ocurría con `POSIX_APPSCHED_EXPLICIT_CALL_WITH_DATA`, se genera como consecuencia de la invocación del planificador desde un thread planificado, en este caso mediante la ejecución de la función `posix_appsched_invoke_scheduler()` (tratada en detalle en el apartado 4.3.5). A diferencia del caso anterior, la función `posix_appsched_invoke_scheduler()` únicamente permite enviar un código numérico que el thread planificador puede obtener accediendo a este campo de la unión `posix_appsched_eventinfo`.

4.3.5. Invocación explícita del planificador

La interfaz proporciona las funciones `posix_appsched_invoke_scheduler()` y `posix_appsched_invoke_withdata()` que permiten a un thread invocar de forma explícita a su planificador. A continuación mostramos los prototipos C de ambas funciones:

```
#include <sched.h>
int posix_appsched_invoke_scheduler (int user_event_code);
int posix_appsched_invoke_withdata (const void *msg,
                                    size_t msg_size,
                                    void *reply,
                                    size_t *reply_size);
```

La utilización por parte de un thread planificado de la función `posix_appsched_invoke_scheduler()` supone que el thread se suspenda y que se genere para su planificador un evento de tipo `POSIX_APPSCHED_EXPLICIT_CALL`. Además, el valor pasado a la función a través del parámetro `user_event_code`, será accesible para el planificador mediante el componente del mismo nombre del campo `event_info` del evento recibido.

La función `posix_appsched_invoke_withdata()` constituye una forma alternativa de invocación explícita del planificador que podrá ser utilizada cuando sea preciso un intercambio más complejo de información entre los threads planificado y planificador. La ejecución de esta función provoca que el thread planificado se suspenda y que el planificador reciba un evento de tipo `POSIX_APPSCHED_EXPLICIT_CALL_WITH_DATA`. Por otra parte, el thread planificado puede utilizar el parámetro `msg` para enviar información adicional a su planificador. En esos casos el parámetro `msg_size` deberá indicar el tamaño en bytes de la información transmitida. El planificador podrá acceder a dicha información mediante el componente `info` del campo `event_info` del evento recibido. Además, para este tipo de evento el campo `info_size` indica la longitud del mensaje transmitido.

Una vez activado de nuevo por su planificador, el thread planificado recibirá la respuesta de éste en el área de memoria apuntada por el parámetro `reply`. El planificador configura esta respuesta utilizando la función `posix_appsched_setreplyinfo()` presentada en el apartado 4.3.3. La información de retorno también es de tamaño variable, y su longitud es la indicada por el parámetro `reply_size`.

En la mayoría de los casos, los threads planificador y planificado podrían conocer la longitud de los mensajes intercambiados, lo que parecería indicar que los parámetros con los que se indica su tamaño son innecesarios. Sin embargo, es fundamental incluir estos parámetros en las funciones puesto que, como dicta el modelo, un thread planificador puede encontrase en un espacio de direcciones diferente al de sus threads planificados. Es por tanto posible que el sistema operativo tenga que copiar los mensajes entre espacios de direcciones diferentes, para lo que necesitará conocer su tamaño. El sistema operativo no conoce este dato, puesto que depende del algoritmo de planificación, por lo que tiene que serle comunicado explícitamente a través de los citados parámetros.

En los casos en los que el thread planificado no desee enviar y/o recibir información del planificador, debe poner los parámetros `msg` y/o `reply` al valor puntero nulo .

4.3.6. Acceso remoto a los datos específicos del thread

Lo normal será que los planificadores asocien cada uno de sus threads planificados con una estructura de control con información relativa a su estado de planificación. Cuando un planificador procesa un evento, las acciones de planificación a ejecutar dependerán del estado de sus threads asociados y especialmente del estado del que causó ese evento. Este thread se identifica mediante el campo `thread` de tipo `pthread_t` de la estructura `posix_appsched_event`, por lo que sería muy útil disponer de un mecanismo que permitiese obtener la información de planificación de una forma sencilla y eficiente a partir de un objeto de tipo `pthread_t`. Sin embargo, el estándar POSIX define este tipo como un objeto opaco. Sobre él únicamente es posible realizar la operación de comparación mediante la función `pthread_equal()`, por lo que no es posible construir a nivel de aplicación y de forma portable una lista indexada por identificadores de threads cuyos tiempos de acceso sean menores de O(n) (siendo n el número de threads).

Esta es la razón por la que nuestra interfaz extiende la funcionalidad POSIX denominada "Thread-Specific Data", la cual permite la asociación de datos con threads mediante el uso de un identificador o llave. El estándar POSIX únicamente proporciona operaciones que permiten la gestión de esos datos desde la tarea propietaria, por lo que nuestra interfaz añade funciones que posibilitan asignar u obtener el valor asociado con una llave desde un thread distinto a su propietario. Los prototipos de las nuevas funciones añadidas se muestran a continuación:

```
#include <pthread.h>
int pthread_setspecific_for (pthread_key_t key,
                             pthread_t thread,
                             const void *value);
int pthread_getspecific_from (pthread_key_t key,
                              pthread_t thread,
                              void **value);
```

El modo de utilización de ambas funciones por parte de un thread planificador podría ser el descrito a continuación: cuando un nuevo thread requiere ser planificado (evento `POSIX_APPSCHED_NEW`) el planificador le asocia su correspondiente estructura de control mediante la función `pthread_setspecific_for()`. En el tratamiento de los sucesivos eventos, el planificador podrá obtener esa estructura utilizando la función `pthread_getspecific_from()` y el campo `thread` del evento recibido.

4.4. Descripción de la interfaz Ada

En este apartado presentaremos una interfaz Ada que define servicios equivalentes a los definidos en la interfaz C descrita en el apartado anterior. De la misma manera que su homóloga C se integraba en el estándar POSIX.1, esta interfaz se integra en el estándar POSIX.5b ("binding" en lenguaje Ada del POSIX.1). Toda la funcionalidad relativa a la definición de algoritmos de planificación se incluye en el nuevo paquete `POSIX_Application_Scheduling` y ha sido diseñada según las pautas marcadas en el POSIX.5b, las cuales se encuentran descritas en su anexo B titulado "Rationale and Notes". Así, se descarta la utilización explícita de punteros (tan utilizada en la interfaz C), se opta por utilizar conjuntos de constantes en lugar de tipos enumerados (para facilitar la expansión), se utilizan unidades genéricas allí donde la interfaz C utilizaba punteros a objetos de tamaño indefinido, se sobrecargan procedimientos allí donde la interfaz C utilizaba valores especiales en los parámetros para indicar distintos comportamientos, etc.

Los servicios definidos en el nuevo paquete `POSIX_Application_Scheduling` permiten a las aplicaciones:

- *Convertir tareas en planificadores y planificadas*: La interfaz proporciona procedimientos que permiten que una tarea normal de política `FIFO_Within_Priorities` se convierta en una tarea planificada o planificadora.
- *Gestionar y ejecutar acciones de planificación*: se proporcionan operaciones que permiten al planificador crear una lista de acciones de planificación y posteriormente ejecutar dichas acciones.
- *Asignar las propiedades de una tarea planificadora de aplicación*. La propiedades que se pueden modificar son: el tipo de tiempo límite de espera de eventos (relativo o absoluto), el reloj utilizado para medir dicho tiempo límite, la información de retorno para las tareas planificadas y la máscara de eventos que serán descartados por el sistema.

- *Obtener la información asociada a los eventos de planificación*: la interfaz define constantes para indentificar los tipos de eventos y funciones para acceder a su información asociada.
- *Invocar explícitamente la tarea planificadora desde una tarea planificada*: se proporciona un procedimiento para la invocación del planificador y un paquete genérico para los casos en los que junto con la invocación se desea enviar al planificador alguna información específica y recibir una respuesta de éste.

Además de estos servicios, también se ha modificado el paquete estándar `POSIX_Timers` para que las aplicaciones puedan utilizar los relojes y temporizadores de tiempo de ejecución, servicio que, aunque ya incluido en el POSIX.1, no se encuentra estandarizado aún en los "bindings" Ada. Como ya se ha comentado con anterioridad, esta es una funcionalidad que puede resultar interesante e incluso indispensable para implementar algunos algoritmos de planificación.

La opción de ofrecer la definición de algoritmos de planificación en forma de una interfaz que se integra en el estándar POSIX no era la única alternativa existente. Podía haberse optado por incluirla como una extensión del propio lenguaje, de forma similar a como el manual de referencia define los anexos especializados (anexo de tiempo real, anexo de sistemas distribuidos, etc.). Con esta segunda opción se podrían ofrecer un conjunto de directivas al compilador ("pragmas") que permitieran definir una tarea como planificadora o planificada desde el momento de su creación. La extensión definida podría apoyarse en un sistema operativo POSIX que ofreciese la interfaz descrita en el apartado anterior o bien la funcionalidad podría estar totalmente implementada dentro de la librería de tiempo de ejecución del lenguaje Ada.

Puesto que uno de los objetivos del trabajo presentado es ofrecer también la interfaz POSIX-C, parece lógico que la funcionalidad esté implementada dentro del propio sistema operativo, por lo que resultaría absurda su duplicación en la librería de tiempo de ejecución de Ada. Sin embargo, el disponer de pragmas que permitiesen definir las tareas como planificadoras o planificadas desde el momento de su creación, aunque no supondría ningún aumento de funcionalidad con respecto a la interfaz presentada, sí que podría simplificar su uso. Por esta razón, aunque en un principio se ha optado por no modificar la librería de tiempo de ejecución para no complicar la implementación, la inclusión de la interfaz como una extensión del lenguaje constituye una clara línea de trabajo futuro que se esbozará en el capítulo 6, "Conclusiones y trabajo futuro".

En los siguientes apartados procederemos a describir en detalle cada uno de los aspectos de la interfaz.

4.4.1. Conversión de tareas en planificadas y planificadoras

El que la interfaz no defina nuevos pragmas impide que el programador pueda indicar desde el momento de creación de un tarea si desea que ésta sea un planificador de aplicación o una tarea planificada. Por tanto, las tareas deberán ser creadas como tareas normales de política `FIFO_Within_Priorities` y posteriormente convertidas a planificadores o planificadas mediante la inclusión en su cuerpo de una llamada a alguno de los procedimientos descritos en este apartado.

Así, para convertir una tarea normal en un planificador de aplicación, la interfaz define el siguiente procedimiento:

```
procedure Become_An_Application_Scheduler;
```

A partir de la ejecución de este procedimiento la tarea comienza a comportarse como un planificador de aplicación, con lo que estará en disposición de aceptar eventos de planificación y otras tareas podrán requerir ser planificadas por ella.

El procedimiento a utilizar para convertir una tarea en planificada por un planificador de aplicación depende de si se desean utilizar parámetros de planificación específicos o no. En los casos en que el algoritmo de planificación no precise de la utilización de tales parámetros utilizaremos el procedimiento `Change_Task_Policy_To_App_Sched`. La interfaz de este procedimiento es la mostrada a continuación:

```
procedure Change_Task_Policy_To_App_Sched
   (Scheduler : in Ada.Task_Identification.Task_Id);
```

Cuando las tareas a la hora de solicitar la vinculación con un planificador, deban especificar un conjunto de parámetros de planificación propios del algoritmo (y por tanto definidos por la aplicación), deberemos utilizar el procedimiento `Change_Task_Policy` del paquete genérico `Application_Defined_Policy`:

```
generic
   type Parameters is private;
package Application_Defined_Policy is

   procedure Change_Task_Policy
      (Scheduler : in Ada.Task_Identification.Task_Id;
       Param     : in Parameters);

   procedure Get_Parameters
      (T     : in  Ada.Task_Identification.Task_Id;
       Param : out Parameters);

end Application_Defined_Policy;
```

En este paquete el parámetro genérico es el objeto de parámetros de planificación específicos del algoritmo de planificación. Cuando una tarea requiere ser planificada por un planificador de aplicación, éste puede obtener sus parámetros de planificación utilizando el procedimiento `Application_Defined_Policy.Get_Parameters`. En función del valor de dichos parámetros y/o de otros factores conocidos por la tarea planificadora, ésta podrá aceptar o rechazar la nueva tarea. En el caso de ser rechazada, los procedimientos `Application_Defined_Policy.Change_Task_Policy` y `Change_Task_Policy_To_App_Sched` elevarán la excepción `POSIX_Error` para indicar a la tarea que su requerimiento ha sido rechazado por el planificador.

Una vez que una tarea que ha invocado `Change_Task_Policy_To_App_Sched` o `Application_Defined_Policy.Change_Task_Policy` ha sido aceptada por su planificador su política de planificación pasa a ser “definida por la aplicación”. No es posible cambiar el planificador de una de estas tareas, aunque sí que está permitido volver a invocar el procedimiento `Application_Defined_Policy.Change_Task_Policy` con el mismo

planificador pero diferentes parámetros de planificación, lo que provocará la generación para su planificador de un evento indicando que se ha producido el cambio de los parámetros.

La interfaz también proporciona una operación que permite reconvertir una tarea planificada por la aplicación a la política Ada estándar `FIFO_Within_Priorities`:

```
procedure Change_Task_Policy_To_FIFO_Within_Priorities;
```

La invocación de este procedimiento por parte de una tarea planificada por la aplicación provocará la generación de un evento de terminación para su planificador.

Utilizando en una misma tarea las operaciones `Become_An_Application_Scheduler` y `Application_Defined_Policy.Change_Task_Policy` (o alternativamente `Change_Task_Policy_To_App_Sched`) sería posible crear una tarea planificadora que fuera a su vez planificada por otro planificador, constituyendo de esta forma una jerarquía de planificadores. El que se soporte o no esta posibilidad se deja como dependiente de la implementación, ya que no se considera un punto clave del modelo, mientras que puede complicar la implementación.

4.4.2. Gestión y ejecución de las acciones de planificación

La interfaz define el tipo privado `Scheduling_Actions`, que constituye una lista de acciones de planificación para ser ejecutadas de forma conjunta por el sistema operativo. Se definen procedimientos para inicializar y destruir uno de estos objetos, así como para añadir acciones de aceptación, rechazo, activación y suspensión de tareas planificadas[1]:

```
type Scheduling_Actions is private;
procedure Initialize (Sched_Actions : in out Scheduling_Actions);
procedure Destroy (Sched_Actions : in out Scheduling_Actions);
procedure Add_Accept
   (Sched_Actions : in out Scheduling_Actions;
    T             : in     Ada.Task_Identification.Task_Id);
procedure Add_Reject
   (Sched_Actions : in out Scheduling_Actions;
    T             : in     Ada.Task_Identification.Task_Id);
procedure Add_Activate
   (Sched_Actions : in out Scheduling_Actions;
    T             : in     Ada.Task_Identification.Task_Id);
procedure Add_Suspend
   (Sched_Actions : in out Scheduling_Actions;
    T             : in     Ada.Task_Identification.Task_Id);
```

El punto central de la interfaz lo constituyen las familias de procedimientos `Execute_Actions` y `Execute_Actions_With_Timeout` que permiten a una tarea planificadora ejecutar una lista de acciones y suspenderse hasta la llegada del siguiente evento de planificación. Las acciones de planificación son ejecutadas en el mismo orden en el que fueron añadidas al objeto de tipo `Scheduling_Actions`. Tras la llegada del evento, la tarea planificadora pasa de nuevo a estado ejecutable con el evento recibido en el parámetro `Event`. En el caso de que en el momento de ejecutar alguno de estos procedimientos ya haya algún

1. En el capítulo 5, “Interfaz de usuario para la definición de protocolos de sincronización” se añadirán a estas acciones otras nuevas que permiten aceptar o rechazar la creación de un mutex de protocolo definido por la aplicación así como conceder uno de estos mutexes a una tarea.

evento pendiente retornaría inmediatamente. La forma básica del procedimiento `Execute_Actions` únicamente tiene dos parámetros: la lista de acciones a ejecutar (`Sched_Actions`) y el evento recibido (`Event`). La interfaz de este procedimiento se muestra a continuación:

```
procedure Execute_Actions
   (Sched_Actions : in  Scheduling_Actions;
    Event         : out Scheduling_Event);
```

La interfaz también define una versión del procedimiento `Execute_Actions` que incluye el parámetro `Set` para representar un conjunto de señales POSIX:

```
procedure Execute_Actions
   (Sched_Actions : in  Scheduling_Actions;
    Set           : in  POSIX_Signals.Signal_Set;
    Event         : out Scheduling_Event);
```

Cuando se genera para el planificador una de las señales incluidas en el conjunto `Set` mientras se encuentra bloqueado este procedimiento, se pone de nuevo en estado ejecutable con el parámetro `Event` indicando que se ha recibido una señal. Esta funcionalidad simplifica el uso de los temporizadores POSIX como fuentes de eventos de planificación (incluidos los de tiempo de ejecución introducidos en el apartado 4.4.6).

El instante de tiempo en el que se reciben los eventos puede constituir un dato relevante para decidir las acciones de planificación a ejecutar. Para los casos en que así sea, se define una versión del procedimiento `Execute_Actions` que permite al planificador obtener la hora medida inmediatamente antes de que el procedimiento retorne:

```
procedure Execute_Actions
   (Sched_Actions : in  Scheduling_Actions;
    Event         : out Scheduling_Event;
    Current_Time  : out POSIX.Timespec);
```

La utilización de este procedimiento evita que la tarea planificadora tenga que obtener la hora actual mediante una nueva llamada al sistema, mejorándose de esta manera la eficiencia del algoritmo. El reloj con el que se desea medir el instante de retorno puede configurarse mediante el procedimiento `Set_Clock` definido en el apartado 4.4.3.

Además, se define una versión del procedimiento que permite utilizar los parámetros `Set` y `Current_Time` de forma simultánea:

```
procedure Execute_Actions
   (Sched_Actions : in  Scheduling_Actions;
    Set           : in  POSIX_Signals.Signal_Set;
    Event         : out Scheduling_Event;
    Current_Time  : out POSIX.Timespec);
```

La familia de procedimientos `Execute_Actions_With_Timeout` es igual a la anterior, salvo en que añade el parámetro `Timeout`. Este parámetro se utiliza para indicar el tiempo límite de espera, de forma que el procedimiento retorne si se alcanza dicho tiempo antes de recibir un evento. Mediante las funciones presentadas en el apartado 4.4.3, es posible indicar el reloj con el que se desea medir dicho tiempo límite y también si éste debe ser interpretado de forma relativa o absoluta, esto es, como un intervalo de tiempo transcurrido el cuál la función deberá

retornar, o como el instante en el que debe abortarse la espera de eventos. La familia de procedimientos `Execute_Actions_With_Timeout` se muestra a continuación:

```
procedure Execute_Actions_With_Timeout
   (Sched_Actions : in  Scheduling_Actions;
    Event         : out Scheduling_Event;
    Timeout       : in  POSIX.Timespec);
procedure Execute_Actions_With_Timeout
   (Sched_Actions : in  Scheduling_Actions;
    Set           : in  POSIX_Signals.Signal_Set;
    Event         : out Scheduling_Event;
    Timeout       : in  POSIX.Timespec);
procedure Execute_Actions_With_Timeout
   (Sched_Actions : in  Scheduling_Actions;
    Event         : out Scheduling_Event;
    Timeout       : in  POSIX.Timespec;
    Current_Time  : out POSIX.Timespec);
procedure Execute_Actions_With_Timeout
   (Sched_Actions : in  Scheduling_Actions;
    Set           : in  POSIX_Signals.Signal_Set;
    Event         : out Scheduling_Event;
    Timeout       : in  POSIX.Timespec;
    Current_Time  : out POSIX.Timespec);
```

4.4.3. Asignación de las propiedades de una tarea planificadora

Una tarea, una vez convertida en planificador de aplicación, puede configurar diferentes aspectos de su comportamiento:

- El reloj utilizado en las familias de procedimientos `Execute_Actions` y `Execute_Actions_With_Timeout` para la medida del instante actual. Este mismo reloj también será utilizado en los procedimientos `Execute_Actions_With_Timeout` para detectar el sobrepaso del tiempo límite.
- El tipo de tiempo límite (absoluto o relativo) utilizado por los procedimientos `Execute_Actions_With_Timeout`.
- La información de respuesta enviada a una tarea planificada que ha invocado explícitamente a su planificador. La forma de configurar esta información será abordada conjuntamente con la invocación explícita del planificador en el apartado 4.4.5.
- El conjunto de eventos de planificación que no resultan relevantes para el algoritmo implementado por el planificador y deben ser descartados por el sistema operativo.

El reloj utilizado por los procedimientos de las familias `Execute_Actions` y `Execute_Actions_With_Timeout` es asignado y obtenido mediante las operaciones `Set_Clock` y `Get_Clock`, cuyas interfaces se muestran a continuación:

```
procedure Set_Clock (Clock : in POSIX_Timers.Clock_Id);
function Get_Clock return POSIX_Timers.Clock_Id;
```

El único valor de `Clock` que es posible especificar es `CLOCK_REALTIME`, puesto que éste es el único reloj que define en la actualidad el estándar POSIX.5b. `CLOCK_REALTIME` representa el reloj que mide la hora global de sistema, o lo que es lo mismo, el tiempo transcurrido desde la "Época" (1 de enero de 1970). Las aplicaciones pueden cambiar el tiempo medido por este reloj

mediante el procedimiento estándar `Set_Time`, por lo que no debe asumirse que su valor crezca monótonamente con el transcurso del tiempo.

Aunque el estándar POSIX.5b únicamente define el reloj `CLOCK_REALTIME`, este mismo estándar avisa que en futuras revisiones es muy posible que se incluyan otros relojes como el monótono ya definido en el estándar POSIX.1. Por lo tanto `Set_Clock` y `Get_Clock` se definen con vistas a esta futura revisión del estándar.

La interfaz también define las operaciones `Set_Flags` y `Get_Flags` que constituyen una forma general de cambiar las propiedades del planificador, ya que utilizan como parámetro un conjunto de opciones que podría ser utilizado para configurar varias propiedades de forma simultánea. En la actualidad la única característica de las tareas planificadoras que es posible cambiar mediante el uso de estas operaciones es el tipo de tiempo límite de espera de eventos de los procedimientos `Execute_Actions_With_Timeout`, definiéndose la constante `POSIX_APPSCHED_ABSTIMEOUT` para indicar que se desea utilizar un tiempo límite absoluto. La interfaz de las operaciones `Set_Flags` y `Get_Flags`, junto con la definición del tipo `Scheduler_Flags` y la constante `POSIX_APPSCHED_ABSTIMEOUT` son:

```
type Scheduler_Flags is new POSIX.Option_Set;
ABSOLUTE_TIMEOUT : constant Scheduler_Flags;
procedure Set_Flags (Flags : in Scheduler_Flags);
function Get_Flags return Scheduler_Flags;
```

Por omisión los planificadores se crean con tiempo límite de espera de eventos relativo.

Como se ha dicho anteriormente, otra de las propiedades que es posible configurar para un planificador es su máscara de eventos de planificación. Mediante esta máscara el planificador comunica al sistema operativo los eventos que quiere que le sean enviados por ser relevantes para el algoritmo de planificación que implementa. Esta funcionalidad permite mejorar la eficiencia del sistema ya que evita el consumo de tiempo innecesario debido a la entrega de eventos que no van a ser utilizados.

La interfaz proporciona el tipo privado `Event_Mask` para representar conjuntos o máscaras de eventos. También se facilitan operaciones para gestionar objetos de ese tipo, las cuales permiten añadir y eliminar eventos así como saber si un evento se encuentra incluido en uno de estos conjuntos. Las interfaces de estas operaciones, junto con la definición del tipo `Event_Mask` son:

```
type Event_Mask is private;
procedure Fill (Mask : in out Event_Mask);
procedure Empty (Mask : in out Event_Mask);
procedure Add (Mask : in out Event_Mask;
               Event : in Event_Code);
procedure Del (Mask : in out Event_Mask;
               Event : in Event_Code);
function Ismember (Mask : in Event_Mask;
                   Event : in Event_Code)
                   return Boolean;
```

Una vez configurada la máscara haciendo uso de las operaciones anteriores, ésta puede ser activada mediante la ejecución por parte del planificador del procedimiento `Set_Event_Mask`. La interfaz también facilita la función `Get_Event_Mask` para que una tarea planificadora pueda obtener su máscara de eventos actual. Las interfaces de ambas operaciones son:

```
procedure Set_Event_Mask (Mask : in Event_Mask);
function Get_Event_Mask return Event_Mask;
```

Tras la conversión de una tarea en planificadora de aplicación, su máscara de eventos se encuentra vacía, lo que significa que el sistema operativo le enviará todos los eventos que sean generados para ella. Si se pretende cambiar este comportamiento por omisión deberá invocarse el procedimiento `Set_Event_Mask` con la máscara deseada.

4.4.4. Eventos de planificación

Los eventos de planificación constituyen el mecanismo utilizado por el sistema operativo para notificar a las tareas planificadoras que ha ocurrido alguna circunstancia potencialmente relevante para la política de planificación que implementan. Pueden ser causados por circunstancias en la vida de sus tareas planificadas como:

- Una tarea requiere ser planificada por el planificador.
- Una tarea planificada ha terminado o se desvincula de su planificadora.
- El sistema operativo bloquea o pone en estado ejecutable a una tarea planificada.
- Una tarea planificada cambia sus parámetros de planificación específicos del algoritmo de planificación definido por la aplicación (invocando `Application_Defined_Policy.Change_Task_Policy`).
- Una tarea planificada cede la CPU mediante una instrucción `delay 0.0`.
- Una tarea planificada invoca explícitamente su planificador.

Además existen otras circunstancias que pueden provocar la generación de un evento de planificación que no están directamente relacionadas con las tareas planificadas como son:

- Se ha cumplido el tiempo límite de espera de eventos en un procedimiento `Execute_Actions_With_Timeout`.
- El planificador ha aceptado una señal de las esperadas en un procedimiento `Execute_Actions_With_Timeout` o `Execute_Actions`.

La interfaz define un conjunto de constantes que permiten identificar los distintos tipos de eventos. La tabla 4.6 muestra dichas constantes junto con la descripción de la causa que provoca la generación del evento.

Tabla 4.6: Eventos de planificación

Tipo de Evento	Causa del Evento
`NEW_TASK`	Una tarea solicita ser planificada por el planificador.
`TERMINATE_TASK`	Una tarea planificada ha terminado o se desvincula de su planificador.
`READY`	El sistema operativo pone en estado ejecutable a una tarea planificada por una razón distinta a que el planificador haya ejecutado una acción de activación sobre él.
`BLOCK`	El sistema operativo bloquea a una tarea planificada por una razón distinta a que el planificador haya ejecutado una acción de suspensión sobre él.

Tabla 4.6: Eventos de planificación (cont.)

Tipo de Evento	Causa del Evento
YIELD	Una tarea planificada cede la CPU mediante la instrucción `delay 0.0`.
CHANGE_SCHED_PARAM	Una tarea planificada cambia sus parámetros de planificación específicos del algoritmo de planificación de usuario.
EXPLICIT_CALL	Una tarea planificada ha invocado explícitamente su planificador utilizando el procedimiento `Invoke_Scheduler`.
EXPLICIT_CALL_-WITH_DATA	Una tarea planificada ha invocado explícitamente su planificador utilizando el procedimiento `Explicit_Scheduler_Invocation.Invoke`.
SIGNAL	El planificador ha aceptado una señal de las esperadas en un procedimiento `Execute_Actions_With_Timeout` o `Execute_Actions`.
TIMEOUT	Se ha cumplido el tiempo límite de espera de eventos en un procedimiento `Execute_Actions_With_Timeout`.
En el capítulo 5, “Interfaz de usuario para la definición de protocolos de sincronización” se añadirán a esta lista otros eventos relacionados con los mutexes de protocolo de sistema y de protocolo definido por la aplicación.	

Como comentábamos en el apartado 4.4.3, un planificador tiene la capacidad de enmascarar algunos eventos mediante la función `Set_Event_Mask`. La generación de un evento no enmascarado provoca el encolado de un objeto del tipo privado `Scheduling_Event` en la cola FIFO de eventos de planificación asociada al planificador. Además, en el caso de que el planificador se encuentre bloqueado en un procedimiento `Execute_Actions` o `Execute_Actions_With_Timeout` será puesto en estado ejecutable con el evento recién generado devuelto en el parámetro `Event`. El tipo `Scheduling_Event` y sus funciones asociadas son[1]:

```
type Scheduling_Event is private;

function Get_Event_Code (Event : in Scheduling_Event)
                         return Event_Code;
function Get_Task (Event : in Scheduling_Event)
                   return Ada.Task_Identification.Task_Id;
function Get_Signal_Info (Event : in Scheduling_Event)
                          return POSIX_Signals.Signal_Info;
function Get_User_Event_Code (Event : in Scheduling_Event)
                              return Integer;
```

La función `Get_Event_Code` permite identificar el tipo del evento, retornando uno de los valores de la tabla 4.6. Por su parte, la función `Get_Task` permite identificar la tarea causante

1. En el capítulo 5, “Interfaz de usuario para la definición de protocolos de sincronización” se añadirán las funciones `Get_Sched_Priority` y `Get_Mutex` que permiten obtener información relacionada con el uso de mutexes.

del evento. La invocación de esta función para un `Scheduling_Event` cuyo tipo sea uno de los no causados por una tarea planificada (`SIGNAL` y `TIMEOUT`) provocará que se eleve la excepción `POSIX_Error`.

Cuando el evento es de tipo `SIGNAL` puede utilizarse la función `Get_Signal_Info` para obtener un objeto de tipo `POSIX_Signals.Signal_Info`. De un objeto de este tipo es posible obtener el número, la causa de generación de la señal y, en el caso de que la hubiera, la información asociada con ella. La posibilidad de asociar información con una señal es una característica de las denominadas "Señales de Tiempo Real". Esta información adicional puede ser utilizada por el elemento generador de la señal (temporizador, otra tarea, manejador de dispositivo, etc.) como mecanismo para proporcionar al planificador información relevante para el algoritmo de planificación que implementa. La invocación de la función `Get_Signal_Info` para evento cuyo tipo no sea `SIGNAL` provocará que se eleve la excepción `POSIX_Error`.

La función `Get_User_Event_Code` permite obtener el valor numérico enviado al planificador por una tarea planificada que ha ejecutado el procedimiento `Invoke_Scheduler` para invocar a su planificador (presentada en el apartado 4.4.5). Como consecuencia de la ejecución de dicho procedimiento, la tarea planificadora será suspendida por el sistema operativo y se generará para el planificador un evento de tipo `EXPLICIT_CALL`. La utilización de `Get_User_Event_Code` con un evento cuyo tipo no sea `EXPLICIT_CALL` provocará que se eleve la excepción `POSIX_Error`.

Con los eventos de tipo `EXPLICIT_CALL_WITH_DATA` puede utilizarse la función `Get_Message` del paquete genérico `Explicit_Scheduler_Invocation` para obtener la información enviada al planificador por una tarea planificada. Esta funcionalidad será tratada en el apartado 4.4.5, "Invocación explícita del planificador".

Hay algunas situaciones especiales en la vida de las tareas Ada para las que no existen eventos de planificación específicos. Por ejemplo, cuando una tarea completa su ejecución y se queda bloqueada esperando a que terminen sus tareas dependientes (tareas declaradas en su parte declarativa) se genera el evento genérico de bloqueo (`BLOCK`); lo mismo ocurre cuando una tarea se bloquea esperando a que finalice la elaboración de una tarea hija. La inexistencia eventos específicos es debida a que el sistema operativo no tiene noticia de la ocurrencia de las circunstancias que los provocan, por ser éstas dependientes de la semántica de tareas Ada y por tanto gestionadas internamente en su librería de tiempo de ejecución.

Sin añadir funcionalidad a la interfaz descrita en este capítulo, resulta posible informar a la tarea planificadora de las circunstancias anteriormente mencionadas en el caso de que así se desee, aunque para ello sea necesaria la colaboración de las tareas planificadas. Esta colaboración podría consistir en la inclusión en su código de llamadas a procedimientos para la invocación específica del planificador (descritos en el apartado 4.4.5), los cuales servirían para notificarle la ocurrencia de las citadas circunstancias.

4.4.5. Invocación explícita del planificador

La interfaz facilita el procedimiento `Invoke_Scheduler` que permite a una tarea invocar de forma explícita a su planificador, enviándole además un valor numérico que puede ser utilizando para indicar la razón de la invocación. Como consecuencia de la ejecución de este procedimiento el sistema operativo generará un evento de tipo `EXPLICIT_CALL` para la tarea planificadora y suspenderá la tarea planificada.

```
procedure Invoke_Scheduler (User_Event_Code : in Integer);
```

En muchas circunstancias puede resultar útil que la tarea planificada invoque al planificador enviándole además información adicional más compleja que un simple valor numérico y, tal vez, recibiendo algún tipo de información como respuesta por su parte. Con este fin, la interfaz proporciona el paquete genérico `Explicit_Scheduler_Invocation`, el cual mostramos a continuación:

```
generic
   type Message is private;
   type Scheduler_Reply is private;
package Explicit_Scheduler_Invocation is

   procedure Invoke (Msg : in Message);
   procedure Invoke (Msg   : in  Message;
                     Reply : out Scheduler_Reply);

   function Get_Message (Event : in Scheduling_Event)
                         return Message;

   function Set_Reply (Reply : in Scheduler_Reply);

end Explicit_Scheduler_Invocation;
```

La ejecución por parte de una tarea planificada de cualquiera de las dos versiones del procedimiento `Invoke`, provocará su suspensión y la generación para su planificador de un evento del tipo `EXPLICIT_CALL_WITH_DATA`.

El parámetro genérico `Message` es la información adicional o mensaje que se desea enviar a la tarea planificadora, la cual podrá ser obtenida por ésta mediante la función `Get_Message` de este mismo paquete. La invocación de `Get_Message` para un evento cuyo tipo no sea `EXPLICIT_CALL_WITH_DATA` provocará que se eleve la excepción `POSIX_Error`.

Por su parte, el parámetro genérico `Scheduler_Reply` describe la estructura de la información de respuesta que recibirá una tarea planificada en el parámetro `Reply` del procedimiento `Invoke`. Antes de reactivar una tarea planificada que ha ejecutado la versión del procedimiento `Invoke` con parámetro `Reply`, el planificador debe haber configurado la respuesta mediante el procedimiento `Set_Reply`. En el caso de que la tarea sea activada sin que el planificador haya ejecutado `Set_Reply`, se elevará la excepción `POSIX_Error` en la tarea planificada.

4.4.6. Relojes y temporizadores de tiempo de CPU

En la actualidad, ni el lenguaje Ada ni el “binding” de POSIX para Ada (estándar POSIX.5b) incluyen ningún mecanismo para utilizar relojes y temporizadores de tiempo de CPU, aunque en los procesos de revisión de ambos estándares que se están llevando a cabo actualmente está siendo considerada su inclusión.

El disponer de este servicio puede resultar muy útil para muchos tipos de aplicaciones, y en particular para la implementación de algoritmos de planificación, puesto que en muchos de ellos el tiempo de ejecución de una tarea se encuentra limitado. Esto ocurre por ejemplo en los algoritmos de reparto de tiempo de CPU o en muchas de las políticas que permiten planificar servidores aperiódicos.

A la espera de que se estandarice alguna solución a nivel del lenguaje Ada o del estándar POSIX, hemos optado por realizar una modificación mínima del paquete estándar

POSIX_Timers para permitir el uso de los relojes y temporizadores de tiempo de CPU. Esta modificación ha consistido en añadir un nuevo tipo de reloj que se suma al único existente (Clock_Realtime) y que se ha denominado Clock_Task_Cputime_Id (igual que su equivalente definido en el POSIX.1).

```
Clock_Task_Cputime_Id : constant Clock_Id;
```

Cuando este reloj es utilizado en una operación de gestión de relojes (Set_Time, Get_Time, Get_Resolution) o temporizadores (Create_Timer), es interpretado como el reloj de tiempo de CPU de la tarea que ejecuta dicha operación.

El otro añadido al paquete POSIX_Timers es la función Get_Cpuclock_Id, la cual permite obtener el reloj de tiempo de ejecución de una tarea. La interfaz de esta función es:

```
function Get_Cpuclock_Id (T : Ada.Task_Identification.Task_Id)
                          return Clock_Id;
```

La inclusión de los relojes y temporizadores de tiempo de ejecución en el paquete POSIX_Timers ha resultado extremadamente sencilla, ya que se apoya directamente en la funcionalidad equivalente descrita en el estándar POSIX.1 e implementada en MaRTE OS.

4.5. Ejemplos de uso de la interfaz

4.5.1. Planificador EDF/CBS

En este ejemplo mostraremos la implementación de un algoritmo de planificación de tareas periódicas extensamente utilizado y relativamente complejo como es el “Earliest Deadline First” (EDF), conjuntamente con la política de planificación de tareas esporádicas “Constant Bandwidth Server” (CBS). Ambas políticas fueron explicadas en detalle en el apartado 1.2 de la introducción de esta memoria. Para la implementación se ha utilizado la versión C de la interfaz para planificación definida por el usuario presentada en este capítulo.

En nuestro ejemplo, el thread planificador es informado de la llegada de un nuevo trabajo para un thread CBS mediante una señal (SEÑAL_NUEVO_TRABAJO). Con el objeto de asegurar que los threads de política CBS no sobrepasan el ancho de banda asignado debe limitarse su tiempo de ejecución. Para ello, el thread planificador asocia un temporizador POSIX de tiempo de ejecución a cada thread CBS. Nuestra implementación aprovecha la característica de las señales POSIX de tiempo real que permite asociarlas un campo de información adicional. Así, aunque la señal generada por la expiración de cualquiera de los temporizadores es siempre la misma (SEÑAL_FIN_CAPACIDAD) su campo de información asociada permite identificar el thread que ha agotado su capacidad de ejecución.

El planificador debe ser informado cuando sus threads finalizan el trabajo correspondiente a su activación actual. Esta notificación se realiza mediante la invocación explícita del planificador utilizando la función posix_appsched_invoke_scheduler().

El planificador EDF/CBS gestiona una lista en la que se encuentran registradas todos los threads planificados. Cada uno de los threads EDF en dicha lista puede encontrarse en los estados ACTIVO (cuando se encuentra ejecutando o en disposición de hacerlo) o ESPERANDO_ACTIVACIÓN (cuando el thread ha finalizado su trabajo actual y se encuentra esperando su próxima activación), por su parte, los threads de política CBS pueden encontrarse en los estados ACTIVO u OCIOSO (este último en el caso de que no tengan trabajos pendientes).

El pseudocódigo presentado no detalla la gestión de dicha lista (así como de otras estructuras internas del planificador), por no afectar a la estructura general del planificador ni a la utilización que éste hace de la interfaz para planificación definida por el usuario.

El planificador realiza la gestión de la lista de tareas planificadas invocando la función `planifica()` tras la llegada de cada uno de los eventos. Dicha función toma como parámetro de entrada la hora actual y, basándose en ella, pone en estado `ACTIVO` todos los threads para los que se haya alcanzado su instante de activación. Posteriormente, obtiene de entre todos los threads activos aquel con un plazo de ejecución más corto y calcula el instante de activación más cercano de entre todos los threads que continúan en estado `ESPERANDO_ACTIVACIÓN`.

El pseudocódigo del thread planificador es el siguiente:

```
void *planificador_edf_cbs (void *arg)
{
   ...;
   while (1) {
      planifica (&thread_siguiente,
                 &próxima_activación,
                 hora_actual);
      /* Acciones de activación y suspensión de threads */
      if (thread_siguiente != thread_actual) {
         if (thread_siguiente != NULL)
            posix_appsched_actions_addactivate (&acciones,
                                                thread_siguiente);
         if (thread_actual != NULL)
            posix_appsched_actions_addsuspend (&acciones,
                                               thread_actual);
         thread_actual = thread_siguiente;
      }
      /* Ejecuta acciones de planificación */
      posix_appsched_execute_actions (&acciones,
                                      &señales_esperadas,
                                      &próxima_activación,
                                      &hora_actual,
                                      &evento);
      /* Procesa eventos de planificación */
      switch (evento.event_code) {
         case POSIX_APPSCHED_NEW :
            // Obtiene parámetros de planificación específicos
            pthread_getappschedparam (...);
            if (thread de política CBS) {
               Crea temporizador de tiempo de ejecución con el
                  identificador de thread como información asociada;
            }
            Añade el thread a la lista de threads planificados;
            Añade acción de aceptación a la lista de acciones;
            break;
         case POSIX_APPSCHED_TERMINATE :
            if (thread de política CBS)
               Elimina temporizador de tiempo de ejecución;
            Elimina el thread de la lista de threads planificados;
            break;
```

```
        case POSIX_APPSCHED_EXPLICIT_CALL :
           // Un thread ha finalizado el trabajo correspondiente
           // a la activación actual
           if (thread de política CBS) {
              thread.numero_de_trabajos_pendientes--;
              if (thread.numero_de_trabajos_pendientes == 0)
                 thread.estado = OCIOSO;
           } else { // thread de política EDF
                 thread.estado = ESPERANDO_ACTIVACIÓN;
                 Calcula siguiente instante de activación y
                                                    de fin plazo;
           }
           break;
        case POSIX_APPSCHED_SIGNAL :
           switch (señal recibida) {
              case SEÑAL_FIN_CAPACIDAD :
                 Obtiene el identificador de thread de la
                                 información asociada con la señal;
                 thread.fin_plazo += thread.period;
                 thread.capacidad = thread.capacidad_máxima;
                 Arma temporizador para expirar en 'thread.capacidad'
                                                      segundos;
                 break;
              case SEÑAL_NUEVO_TRABAJO :
                 thread.numero_de_trabajos_pendientes++;
                 if (thread.estado == OCIOSO) {
                    thread.estado = ACTIVO;
                    if (el trabajo consumiría demasiado
                                                 ancho de banda) {
                       thread.fin_plazo = hora_actual
                                                 + thread.periodo;
                       thread.capacidad = thread.capacidad_inicial;
                       Arma temporizador para expirar en
                                       'thread.capacidad' segundos;
                    }
                 }
           }
           break;
        case POSIX_APPSCHED_TIMEOUT :
           // Se ha alcanzado el instante de activación de un thread
           // No es necesario hacer nada, ya que los threads serán
           // planificados por 'planifica ()'
           break;
     } // switch
   } // while (1)
}
```

La estructura de un thread planificado es:

```
void * edf_or_cbs_thread (void * arg)
{
   while (1) {
      // Realiza el trabajo útil
      ...;
```

```
      // Informa a su planificador que ha terminado el
      //               trabajo correspondiente a la activación actual
      posix_appsched_invoke_scheduler ();
   }
}
```

El planificador necesita conocer el periodo y el plazo de los threads de política EDF, así como la capacidad máxima de ejecución y el periodo de los threads CBS. Para almacenar dichos parámetros de planificación que son específicos del algoritmo de planificación definido por la aplicación definimos el tipo `parametros_edf_cbs_t`:

```
typedef struct {
   int es_un_thread_cbs;
   struct timespec plazo, periodo, capacidad_máxima;
} parametros_edf_cbs_t;
```

Para informar al sistema de los parámetros de planificación específicos de un thread EDF o CBS se asignará una estructura de tipo `parametros_edf_cbs_t` a su atributo `appsched_param` utilizando la función `pthread_attr_setappschedparam()`. Posteriormente, y tal y como mostrábamos en el pseudocódigo del planificador (en la parte dedicada al tratamiento del evento `POSIX_APPSCHED_NEW`), éste podrá obtener los citados parámetros mediante la función `pthread_getappschedparam()`.

Finalmente, el pseudocódigo del thread principal que crea el thread planificador y otros dos threads uno de política EDF y otro CBS es el siguiente:

```
int main ()
{
   ...;

   // Configura el objeto de atributos para el thread planificador
   Asigna política FIFO;
   Asigna tipo de thread “planificador de aplicación”;
   // Crea el thread planificador
   pthread_create (objeto de atributos, planificador_edf_cbs);

   // Configura el objeto de atributos para un thread EDF
   Asigna política “definida por la aplicación”;
   Asigna tipo de thread “regular”;
   Asigna planificador_edf_cbs como su planificador;
   Asigna periodo y plazo en los parámetros de planificación
                definidos por la aplicación (parametros_edf_cbs_t);
   // Crea el thread EDF
   pthread_create (objeto de atributos, thread_edf);

   // Configura el objeto de atributos para un thread EDF
   Asigna política “definida por la aplicación”;
   Asigna tipo de thread “regular”;
   Asigna planificador_edf_cbs como su planificador;
   Asigna periodo y capacidad máxima en parámetros de planificación
                definidos por la aplicación (parametros_edf_cbs_t);
   // Crea el thread CBS
   pthread_create (objeto de atributos, thread_cbs);
}
```

Como puede apreciarse en el pseudocódigo mostrado, la utilización de la interfaz no complica de forma significativa el código, permitiendo la escritura de planificadores con una estructura clara y fácil de seguir. Su utilización tampoco supone un aumento considerable del número de líneas de código. Así, la implementación completa del ejemplo anterior consta de 190 líneas de código. Este valor incluye el planificador y todas las funciones auxiliares (incluidas las correspondientes a la gestión de listas, etc.).

4.5.2. Planificador de tareas esporádicas

Con este segundo ejemplo pretendemos ilustrar la utilización de la interfaz Ada para planificación definida por la aplicación que ha sido presentada en este capítulo. Hemos elegido para ello la política de planificación de servidor esporádico, la cual fue descrita en el apartado 1.2 de la introducción de esta memoria. En el ejemplo la política definida por la aplicación se integra con las políticas estándar, de manera que las prioridades alta y baja asignadas a los servidores esporádicos (y cualquiera de las comprendidas entre ambos valores) pueden también ser utilizadas por otras tareas planificadas bajo la política estándar “FIFO con prioridades”.

Esta misma técnica de integración de las políticas definidas por la aplicación con las directamente soportadas por el sistema operativo podría ser utilizada con otras políticas de planificación, especialmente con otros tipos de servidores aperiódicos. El que hayamos elegido precisamente una política ya implementada en el núcleo de MaRTE OS nos permitirá en el apartado 4.7 evaluar la penalización introducida por nuestra interfaz frente a una política equivalente implementada en el núcleo del sistema operativo.

Para la implementación del planificador de usuario, en primer lugar definimos un tipo de datos para los parámetros de planificación específicos del algoritmo. Posteriormente instanciamos el paquete genérico `Application_Defined_Policy` definido en el paquete `POSIX_Application_Scheduling` utilizando el citado tipo:

```
type Parametros_SE is record
  Prioridad_Alta     : System.Any_Priority;
  Prioridad_Baja     : System.Any_Priority;
  Periodo_De_Recarga : Duration;
  Capacidad_Inicial  : Duration;
end record;

package Politica_SE is
  new POSIX_Application_Scheduling.Application_Defined_Policy
                                     (Parameters => Parametros_SE);
```

Con cada tarea planificada, el planificador asocia una estructura de control utilizando el paquete Ada estándar `Ada.Task_Attributes`. En ella se almacenan de forma permanente los parámetros de planificación específicos del algoritmo junto con otros datos necesarios para gestionar el estado de cada tarea, como son el instante en que se produjo su última activación, el tiempo de ejecución en el que finaliza su capacidad y el temporizador de tiempo de ejecución utilizado para detectar el fin de la capacidad.

```
type Datos_Tarea_SE is record
  --  Parámetros de planificación específicos del algoritmo
  Prioridad_Alta     : System.Any_Priority;
  Prioridad_Baja     : System.Any_Priority;
  Periodo_De_Recarga : Duration;
```

```
    Capacidad_Inicial  : Duration;
    --  Estado tarea
    Capacidad_Agotada          : Boolean;
    Instante_De_Activación     : POSIX.Timespec;
    Límite_Tiempo_CPU          : POSIX.Timespec;
    Temporizador_Fin_Capacidad : POSIX_Timers.Timer_Id;
   end record;

   package Atributos_Tarea_SE is
     new Ada.Task_Attributes (Attribute => Datos_Tarea_SE);
```

Cuando una tarea es aceptada por el planificador, éste le asocia un temporizador POSIX de tiempo de ejecución cuyo identificador se almacena en el campo `Temporizador_Fin_Capacidad`. El algoritmo se encarga de mantener dicho planificador programado para expirar cuando se alcance el límite de capacidad de ejecución de la tarea (indicado por el valor del campo `Límite_Tiempo_CPU`). Al igual que hacíamos en el ejemplo anterior del planificador EDF/CBS, el campo de información asociado con la señal generada por el temporizador permite acceder a la estructura de control (`Datos_Tarea_SE`) de la tarea que ha agotado su capacidad de ejecución.

Además de los eventos temporales debidos al agotamiento de la capacidad de ejecución, en la política de servidor esporádico se utilizan otros eventos para indicar los instantes en los que debe producirse una operación de recarga de parte o toda la capacidad de ejecución de una tarea esporádica. Para la gestión de dichos eventos, nuestro planificador utiliza una cola (`Cola_Eventos_De_Recarga`) en la que éstos se encuentran ordenados de mayor a menor urgencia. En el caso de que haya algún evento de recarga pendiente (la cola `Cola_Eventos_De_Recarga` no se encuentra vacía), se utiliza el procedimiento `Execute_Actions_With_Timeout` utilizando como tiempo límite el instante de expiración del evento de recarga que ocupa la cabeza de la cola. Cada evento se representa mediante un registro que contiene el instante de tiempo absoluto en el que debe producirse la recarga, la cantidad de capacidad a recargar y un puntero a la estructura de control (`Datos_Tarea_SE`) de la tarea a recargar. La programación de un evento de recarga se realiza llamando al procedimiento `Planifica_Operación_Recarga`, el cual se encarga de dar valor a dichos campos y de encolarle por orden de urgencia en la cola `Cola_Eventos_De_Recarga`.

El pseudocódigo de la tarea planificadora se muestra a continuación:

```
task body Tarea_Planificadora is
begin
  --  Lazo de atención de eventos
  loop
    --  Ejecuta acciones de planificación y espera siguiente evento
    if hay algún evento de recarga pendiente then
      Execute_Actions_With_Timeout (Acciones,
                                    Señales_Esperadas,
                                    Evento,
                                    Cola_Eventos_De_Recarga.Cabeza,
                                    Hora_Actual);
    else
      Execute_Actions (Acciones,
                       Señales_Esperadas,
                       Evento,
                       Hora_Actual);
```

```
    end if;
    case Get_Event_Code (Evento) is
      when NEW_TASK =>
        --  Obtiene los parámetros de planificación específicos
        Politica_SE.Get_Parameters (Get_Task (Evento), ...);
        --  Asigna los datos asociados a la tarea
        Nueva_Tarea := ...;
        Atributos_Tarea_SE.Set_Value (Get_Task (Evento),
                                      Nueva_Tarea);
        --  Arma temporizador de fin de capacidad
        POSIX_Timers.Arm_Timer
                          (Nueva_Tarea.Temporizador_Fin_Capacidad,
                           Nueva_Tarea.Límite_Tiempo_CPU);
        --  Acepta y activa tarea
        Add_Accept (Acciones, Get_Task (Evento));
        Add_Activate (Acciones, Get_Task (Evento));

      when TIMEOUT => --  Recarga de capacidad de ejecución
        Cola_Eventos_De_Recarga.Extrae_Cabeza;
        -- Realiza la recarga
        if Tarea_Recargada.Capacidad_Agotada then
          Tarea_Recargada.Límite_Tiempo_CPU :=
                Tiempo de ejecución consumido + Cantidad_Recargada;
          Tarea_Recargada.Capacidad_Agotada := False;
          Sube prioridad a Tarea_Recargada.Prioridad_Alta;
          Tarea_Recargada.Instante_De_Activación := Hora_Actual;
        else
          Tarea_Recargada.Límite_Tiempo_CPU :=
                               Tarea_Recargada.Límite_Tiempo_CPU
                               + Cantidad_Recargada;
        end if;
        --  Rearma temporizador de fin de capacidad
        POSIX_Timers.Arm_Timer
                        (Tarea_Recargada.Temporizador_Fin_Capacidad,
                         Tarea_Recargada.Límite_Tiempo_CPU);

      when SIGNAL => -- Fin de capacidad de ejecución
        Obtiene la estructura de control de la tarea a partir de
                                la información asociada con la señal;
        --  Programa la operación de recarga
        Planifica_Operación_Recarga
          (Instante => Tarea_Agotada.Instante_De_Activación
                       + Periodo_De_Recarga,
           Cantidad => tiempo ejecutado desde la última activación);

        Baja prioridad a Tarea_Agotada.Prioridad_Baja;
        Tarea_Agotada.Capacidad_Agotada := True;

      when READY =>
        Tarea_Activa :=
                        Atributos_Tarea_SE.Value (Get_Task (Evento));
        if not Tarea_Activa.Capacidad_Agotada then
          Tarea_Activa.Instante_De_Activación := Hora_Actual;
        end if;
```

```
      when BLOCK =>
        Tarea_Bloqueada :=
                       Atributos_Tarea_SE.Value (Get_Task (Evento));
        if not Tarea_Bloqueada.Capacidad_Agotada then
          --  Programa la operación de recarga
          Planifica_Operación_Recarga
             (Instante => Tarea_Bloqueada.Instante_De_Activación
                          + Periodo_De_Recarga,
              Cantidad => tiempo ejecutado desde la
                                                 última activación);
        end if;

      when others => null;
    end case;
  end loop;
end Tarea_Planificadora;
```

A continuación mostramos el pseudocódigo correspondiente a una tarea esporádica. Como puede apreciarse, se trata de una tarea Ada normal que requiere ser planificada por el planificador de aplicación mediante la llamada al procedimiento `Politica_SE.Change_Task_Policy`. En la misma llamada envía al planificador los parámetros bajo los que requiere ser planificada. En el caso de que fuera rechazada se elevaría la excepción `POSIX_Error`.

```
task body Tarea_Esporádica is
begin
  --  Requiere ser planificada por la tarea planificadora
  Politica_SE.Change_Task_Policy
                        (Tarea_Planificadora'Identity,
                         Parametros_SE'(Prioridad_Alta => 10,
                                        Prioridad_Baja  => 6,
                                        Periodo_De_Recarga => 0.1,
                                        Capacidad_Inicial => 0.05));
  loop
    -- Realiza el trabajo útil
    ...;
  end loop;

exception
  when POSIX_Error =>
    Error: tarea no aceptada por el planificador;
end Tarea_Esporádica;
```

La implementación del ejemplo anterior consta de aproximadamente 250 líneas de código. Este valor incluye el planificador y todas las funciones auxiliares (incluidas las correspondientes a la gestión de la cola de eventos temporales, etc.).

4.6. Implementación en MaRTE OS

Uno de los principales motivos para los que fue desarrollado el sistema operativo MaRTE OS, era para que sirviera de plataforma sobre la que implementar la funcionalidad descrita en este capítulo. Su inclusión en MaRTE OS nos ha permitido obtener experiencia sobre otro

importante factor a la hora de valorar la validez de nuestro marco para planificación flexible, como es su dificultad de implementación en un sistema operativo real, en particular en uno conforme con el perfil de sistema mínimo de tiempo real como es MaRTE OS.

La parte central de la funcionalidad se ha implementado en cuatro paquetes:

- `Kernel.Application_Scheduler`: contiene el procedimiento que realiza las acciones correspondientes a la generación de un evento de planificación, básicamente consistentes en el encolado del evento y la posible activación de la tarea planificadora en el caso de que estuviera suspendida a la espera un nuevo evento.
- `Kernel.Appsched_Data`: en este paquete se definen los códigos de eventos, el tipo utilizado para representar los eventos de planificación y la estructura de datos que MaRTE OS asocia con cada tarea planificadora.
- `Kernel.Tasks_Operations.Application_Scheduler`: contiene las funciones descritas en la interfaz en la forma correspondiente a su versión C (aunque por supuesto escritas en Ada).
- `Kernel.Signals.Application_Scheduler`: se definen las operaciones específicas de las tareas planificadoras que están relacionadas con las señales.

También ha sido necesario modificar otros seis paquetes ya existentes. La mayor parte de las modificaciones realizadas tienen que ver con la inclusión del código utilizado para la generación de los eventos de planificación, afectando principalmente a las partes del núcleo encargadas de gestionar los cambios de estado de las tareas.

En total, el número de instrucciones añadidas es alrededor de 660, muy similar a las instrucciones correspondientes a la implementación de los mutexes (670 aproximadamente). En base al número de instrucciones, la implementación de la interfaz para planificación definida por la aplicación constituye un 4.2% del tamaño total del núcleo.

En cuanto a las nuevas estructuras de datos utilizadas, la más importante es la que se asocia con cada tarea planificadora, que está básicamente compuesta por los siguientes campos:

- El reloj utilizado por la función de ejecución de las acciones de planificación.
- El tipo de tiempo límite (absoluto o relativo) utilizado por la función de ejecución de las acciones de planificación.
- La información de respuesta que ha de ser enviada a un thread planificado que invoca explícitamente al planificador.
- La máscara de eventos de planificación.
- La lista de tareas planificadas.
- La cola FIFO de eventos de planificación pendientes.

La lista de tareas planificadas se utiliza para obtener la prioridad activa del planificador, la cual se revisa cada vez que una de sus tareas planificadas hereda o pierde una prioridad debido al uso de un recurso del sistema. Cuando la prioridad heredada por una de sus tareas es mayor que la del planificador, la prioridad de este último debe ser elevada. Por el contrario, cuando una tarea pierde una prioridad igual a la prioridad activa del planificador, deberá calcularse la nueva prioridad activa de este último como el mayor valor de entre: su propia prioridad base, la mayor de las prioridades heredadas por sus tareas y la mayor de las prioridades heredadas por el propio planificador.

Con la gestión de la prioridad del planificador descrita anteriormente no basta para asegurar que una tarea planificadora ejecute siempre antes que cualquiera de sus tareas planificadas, ya que aún podría darse una situación en que una tarea planificada se encontrara antes que su planificador en la cola FIFO correspondiente al nivel de prioridad de ambas. Para resolver esta situación de acuerdo con lo expuesto en la descripción del modelo (apartado 4.2), en la rutina de cambio de contexto se introduce una comprobación adicional, de forma que cuando una tarea planificada por la aplicación va a ser elegida para ejecutar se comprueba si su planificador también se encuentra activo. En caso afirmativo, será el planificador el que sea escogido como próxima tarea a ejecutar.

4.7. Prestaciones

En el diseño de la interfaz para planificación definida por el usuario hemos perseguido como objetivos fundamentales la compatibilidad con el estándar POSIX y la generalidad, esto es, la capacidad de implementar la mayor variedad de algoritmos de planificación posible. Aún logrando estos dos objetivos citados, la interfaz vería en gran medida limitada su utilización práctica si su uso supusiera una importante penalización temporal para las aplicaciones.

En este apartado se pretende evaluar la penalización sufrida por los algoritmos de planificación como consecuencia de la utilización de nuestra interfaz. Esta penalización, debida a la necesidad de ejecutar el thread planificador cada vez que se produce un evento de planificación, se divide en dos partes:

- El tiempo empleado en la activación de un thread planificador suspendido en la función `posix_appsched_execute_actions()` para el que ha sido generado un evento de planificación.
- El tiempo necesario para que un thread planificado que ha sido activado como consecuencia de la ejecución de la función `posix_appsched_execute_actions()` comience su ejecución mientras el thread planificador queda suspendido a la espera de un nuevo evento.

Ambos tiempos son independientes de las políticas de planificación implementadas, no pretendiéndose aquí medir los tiempos debidos al propio algoritmo (aplicación de las reglas de cada política, búsqueda de tarea más prioritaria, etc). Esas operaciones podrán ser más o menos rápidas dependiendo de cada implementación particular, pero en todo caso serán independientes del entorno en el que se ejecute el algoritmo, ya haya sido éste implementado como thread planificador, como parte del núcleo o bien utilizando cualquiera de las otras soluciones para planificación flexible citadas en el apartado 2.3, “Soluciones para la planificación a nivel de aplicación”.

La tabla 4.7 muestra el tiempo[1] empleado en la activación de un thread planificador suspendido en la función `posix_appsched_execute_actions()` para el que se genera un evento de planificación. El primero de los valores corresponde al caso más sencillo en el que el thread planificador, que no se encontraba esperando ninguna señal ni con el tiempo límite activo, recibe un evento de tipo `POSIX_APPSCHED_EXPLICIT_CALL` (provocado por una invocación explícita desde uno de sus threads planificados). La duración en los otros dos casos es mayor ya que ambos implican la atención por parte del núcleo de una interrupción del temporizador del sistema y además, en el caso de `POSIX_APPSCHED_SIGNAL`, de la entrega de la señal generada.

1. Todas las medidas presentadas en este apartado han sido realizadas en un ordenador con procesador Pentium III a 1.1GHz.

Tabla 4.7: Prestaciones de la interfaz C. Activación del planificador.

Evento recibido	Tiempo (μs)
`POSIX_APPSCHED_EXPLICIT_CALL`	0.57
`POSIX_APPSCHED_TIMEOUT`	0.87
`POSIX_APPSCHED_SIGNAL` (señal debida a la expiración de un temporizador de tiempo de ejecución)	0.97

Para valorar la eficiencia de la implementación, podemos comparar los tiempos mostrados en la tabla con algunos de los valores correspondientes a las prestaciones generales del sistema que mostrábamos en el apartado 3.8, “Prestaciones y tamaño de las aplicaciones”. Buscaremos para la comparación situaciones de complejidad similar, lo que nos permitirá evaluar la penalización debida a nuestra interfaz en lo que se refiere a la generación y entrega de los eventos. Así, el primero de los valores podría compararse con un cambio de contexto entre threads de política `SCHED_FIFO` como consecuencia de una cesión del procesador mediante la función `sched_yield()`, como mostrábamos en la tabla 3.9, este tiempo es de 0.44μs. Una situación equivalente a la activación del planificador tras la expiración del tiempo límite (mostrada en la segunda fila de la tabla) sería la activación de un thread `SCHED_FIFO` al finalizar una suspensión temporizada (0.75μs como se muestra en la tabla 3.13). El tercer valor de la tabla 4.7 podemos compararlo con la activación de un thread tras la expiración de un temporizador de tiempo de ejecución. Este tiempo (mostrado en la tabla 3.12) es de 0.9μs. Por consiguiente, el tiempo empleado en la gestión de los eventos de planificación no es excesivo, suponiendo un aumento del tiempo de las operaciones que oscila entre el 9% y el 29%.

La tabla 4.8 muestra las mismas medidas que la anterior pero realizadas para un programa que utiliza la versión Ada de la interfaz. Como podemos apreciar, los valores obtenidos son superiores que sus equivalentes C. Esto se debe a que la forma de las funciones implementadas en el núcleo de MaRTE OS coincide con la descrita para la versión C de la interfaz, mientras que la interfaz Ada se construye como una capa más externa basada en las funciones directamente proporcionadas por el núcleo.

Tabla 4.8: Prestaciones de la interfaz Ada. Activación del planificador

Evento recibido	Tiempo (μs)
`EXPLICIT_CALL`	0.7
`TIMEOUT`	1.2
`SIGNAL` (señal debida a la expiración de un temporizador de tiempo de ejecución)	1.5

También es interesante mencionar, que las medidas mostradas en la tabla 4.8 son inferiores al tiempo de cambio de contexto entre tareas Ada de política FIFO con prioridades. Esto es debido a que en las valores presentados en este apartado no interviene apenas la librería de tiempo de ejecución de GNAT.

En la tabla 4.9 se muestra la segunda parte de la penalización debida al uso de la interfaz, esto es, el tiempo necesario para que un thread planificado que ha sido activado como consecuencia de la ejecución de la función `posix_appsched_execute_actions()` comience su

ejecución. La tabla muestra distintos valores dependiendo de la funcionalidad de `posix_appsched_execute_actions()` utilizada.

Tabla 4.9: Prestaciones de la interfaz C. Activación de un thread planificado.

Descripción	Tiempo (μs)
Activación de un thread planificado tras la ejecución de la función `posix_appsched_execute_actions()`	0.5
Activación de un thread planificado tras la ejecución de la función `posix_appsched_execute_actions()` utilizando el tiempo límite	0.9
Activación un thread planificado tras la ejecución de la función `posix_appsched_execute_actions()` esperando un conjunto de señales	1.4
Activación de una tarea planificada tras la ejecución de la función `posix_appsched_execute_actions()` esperando un conjunto de señales y utilizando el tiempo límite	1.7

La tabla 4.10 muestra las mismas medidas que la anterior pero realizadas para un programa que utiliza la interfaz Ada para planificación definida por la aplicación.

Tabla 4.10: Prestaciones de la interfaz Ada. Activación de tarea planificada.

Descripción	Tiempo (μs)
Activación de una tarea planificada tras la ejecución del procedimiento `Execute_Actions()`	0.9
Activación de una tarea planificada tras la ejecución del procedimiento `Execute_Actions_With_Timeout()`	1.4
Activación de una tarea planificada tras la ejecución del procedimiento `Execute_Actions()` cuando se espera un conjunto de señales	2.1
Activación de una tarea planificada tras la ejecución del procedimiento `Execute_Actions_With_Timeout()` cuando se espera un conjunto de señales	2.3

Finalmente analizaremos las prestaciones de la interfaz en los casos concretos de los planificadores presentados como ejemplos en los apartados 4.5.1 y 4.5.2.

La tabla 4.11 muestra las prestaciones del planificador EDF/CBS cuyo pseudocódigo fue presentado en el apartado 4.5.1. El caso estudiado corresponde a un cambio de contexto entre un thread CBS que consume su capacidad de ejecución y un thread EDF que es puesto en estado activo por el planificador y toma la CPU. Las tres primeras medidas corresponden al tiempo consumido en las distintas partes del cambio de contexto, mientras que la cuarta corresponde a la suma de todas ellas, esto es, representa el tiempo transcurrido entre que el thread CBS es expulsado de la CPU hasta que el thread EDF comienza su ejecución (habiendo ejecutado por medio el thread planificador). En este caso, el tiempo de cambio de contexto (4.0μs) representaría únicamente una sobrecarga de 0.8% para un thread periódico de 1KHz, asumiendo

dos cambios de contexto por ejecución. La sobrecarga introducida no parece en absoluto prohibitiva, y por supuesto, resultaría aún menor para threads de menor frecuencia.

Tabla 4.11: Prestaciones planificador EDF/CBS.

Descripción	Tiempo (μs)
Activación del planificador tras la expiración del temporizador de tiempo de ejecución	1.0
Algoritmo de planificación	1.3
Activación del thread EDF	1.7
Tiempo total de cambio de contexto entre el thread EDF y el thread CBS	4.0
Cambio de contexto después de la expiración de temporizador de tiempo de ejecución	0.9

La penalización debida a la utilización de la interfaz es 2.7μs. Este valor se obtiene sumando el tiempo empleado en la activación del planificador (1.0μs) mas el debido a la ejecución de la acción de planificación que causa la activación del thread EDF (1.7μs). En una implementación de este algoritmo de planificación sin hacer uso de nuestra interfaz, sería necesario, al menos, utilizar un temporizador de tiempo de ejecución. El tiempo de activación de una tarea que recibe una señal generada por uno de esos temporizadores es 0.9μs. Por consiguiente, el tiempo empleado por nuestra interfaz es 3 veces superior al correspondiente al planificador de MaRTE OS. Sin embargo, en este cálculo no se ha tenido en cuenta el tiempo debido al propio algoritmo de planificación. Suponiendo que dicho tiempo fuera el mismo en ambos casos, deberíamos comparar los 4.0μs del cambio de contexto utilizando la interfaz con 2.2μs (resultantes de sumar los 0.9μs debidos al cambio de contexto tras la expiración del temporizador mas los 1.3ms empleados por el algoritmo). Con estos valores resultaría que el algoritmo implementado utilizando nuestra interfaz sería aproximadamente 1.8 veces más lento que uno en el que la interfaz no hubiera sido utilizada.

Un resultado más fiable sobre la penalización que supone la implementación de algoritmos de planificación utilizando nuestra interfaz, frente a su inclusión directa en el núcleo del sistema operativo, puede realizarse utilizando el planificador mostrado en el apartado 4.5.2. En este ejemplo presentábamos el pseudocódigo de una implementación a nivel de aplicación de la política de servidor esporádico, la cual también se haya implementada en el núcleo de MaRTE OS. El ejemplo correspondía a un planificador escrito en Ada, lo que dificulta la evaluación directa de la penalización debida a nuestra interfaz, ya que en las aplicaciones Ada además hay que contar con una penalización adicional, debida a la librería de tiempo de ejecución. Con el objeto de evaluar el impacto de la utilización de nuestra interfaz de una forma más precisa se ha realizado también una versión de este mismo planificador pero escrita en lenguaje C.

En la tabla 4.12 se comparan las tres implementaciones de la política de servidor esporádico realizadas: las implementadas a nivel de aplicación utilizando los lenguajes Ada y C y la existente en el núcleo de MaRTE OS. Los tiempos comparados corresponden a cambios de contexto entre un servidor esporádico y una tarea FIFO que ejecuta con una prioridad intermedia entre las dos prioridades utilizadas por la tarea esporádica. En uno de los casos estudiados, el cambio de contexto se produce como consecuencia de que el servidor esporádico agota su capacidad de ejecución, mientras que en el otro la tarea FIFO es expulsada del

procesador como consecuencia de que el servidor esporádico ha recargado parte de su capacidad de ejecución y ve elevada su prioridad.

Tabla 4.12: Comparación entre tres implementaciones de la política de servidor esporádico.

Implementación	Causa del cambio de contexto	Tiempo (μs)
Ada (utilizando la interfaz para planificación definida por la aplicación)	Tarea esporádica agota su capacidad de ejecución	6.6[a]
	Tarea esporádica recarga parte de su capacidad de ejecución	6.9
C (utilizando la interfaz para planificación definida por la aplicación)	Thread esporádico agota su capacidad de ejecución	3.6
	Thread esporádico recarga parte de su capacidad de ejecución	3.8
Planificador MaRTE OS	Thread esporádico agota su capacidad de ejecución	1.2
	Thread esporádico recarga parte de su capacidad de ejecución	1.3

a. La diferencia entre las versiones Ada y C es debida principalmente a la duración de la operación de cambio de prioridad dinámica proporcionada por el lenguaje Ada (2.4μs aproximadamente).

Como puede apreciarse comparando las prestaciones de la implementación C a nivel de aplicación con las correspondientes a las del planificador de MaRTE OS, el uso de nuestra interfaz provoca que los cambios de contexto sean alrededor de 3 veces más lentos. Desde el punto de vista de la sobrecarga sufrida por un thread periódico de 1KHz de frecuencia, la implementación interna consumiría como máximo el 0.26% del tiempo de ejecución disponible, mientras que la realizada a nivel de aplicación el 0.76%. De estos datos se desprende que la sobrecarga impuesta por nuestra interfaz en este ejemplo considerando un thread periódico de 1KHz es únicamente del 0.5%.

También resultaría interesante comparar las prestaciones de nuestra interfaz con las de otros marcos para planificación a nivel de aplicación existentes en al bibliografía. Esto no es posible en todos los casos, ya que algunas de las soluciones existentes no proporcionan datos acerca de sus prestaciones, por constituir su principal objetivo la flexibilidad, quedando la eficiencia en un segundo plano. En otros casos, los tiempos proporcionados no son comparables con los obtenidos para nuestra interfaz, al tratarse de sistemas que ejecutan los algoritmos dentro del núcleo del sistema operativo y/o constituyen soluciones con mucha menor funcionalidad que nuestra interfaz.

Dos soluciones comparables con la nuestra y que proporcionan datos sobre sus prestaciones son RED-Linux [WAN99] y la basada en la herencia de CPU [FOR96]. En el caso de RED-Linux, la utilización de su planificador de dos niveles supone una penalización para las aplicaciones casi 5 veces mayor que si el algoritmo de planificación hubiera sido implementado en el núcleo del sistema operativo. Para la herencia de CPU, la utilización de su entorno para la planificación a nivel de la aplicación, supone unos tiempos de cambio de contexto aproximadamente 2.3 veces mayores que los debidos a un algoritmo implementado en el mismo sistema operativo. La

comparación de nuestra interfaz con estas dos soluciones resulta más favorable aún si se tiene en cuenta que supera algunas de sus carencias, tal y como se puso de manifiesto la introducción de este capítulo.

A la vista de las prestaciones medidas para nuestra interfaz y de la comparación con los resultados proporcionados por otras soluciones, consideramos que la penalización introducida es pequeña y totalmente asumible. Más aún si se consideran las ventajas que se desprenden de disponer de una interfaz que permite definir planificadores de aplicación de forma flexible y potencialmente portable.

4.8. Conclusiones

Se ha diseñado una interfaz que permite a las aplicaciones definir los algoritmos de planificación bajo los cuales desean planificar sus tareas. Se han presentado dos versiones equivalentes de la citada interfaz, una escrita en lenguaje C y otra en lenguaje Ada, ambas integradas en el conjunto de interfaces POSIX. Con el fin de proceder a su validación, se ha procedido a su implementación en el sistema operativo MaRTE OS (presentado en el capítulo 3 de esta memoria), desarrollándose con ellas varios algoritmos de planificación que han servido para su prueba.

Frente a las carencias detectadas en otras soluciones existentes en la bibliografía (analizadas en el apartado 2.3, “Soluciones para la planificación a nivel de aplicación”), la interfaz desarrollada presenta las siguientes características:

- *Constituye una solución general*, permitiendo la implementación de una amplia variedad de algoritmos de planificación[1]. Esta flexibilidad se debe principalmente a que el planificador es invocado puntualmente tras cada circunstancia que pueda requerir la toma de una decisión de planificación, a que no se imponen restricciones a los parámetros de planificación específicos del algoritmo y a que en la implementación de los algoritmos es posible utilizar otros servicios de la interfaz POSIX, tales como señales y temporizadores (incluidos los de tiempo de ejecución). La versatilidad de la interfaz ha sido comprobada con la realización de los planificadores EDF, CBS y servidor esporádico presentados en el apartado 4.5, “Ejemplos de uso de la interfaz”, así como por la implementación de otras muchas políticas no incluidas en esta memoria, tales como “round-robin”, reparto proporcional, planificación conjunta de tareas de políticas EDF y FIFO con prioridades, ejecutivo cíclico, etc.
- *Permite la implementación de algoritmos de planificación para sistemas multiprocesadores*. Una opción para este tipo de sistemas consiste en utilizar un planificador compuesto por varias tareas planificadoras cada una ejecutando en un procesador. Dichas tareas podrían sincronizarse o compartir información mediante el uso de objetos protegidos o mutexes y variables condicionales. Pero también, gracias a la capacidad de las tareas planificadoras de ejecutar varias acciones de planificación de forma simultánea, un planificador para un sistema multiprocesador podría estar formado por una única tarea planificadora. Esta segunda posibilidad permitiría implementar algoritmos de planificación para sistemas compuestos por un único procesador multipropósito (donde ejecuta el planificador), y varios procesadores de propósito específico (por ejemplo procesadores digitales de señal) en los que ejecutan las tareas planificadas.

1. A fecha de escritura de esta memoria, no se ha encontrado ningún tipo de algoritmos de planificación que no sea implementable con la interfaz presentada.

- *Permite proteger el resto del sistema ante fallos en un algoritmo de planificación*. La interfaz permite que las tareas planificadoras sean ejecutadas en el mismo espacio de direcciones que el resto de las tareas de la aplicación. De esta forma, un fallo en un algoritmo de planificación definido por la aplicación no puede afectar al resto del sistema. Por razones de eficiencia, la interfaz también permite que las tareas planificadoras pueden ser ejecutadas en un entorno distinto al del resto de las tareas de usuario, por ejemplo dentro del núcleo del sistema operativo.
- *Persigue su integración en el estándar POSIX*. A diferencia de las demás soluciones existentes en la bibliografía, la interfaz expuesta en este capítulo se integra con el resto de interfaces que forman el estándar POSIX. El modelo en el que se basa es compatible con las políticas de planificación definidas en los estándares POSIX y Ada 95, lo que hace posible que en un mismo sistema puedan convivir tareas planificadas por la aplicación junto con otras planificadas bajo una de las políticas estándar. Por otra parte, de la compatibilidad con POSIX también se deriva que sea posible escribir algoritmos que se integren con las políticas estándar, como por ejemplo servidores aperiódicos que varíen la prioridad de sus tareas entre dos niveles de prioridad POSIX.

Con las características mencionadas, nuestra interfaz auna las mejores cualidades de las soluciones existentes en la bibliografía, como son la ejecución de los algoritmos en una tarea de usuario, como ocurría en la herencia de CPU o la invocación del algoritmo en cada punto de planificación, como en las soluciones basadas en módulos (S.Ha.R.K y Vassal). A la vez, la interfaz supera las principales carencias de dichas soluciones: falta de generalidad (RED-Linux), dificultad de implementación en sistemas multiprocesadores (herencia de CPU) y falta de aislamiento entre planificadores y sistema operativo (todas las basadas en módulos como son S.Ha.R.K y Vassal).

Además de la mejora en la seguridad del sistema, otra importante ventaja de la ejecución de los algoritmos de planificación desde una tarea de usuario es que el código del algoritmo puede utilizar los servicios proporcionados por el sistema operativo, tales como las interfaces POSIX. En general esto no sería posible en el caso de que el algoritmo hubiera sido implementado utilizando el mecanismo de módulos, ya que en muchos sistemas operativos no está permitida la utilización de tales servicios desde dentro del núcleo.

Como se desprende de los ejemplos presentados en el apartado 4.5, la utilización de la interfaz no complica de forma significativa el código, permitiendo la escritura de planificadores con una estructura clara y fácil de seguir. Su uso tampoco supone un aumento considerable del número de líneas de código necesarias para la escritura de un algoritmo de planificación.

Midiendo las prestaciones de algunos de los algoritmos de planificación implementados y comparándolas con las del planificador de MaRTE OS, se ha calculado la penalización que supone para las aplicaciones el uso de la interfaz. Hemos encontrado que los tiempos empleados por nuestra interfaz para una operación de planificación son aproximadamente 3 veces mayores que los correspondiente al planificador de MaRTE OS. Este aumento, de por sí no muy importante a la vista de la importantes ventajas obtenidas, resulta más claramente asumible si se considera que la sobrecarga debida a la interfaz rondaría el 0.5% del periodo de ejecución de una tarea periódica de 1KHz. La comparación con los resultados proporcionados por otras soluciones para planificación flexible existentes en la bibliografía también resulta favorable, sobre todo si se considera la mayor generalidad de nuestra solución.

5. Interfaz de usuario para la definición de protocolos de sincronización

5.1. Introducción

En el capítulo anterior se presentaba una interfaz para permitir a las aplicaciones definir sus propios algoritmos de planificación. Dicha interfaz, a pesar de resolver muchos de los problemas adolecidos por las demás soluciones existentes en la bibliografía, no constituye aún una solución general al problema de la planificación flexible, ya que únicamente permite la implementación de políticas de planificación, no contemplando la problemática de la sincronización entre las tareas.

La existencia de tareas como entidades independientes no es habitual en los sistemas de tiempo real. Por el contrario, lo normal es que las tareas se relacionen entre si sincronizándose y compartiendo información. En la mayoría de los casos, el acceso a la información compartida debe realizarse de forma mutuamente exclusiva entre las tareas que la utilizan, por lo que resulta necesario definir protocolos de sincronización que rijan el acceso a esos recursos comunes. Como ya se expuso en la introducción de esta memoria, la utilización de protocolos de sincronización inadecuados puede provocar efectos muy perjudiciales para el sistema (bloqueos innecesarios, bloqueos mutuos, inversión de prioridad no acotada, etc.).

En consecuencia, una solución general al problema de la planificación flexible deberá permitir definir, junto a cada algoritmo de planificación, el protocolo de sincronización que mejor se adapte a él. Precisamente la no existencia de un mecanismo general que permita definir los citados protocolos es una de las principales carencias encontradas en la mayoría de las soluciones para planificación definida por la aplicación que fueron analizadas en el apartado 2.3, “Soluciones para la planificación a nivel de aplicación”.

En el presente capítulo se pretende extender la interfaz propuesta en el capítulo anterior de forma que las aplicaciones puedan además definir sus propios algoritmos de sincronización. Los principales requerimientos marcados para la interfaz presentada son:

- La interfaz debe constituir una ampliación de las interfaces POSIX (en sus versiones Ada y C). Allí donde sea posible habrá que utilizar operaciones ya definidas por el estándar POSIX (posiblemente extendiendo su funcionalidad) en lugar de definir otras nuevas.

  Justificación: La integración con POSIX facilitará el uso de la interfaz por programadores familiarizados con este estándar, en número creciente debido a su alto grado de aceptación entre los fabricantes de sistemas operativos.

- La interfaz debe constituir una ampliación de la presentada en el capítulo anterior de esta memoria (4, “Interfaz de usuario para la definición de algoritmos de planificación”).

  Justificación: Se pretende ampliar la funcionalidad soportada por las tareas planificadoras definidas en el anterior capítulo, de forma que sea posible también implementar protocolos de sincronización.

- Deberá ser posible crear mutexes cuyo protocolo vaya a estar definido por una tarea planificadora de aplicación. Estos mutexes serán utilizados para sincronizar tareas planificadas por la aplicación.

  Justificación: Este tipo de mutexes permitirá a la tarea planificadora definir el protocolo de sincronización que mejor se adapte al algoritmo de planificación que implementa.

- Deberá ser posible asignar a los mutexes de protocolo definido por la aplicación parámetros de planificación específicos de dicho protocolo. Esos parámetros específicos podrán ser obtenidos por el planificador en cualquier momento, en particular cuando se le notifica la creación del mutex.

  Justificación: Lo normal será que un protocolo definido por la aplicación requiera parámetros de planificación diferentes de los definidos para los mutexes que utilizan los protocolos estándar.

- Existirá un mecanismo que permita a los planificadores de aplicación aceptar o rechazar la creación de un mutex de protocolo definido por la aplicación. Este rechazo deberá ser notificado tanto a la aplicación como al propio sistema operativo.

  Justificación: En ocasiones un algoritmo de planificación podrá decidir no aceptar la creación de un nuevo mutex, bien porque sus parámetros de planificación son incorrectos o porque considera que el sistema no dispone de suficientes recursos para hacerlo. El sistema operativo deberá tener noticia de este hecho para abortar la creación del mutex. Por su parte la notificación a la aplicación será necesaria para que ésta pueda tomar las acciones que considere oportunas.

- La tarea planificadora podrá ser informada por el sistema operativo de las operaciones que sus tareas planificadas realizan sobre los mutexes de protocolo definido por la aplicación.

  Justificación: Esta información es necesaria puesto que será en esos instantes cuando la tarea planificadora tenga que realizar las acciones de planificación resultantes de la aplicación del protocolo.

- La tarea planificadora debe ser capaz de determinar a qué tarea y en qué momento se realiza la concesión de un mutex libre.

  Justificación: Esto permitirá implementar protocolos de sincronización en los que la concesión de los recursos no depende únicamente del estado del mutex, sino que lo hace también de algún otro parámetro global del sistema. La implementación de tales protocolos sería imposible si el sistema operativo entregara automáticamente un mutex libre a la tarea que trata de tomarlo sin que el planificador pudiera intervenir en esa decisión.

- Las tareas planificadas por la aplicación deberán poder utilizar también mutexes creados con los protocolos definidos en el estándar POSIX.

  Justificación: Estos mutexes se utilizarán para sincronizar tareas planificadas por la aplicación con otras que no lo sean. Además, esta posibilidad permitirá a las tareas planificadas por la aplicación utilizar librerías de propósito general que hagan uso de este tipo de mutexes.

- La tarea planificadora deberá tener la posibilidad de conocer cuando una de sus tareas planificadas ha heredado o perdido una prioridad como consecuencia del uso de un mutex de sistema.

Justificación: Esta información puede permitir al planificador limitar la inversión de prioridad sufrida por las demás tareas que comparten recursos con las tareas por él planificadas.

- Deberá ser posible asociar datos específicos con los mutexes.

  Justificación: La mayoría de los protocolos de sincronización requerirán asociar con cada mutex alguna información específica (cola de tareas bloqueadas, tarea propietaria, etc.). A diferencia de los datos específicos de los threads (funcionalidad ya definida en el estándar POSIX) en este caso no se precisa asociar un valor llave que identifique cada conjunto de datos, puesto que cada mutex sólo puede estar asociado con un planificador.

5.2. Descripción del modelo

La necesidad de que las tareas planificadoras dispongan de algún mecanismo para establecer sus propios protocolos de sincronización, junto con el requisito de compatibilidad con el estándar POSIX, conduce a la definición en el modelo de dos tipos de mutexes:

- *Mutexes con protocolo definido por el sistema*: son los creados con alguno de los protocolos actualmente definidos en el estándar POSIX, esto es, sin herencia de prioridad, protocolo de techo de prioridad inmediato o protocolo de herencia básica de prioridad.
- *Mutexes con protocolo definido por la aplicación*: su comportamiento está definido por la aplicación. Se les puede asociar un conjunto de atributos de planificación específicos del protocolo y que serán interpretados por su tarea planificadora. El sistema se encarga de informar al planificador de las operaciones realizadas sobre este tipo de mutexes de forma que pueda tomar las acciones de planificación apropiadas.

En la figura 5.1 se muestra la utilización de ambos tipos de mutexes para la sincronización de los diferentes tipos de tareas presentados en el capítulo 4. Los mutexes con protocolo definido por la aplicación únicamente pueden ser utilizados para la sincronización entre tareas vinculadas con una misma tarea planificadora. Por su parte, los mutexes con protocolo definido por el sistema operativo pueden ser utilizados para controlar el acceso a recursos compartidos entre:

- Conjuntos de tareas planificadas por diferentes planificadores.
- Varias tareas planificadoras.
- Tareas planificadas por la aplicación y tareas normales planificadas por el sistema.

Por supuesto, siempre que se desee sincronizar tareas pertenecientes a distintos procesos, e independientemente de los tipos de estas, será necesario que el mutex utilizado se encuentre definido dentro de un objeto de memoria compartida.

En la figura 5.1 las flechas discontinuas representan la generación de eventos de planificación que son encolados en el planificador tal y como se explicó en el apartado 4.2.2, “Relación entre el planificador y sus tareas planificadas”. Como puede apreciarse en la figura, todas las operaciones realizadas por las tareas planificadas sobre un mutex, ya sea de aplicación o de sistema, generan eventos. En los siguientes apartados se realiza una discusión detallada de los eventos generados y de su significado, así como de las acciones de planificación relacionadas con los mutexes con protocolo definido por la aplicación.

Como expusimos en el capítulo 4, el modelo adoptado no hace ninguna suposición sobre el espacio de direcciones en el que se encuentran las tareas planificadoras, siendo dependiente de

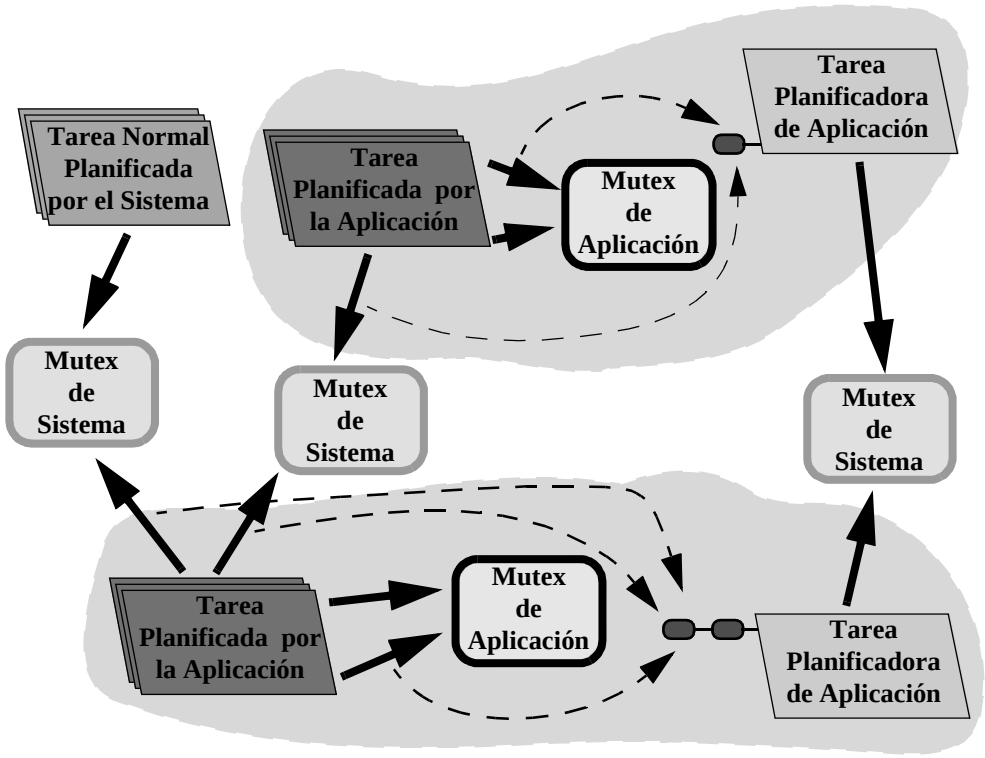

Figura 5.1: Modelo para la sincronización definida por la aplicación

la implementación el que compartan espacio de direcciones con otras tareas o que ejecuten en el propio espacio de direcciones del núcleo del sistema operativo. Por lo tanto, no es posible que una tarea planificadora comparta información con sus tareas planificadas de forma directa ni mediante objetos de memoria compartida. La única forma segura de compartir información entre el planificador de aplicación y sus tareas planificadas es utilizando las funciones para invocación explícita del planificador descritas en los apartados 4.3.5 y 4.4.5.

5.2.1. Uso de mutexes del sistema por parte de las tareas planificadas

Los mutexes con protocolo definido por el sistema se comportan tal y como describe el POSIX, independientemente del tipo de tarea que los utilice: normal, planificada por la aplicación o planificadora. El estándar dice que una tarea tomará el mutex siempre que éste se encuentre libre, y que cuando una tarea libere un mutex, éste será tomado por la más prioritaria de las bloqueadas en él, quedando libre en el caso de que no hubiera ninguna. La prioridad activa de una tarea en posesión de un mutex puede verse aumentada al heredar la prioridad del techo del mutex (si se trata de un mutex de protocolo de techo de prioridad inmediato) o las prioridades de las tareas que se bloqueen en él (si es un mutex de protocolo de herencia básica de prioridad).

Como expusimos anteriormente, las tareas planificadas por la aplicación pueden utilizar mutexes de sistema para sincronizarse con otras tareas. Lo normal será que los protocolos estándar no encajen con la política de dichas tareas (en especial cuando ésta no se base en prioridades). En esos casos, aunque el planificador no puede modificar el comportamiento de los mutexes del sistema, sí que puede saber cúando sus tareas utilizan algún recurso global mediante los eventos de planificación mostrados en la tabla 5.1.

Además de servir para conocer en qué momento una de sus tareas se encuentra en posesión de un mutex de sistema, la información sobre la prioridad heredada puede ser utilizada por la tarea

Tabla 5.1: Eventos debidos al uso de mutexes del sistema

Tipo de Evento	Descripción	Información Adicional
Priority Inherit	Una tarea ha heredado una prioridad de sistema debido al uso de un mutex del sistema.	Prioridad de sistema heredada
Priority Uninherit	Una tarea ha perdido una prioridad de sistema previamente heredada.	Prioridad de sistema perdida

planificadora como indicación de la importancia del recurso. El planificador podrá entonces dar preferencia a la tarea en posesión del recurso sobre las otras tareas por él planificadas. De esta forma es posible minimizar el efecto de inversión de prioridad sobre las demás tareas del sistema que utilizan el recurso global.

Salvo que el evento se encuentre enmascarado, el sistema operativo suspenderá a la tarea causante de la generación del evento hasta que su planificador decida de nuevo su activación. Este comportamiento es fundamental para los sistemas multiprocesadores, ya que de no hacerse así, una tarea planificada que provocara la generación de un evento, continuaría ejecutando en paralelo con su planificador antes de que éste tuviera oportunidad de decidir cuál debe ser su nuevo estado.

5.2.2. Gestión de mutexes con protocolo definido por la aplicación

Como comentamos con anterioridad, el protocolo de sincronización implementado por este tipo de mutexes está definido por su planificador de aplicación. Por tanto, este es el mecanismo que debe utilizar una tarea planificadora para definir los protocolos, compatibles con la política de planificación que implementa, bajo los que quiere que se sincronicen sus tareas planificadas. El planificador debe ser capaz de determinar qué tarea, de entre las que compiten por un mutex de aplicación, debe tomarlo y el momento en que debe hacerlo. Para este fin, es necesario que el sistema operativo notifique al planificador las operaciones que sus tareas realizan (o pretenden realizar) sobre los mutexes por él controlados y, por otra parte, la tarea planificadora debe ser capaz de ejecutar acciones de planificación que determinen el funcionamiento de dichos mutexes.

Sin embargo, no toda la gestión de los mutexes de aplicación es realizada por su tarea planificadora: ésta determina la tarea que toma el mutex, mientras que el sistema operativo se encarga de garantizar el acceso mutuamente exclusivo al recurso. Para el sistema operativo, un mutex de protocolo definido por la aplicación es un objeto que tiene una tarea propietaria y un conjunto de tareas en él bloqueadas. El sistema operativo no decide la tarea y el momento en que se entrega el mutex, dejando esta decisión en manos del planificador de aplicación, pero sí que se asegura de que el mutex no sea concedido a dos tareas simultáneamente y de que la tarea a la que es entregado sea una de las que se encontraban previamente bloqueadas en él. Cualquier intento por parte del planificador de violar una de estas reglas significará un error que será detectado por el sistema operativo.

Una tarea planificadora implementa el protocolo de sincronización de sus mutexes mediante el uso de un conjunto de acciones de planificación que se suman a las descritas en el apartado 4.2.2, “Relación entre el planificador y sus tareas planificadas”. Estas nuevas acciones son:

- *Aceptación de un mutex de aplicación*: el planificador acepta la creación, realizada desde una de sus tareas planificadas, de un mutex con protocolo definido por la aplicación. Para tomar la decisión de aceptación, el planificador puede basarse en los atributos de planificación del mutex, así como en otros factores conocidos por él, como puede ser el número de mutexes que tiene asociados en ese momento, etc.
- *Rechazo de un mutex de aplicación*: el planificador rechaza la creación, realizada desde una de sus tareas planificadas, de un mutex con protocolo definido por la aplicación. Esto provoca que la operación de creación retorne un código de error o eleve una excepción dependiendo de si se está usando la interfaz POSIX-C o POSIX-Ada. Ambos casos se tratarán en detalle en los apartados 5.3, “Descripción de la interfaz C” y 5.4, “Descripción de la interfaz Ada”.
- *Entrega de un mutex a una tarea de aplicación*: sirve para entregar el mutex a una de las tareas en él bloqueadas. En respuesta a esta acción, el sistema operativo concede el mutex a la tarea y la pone de nuevo en estado ejecutable. Se considera un error, que será notificado al planificador, el que la tarea no se encuentre bloqueada en el mutex o que el mutex sea propiedad de otra tarea.
- *Activación de una tarea de aplicación*: cuando esta acción se ejecuta sobre una tarea que ha invocado una operación de toma condicional (“`try_lock`”) sobre un mutex, sirve para indicar al sistema operativo que el intento de toma del mutex ha fracasado. En consecuencia, la tarea volverá a pasar al estado activo y la operación de toma condicional retornará el código de error correspondiente.

Los eventos de planificación relacionados con los mutexes con protocolo definido por la aplicación, que son generados por el sistema operativo para las tareas planificadoras son los mostrados en la tabla 5.2. Es importante destacar que el evento “Lock Mutex” significa que una tarea ha invocado la operación de toma de un mutex, no que el mutex le haya sido concedido por el sistema operativo. La concesión del mutex se llevará a cabo cuando el planificador de aplicación ejecute la acción de entrega de mutex descrita anteriormente. Lo mismo ocurre con el evento “Try Lock Mutex”.

Tabla 5.2: Eventos debidos al uso de mutexes de protocolo definido por la aplicación

Tipo de Evento	Descripción	Información Adicional
Init Mutex	Una tarea requiere la creación de un mutex de aplicación	Puntero al mutex
Destroy Mutex	Una tarea ha eliminado un mutex de aplicación	Puntero al mutex
Lock Mutex	Una tarea ha invocado la operación “lock” sobre un mutex de aplicación libre.	Puntero al mutex
Try Lock Mutex	Una tarea ha invocado la operación “try lock” sobre un mutex de aplicación libre.	Puntero al mutex
Unlock Mutex	Una tarea ha liberado un mutex de aplicación.	Puntero al mutex

Tabla 5.2: Eventos debidos al uso de mutexes de protocolo definido por la aplicación (cont.)

Tipo de Evento	Descripción	Información Adicional
Block at Mutex	Una tarea se ha bloqueado en un mutex de aplicación tomado por otra tarea.	Puntero al mutex
Change Mutex Scheduling Parameters	Una tarea ha cambiado los parámetros de planificación de un mutex de aplicación.	Puntero al mutex

Al igual que ocurría con los eventos relacionados con los mutexes de sistema y para la mayoría de los eventos presentados en el apartado 4.2.2, la generación por parte de una tarea planificada de cualquiera de los eventos de la tabla 5.2, provocará la suspensión de ésta a la espera de que sea activada de nuevo por su planificador.

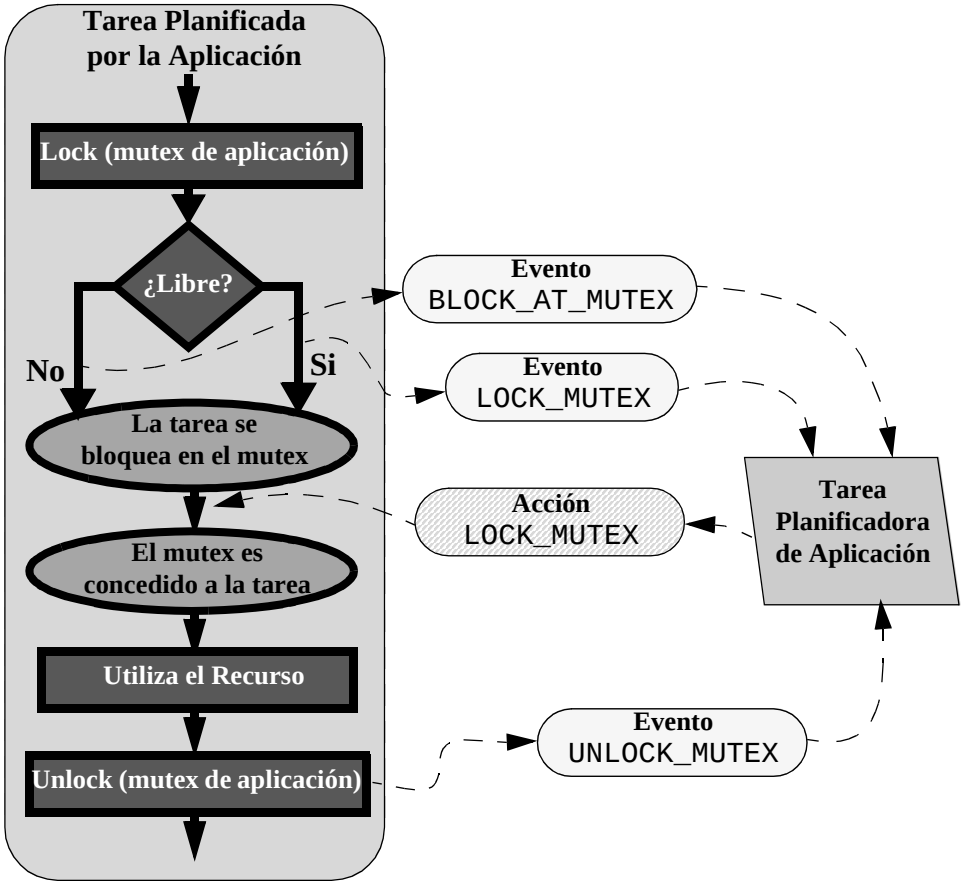

Figura 5.2: Toma de un mutex de aplicación

Con los eventos y acciones descritas, el proceso de toma de un mutex de aplicación es el mostrado en la figura 5.2. Cuando una tarea ejecuta la operación "`lock`" sobre el mutex se genera un evento para su planificador que será distinto en función del estado del mutex: `LOCK_MUTEX` cuando el mutex se encuentra libre o `BLOCK_AT_MUTEX` cuando en ese momento el mutex sea propiedad de otra tarea. En ambos casos el sistema operativo procederá al encolado de la tarea en el mutex, permaneciendo en esa situación hasta que el planificador ejecute una

acción LOCK_MUTEX sobre ella. Puesto que el sistema operativo se encarga de asegurar el acceso mutuamente exclusivo a los recursos, en el caso de que al ejecutar la citada acción el mutex no se encuentre libre, se notifica un error al planificador y la tarea permanecerá encolada en el mutex. Una vez concedido el mutex y ejecutada la sección crítica, la tarea liberará el mutex lo que provocará la generación del evento UNLOCK_MUTEX. Tras la llegada de este evento, el planificador podrá decidir entregar el mutex a otra tarea encolada ejecutando la acción LOCK_MUTEX sobre la nueva tarea elegida.

Por su parte, la operación de toma condicionada de un mutex por una tarea planificada por la aplicación se muestra en la figura 5.3. En el caso en que la tarea ejecute la operación "try_lock" sobre un mutex tomado, el comportamiento es el mismo que para los mutexes de protocolo de sistema, esto es: la tarea continúa en ejecución y la operación de toma condicional del recurso retorna un código de error. Por el contrario, si el mutex se encuentra libre el sistema operativo procede al encolado de la tarea, generándose el evento TRY_LOCK_MUTEX. Tras la generación de este evento, el planificador podrá decidir la concesión del mutex a la tarea encolada mediante la acción LOCK_MUTEX, con lo que el comportamiento posterior sería igual que en el caso de la operación de toma de un mutex anteriormente descrita. Pero a diferencia de las tareas encoladas en un mutex como consecuencia de una operación "lock", las encoladas por una operación "try_lock" pueden ser activadas por su planificador sin que les sea concedido el mutex. Para este fin el planificador puede utilizar la acción de activación de una tarea, con lo que ésta será puesta en estado ejecutable sin estar en posesión del recurso, y la operación de toma condicional retornará el código de error correspondiente al fracaso en la toma del mutex.

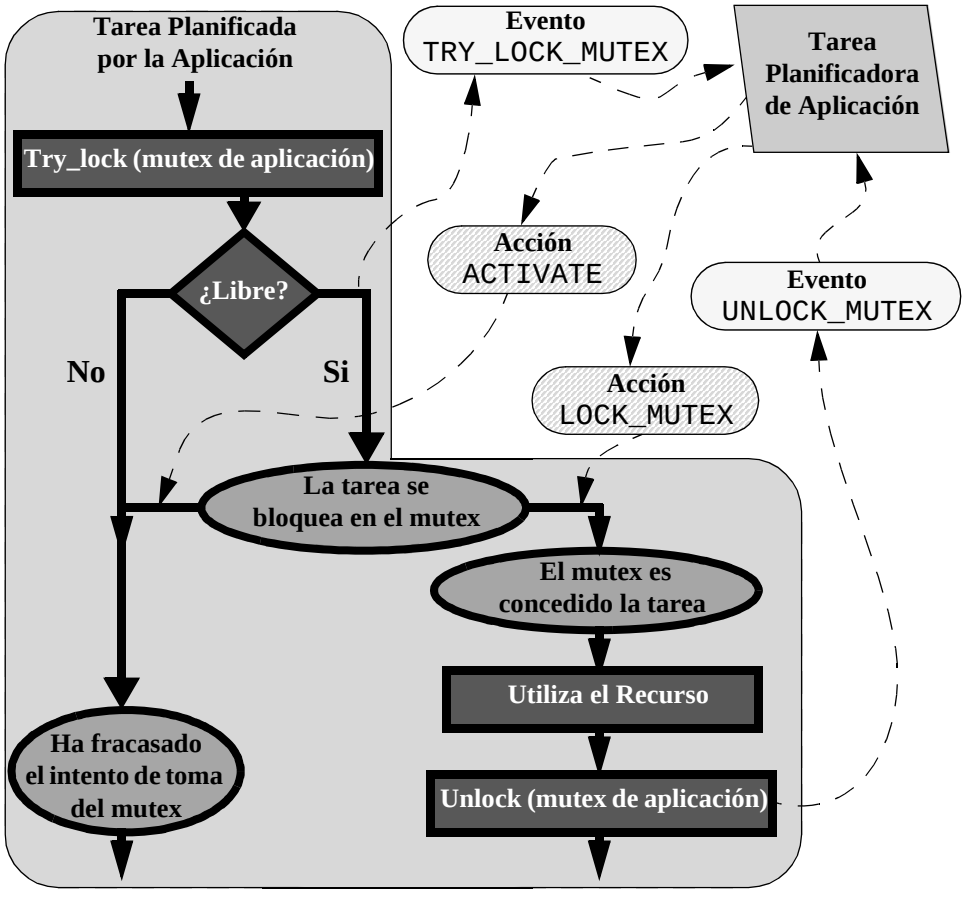

Figura 5.3: Toma condicionada de un mutex de aplicación

El comportamiento descrito para las operaciones "`lock`" y "`try_lock`", y en particular el hecho de que las tareas se bloqueen en el mutex aún en el caso de que éste se encuentre libre, es necesario para permitir la implementación de protocolos de sincronización en los que la toma de un recurso no depende únicamente de su estado (libre o tomado), sino que lo hace además de otros parámetros externos. Para que sea posible la implementación de este tipo de protocolos, es necesario que el planificador disponga de la opción de conceder o no un mutex libre a una tarea que lo solicite. Un ejemplo de este tipo de protocolos es el "Protocolo de Techo de Prioridad Global", también conocido como "Priority Ceiling Protocol" o simplemente PCP [SHA90] que fue descrito en la introducción de esta memoria. En este protocolo, la toma de un mutex libre únicamente se produce cuando la prioridad activa de la tarea sea estrictamente mayor que el techo de prioridad global; en caso contrario, la tarea deberá permanecer bloqueada en el mutex hasta que se verifique esa condición. En el apartado 5.5.1 mostraremos el pseudocódigo correspondiente a la implementación de este protocolo utilizando nuestra interfaz.

5.3. Descripción de la interfaz C

La interfaz presentada en el este apartado se ha diseñado de forma que constituya una extensión de la descrita en el apartado 4.3 para la definición de políticas de planificación. Como tal extensión, todos los criterios seguidos en el diseño de la interfaz base son válidos también para ésta. El principal de todos ellos es la integración en el estándar POSIX tratando de introducir el menor número posible de nuevas operaciones, prefiriéndose la ampliación de servicios ya existentes a la definición de interfaces totalmente nuevas.

Con esta interfaz se amplían las capacidades de los threads planificadores de aplicación en dos aspectos:

- Permitiendo que tengan noticia de la utilización de mutexes del sistema realizada por parte de sus threads planificados.
- Permitiéndoles definir los protocolos de sincronización de los mutexes con ellos vinculados.

Los nuevos servicios añadidos por la interfaz se incluyen en los ficheros de cabeceras estándar `<pthread.h>` y `<sched.h>`. La nueva funcionalidad introducida permite a las aplicaciones:

- *Crear mutexes con protocolo definido por la aplicación*: la interfaz define el nuevo protocolo de sincronización `PTHREAD_APPSCHED_PROTOCOL`. También se añaden dos nuevos atributos de los mutexes que permiten especificar su planificador y sus parámetros de planificación específicos del protocolo definido por la aplicación.
- *Añadir a una lista de acciones de planificación la acción de entrega de mutex*. Además se amplía el significado de la acción de activación, usándose también para indicar que un thread ha fracasado en una operación de toma condicional de un mutex.
- *Obtener la información asociada a los eventos de planificación*: la interfaz define nuevos tipos de eventos relacionados con el uso de mutexes de sistema y de protocolo definido por la aplicación. También añade nuevos campos a las estructuras que representan los eventos y permiten acceder a su información asociada.
- *Asignar y obtener los datos específicos de un mutex*: se añade una nueva funcionalidad similar a la denominada "Thread-Specific Data" pero aplicable a los mutexes.

Cada uno de los aspectos anteriormente citados se describen en detalle en los siguientes apartados. Cuando una funcionalidad extiende alguna de las definidas en la interfaz para la

definición de algoritmos de planificación descrita en el apartado 4.3, en este capítulo únicamente procederemos a describir las extensiones relacionadas con la gestión de los mutexes, debiendo referirse al citado apartado para una descripción de la parte común de ambas interfaces.

En el anexo A se presenta la descripción completa de la interfaz tal y como se propone para su integración en el estándar POSIX. En ella se encuentran englobadas tanto la parte relativa a los protocolos de sincronización como la que hace relación a las políticas de planificación definidas por la aplicación.

5.3.1. Creación de mutexes con protocolo definido por la aplicación

En el estándar POSIX, la forma de especificar las características de un mutex en el momento de su creación es mediante un objeto de atributos de tipo `pthread_mutexattr_t`. Entre otras características, en este objeto se especifica el protocolo de sincronización del mutex, pudiéndose elegir entre los protocolos estándar: `PTHREAD_PRIO_NONE`, `PTHREAD_PRIO_INHERIT` y `PTHREAD_PRIO_PROTECT`. La nueva interfaz añade el símbolo `PTHREAD_APPSCHED_PROTOCOL` para permitir la creación de mutexes con protocolo definido por la aplicación.

Para los mutexes de protocolo `PTHREAD_APPSCHED_PROTOCOL` es necesario además indicar su thread planificador de aplicación, que será el encargado de implementar su protocolo de sincronización. Para ello se define el nuevo atributo `appscheduler` cuyo valor es asignado y obtenido de un objeto de atributos mediante las funciones:

```
#include <pthread.h>
int pthread_mutexattr_setappscheduler (
                        pthread_mutexattr_t *attr,
                        pthread_t scheduler);
int pthread_mutexattr_getappscheduler (
                        const pthread_mutexattr_t *attr,
                        pthread_t *scheduler);
```

Como se ha comentado en la descripción del modelo, los mutexes de aplicación pueden crearse con unos parámetros de planificación específicos de su protocolo y que serán interpretados por su thread planificador. Con este propósito, la interfaz define el atributo `appschedparam` cuyo valor en un objeto de atributos se gestiona mediante las funciones mostradas a continuación:

```
#include <pthread.h>
int pthread_mutexattr_setappschedparam (
                       pthread_mutexattr_t *attr,
                       const void *param,
                       size_t param_size);
int pthread_mutexattr_getappschedparam (
                       const pthread_mutexattr_t *attr,
                       const void *param,
                       size_t *param_size);
```

De la misma forma que ocurre con los threads planificados por la aplicación y sus parámetros de planificación específicos, en el caso de los parámetros específicos del protocolo de aplicación es necesario que el sistema operativo sea informado de forma explícita sobre su tamaño. Este dato es gestionado mediante el parámetro `param_size` de las funciones anteriormente mostradas.

La interfaz también proporciona funciones que permiten cambiar los parámetros específicos del protocolo de forma dinámica una vez creado el mutex. Los prototipos de dichas funciones se muestran a continuación:

```
#include <pthread.h>
int pthread_mutex_setappschedparam (pthread_mutex_t *mutex,
                                    const void *param,
                                    size_t param_size);
int pthread_mutex_getappschedparam (const pthread_mutex_t *mutex,
                                    void *param,
                                    size_t *param_size);
```

El estándar POSIX no proporciona funciones que permitan cambiar dinámicamente el protocolo de sincronización con el que fue creado un mutex, puesto que se considera que ese dato es conocido y determinado en el momento de su creación. Por la misma razón, en la interfaz presentada no se han definido funciones para convertir un mutex de uno de los protocolos estándar en un mutex de protocolo de aplicación ni para cambiar el planificador de un mutex de este último tipo. Únicamente se proporciona una función que permite obtener el valor del atributo `appscheduler` de un mutex en un momento posterior a su creación:

```
#include <pthread.h>
int pthread_mutex_getappscheduler (pthread_mutex_t *mutex,
                                   pthread_t *scheduler);
```

5.3.2. Acciones de planificación

Las acciones de planificación constituyen el mecanismo mediante el cual un thread planificador comunica al sistema operativo las operaciones que desea realizar sobre los threads y mutexes por él planificados. En el apartado 4.3.2, “Gestión y ejecución de las acciones de planificación”, explicábamos como se añaden las acciones una lista de acciones (un objeto de tipo `posix_appsched_actions_t`) para posteriormente ser ejecutadas mediante la función `posix_appsched_execute_actions()`.

Con la extensión de la interfaz para permitir la definición de protocolos de sincronización, se añaden nuevas acciones a las descritas en el citado apartado 4.3.2. Estas nuevas acciones son:

- Aceptación de un mutex creado para ser planificado por el planificador.
- Rechazo de un mutex creado para ser planificado por el planificador.
- Entrega de un mutex de aplicación a uno de los threads en él bloqueados.

Los prototipos de las funciones proporcionadas por la interfaz para añadir las acciones anteriores a una lista de acciones son:

```
#include <sched.h>
int posix_appsched_actions_addacceptmutex (
                posix_appsched_actions_t *sched_actions,
                const pthread_mutex_t *mutex);
int posix_appsched_actions_addrejectmutex (
                posix_appsched_actions_t *sched_actions,
                const pthread_mutex_t *mutex);
```

```
int posix_appsched_actions_addlockmutex (
                posix_appsched_actions_t *sched_actions,
                pthread_t thread,
                const pthread_mutex_t *mutex);
```

Además de añadir estas nuevas acciones, con la introducción de los mutexes de protocolo de aplicación se da un nuevo significado a la acción de activación de un thread: la ejecución de esta acción sobre un thread que ha intentado tomar un mutex de aplicación invocando la función `pthread_mutex_trylock()`, significa que ha fracasado en su intento. Como consecuencia de la ejecución de esta acción, el thread será eliminado de la cola del mutex pasando a estado ejecutable. Una vez le sea concedida la CPU finalizará la ejecución de la función `pthread_mutex_trylock()` obteniendo el valor de retorno correspondiente al fallo en la toma del recurso.

5.3.3. Eventos de planificación

La interfaz define nuevos eventos relacionados con la utilización que los threads planificados hacen de los mutexes de protocolo definido por el sistema y de protocolo definido por la aplicación. Los símbolos definidos en el fichero de cabeceras `<sched.h>` para los nuevos eventos relacionados con el uso de mutexes del sistema son los mostrados en la tabla 5.3. En la tabla también se indican las causas que provocan la generación de tales eventos.

Tabla 5.3: Eventos relacionados con el uso de mutexes del sistema.

Tipo de Evento	Causa del Evento
`POSIX_APPSCHED_PRIORITY_INHERIT`	Un thread de aplicación ha heredado una prioridad debido al uso de un mutex de sistema.
`POSIX_APPSCHED_PRIORITY_UNINHERIT`	Un thread de aplicación ha perdido una prioridad heredada como consecuencia del uso de un mutex de sistema.

Estos eventos tienen un carácter meramente informativo puesto que el planificador no puede influir en la decisión de entrega de un mutex a un determinado thread, la cual es tomada por el sistema operativo en base a las políticas de planificación y protocolos de sincronización estándares. A pesar de ese carácter informativo estos eventos pueden resultar de gran utilidad, ya que permiten al planificador dar preferencia al thread en posesión del recurso sobre los otros threads por él planificados. De esta forma es posible minimizar el efecto de inversión de prioridad sobre los demás threads del sistema que utilizan el recurso global.

La interfaz también añade un conjunto de nuevos eventos relacionados con el uso de los mutexes de protocolo definido por la aplicación. La constantes simbólicas definidas en el fichero de cabeceras `<sched.h>` para estos nuevos eventos junto con la causa que provoca la generación de cada uno de ellos son las que aparecen en la tabla 5.4.

Como se vio en el apartado 4.3.4, como consecuencia de la generación de un evento el sistema operativo encola una estructura `posix_appsched_event` en la cola FIFO de eventos asociada con el thread planificador. Esta estructura incluye campos que permiten identificar el tipo de evento y el thread causante de su generación. Además, el campo `event_info` (una unión `posix_appsched_eventinfo`) contiene información dependiente del tipo de evento. Con la incorporación de los nuevos tipos de eventos relacionados con los mutexes de sistema y de

Tabla 5.4: Eventos relacionados con el uso de mutexes de aplicación

Tipo de Evento	Causa del Evento
`POSIX_APPSCHED_INIT_MUTEX`	Un thread de aplicación solicita la creación de un mutex de protocolo definido por la aplicación.
`POSIX_APPSCHED_DESTROY_MUTEX`	Un thread de aplicación ha destruido un mutex de protocolo definido por la aplicación.
`POSIX_APPSCHED_LOCK_MUTEX`	Un thread de aplicación ha invocado la operación `pthread_mutex_lock()` sobre un mutex de aplicación libre.
`POSIX_APPSCHED_TRY_LOCK_MUTEX`	Un thread de aplicación ha invocado la operación `pthread_mutex_trylock()` sobre un mutex de aplicación libre.
`POSIX_APPSCHED_UNLOCK MUTEX`	Un thread de aplicación ha invocado la operación `pthread_mutex_unlock()` sobre un mutex de aplicación.
`POSIX_APPSCHED_BLOCK_AT_MUTEX`	Un thread de aplicación ha invocado la operación `pthread_mutex_lock()` sobre un mutex de aplicación tomado.
`POSIX_APPSCHED_-CHANGE_MUTEX_SCHED_PARAM`	Un thread de aplicación ha cambiado los parámetros de planificación de un mutex de aplicación.

aplicación, se añaden a esta unión los campos mostrados en la tabla 5.5. El campo `sched_priority` será utilizado con los eventos relacionados con los mutexes del sistema (mostrados en la tabla 5.3), mientras que el campo `mutex` lo será con todos los eventos relacionados con los mutexes de protocolo definido por la aplicación, que son los que aparecen en la tabla 5.4.

Tabla 5.5: Campos de la unión `posix_appsched_eventinfo`.

Tipo	Nombre	Descripción
`int`	`sched_priority`	Prioridad de sistema heredada o perdida por un thread planificado debido al uso de un mutex del sistema.
`pthread_mutex_t *`	`mutex`	Mutex de protocolo de aplicación sobre el que un thread planificado ha realizado una operación.
Además esta unión contiene los campos `siginfo`, `info` y `user_event_code` cuyo significado fue comentado en el apartado 4.3.4.		

5.3.4. Datos específicos de los mutexes

Al igual que ocurría con los threads planificados, lo normal será que los planificadores asocien una estructura de control con cada uno de sus mutexes. Esta estructura contendrá información

relativa a su estado, como por ejemplo el identificador del thread propietario y la lista de threads encolados. Como expusimos en el apartado anterior, el planificador puede conocer el mutex sobre el que se ha realizado una operación mediante el puntero a un objeto de tipo `pthread_mutex_t` que le es pasado en el campo `mutex` de la unión `posix_appsched_eventinfo`. Resultaría muy útil disponer para los mutexes de una funcionalidad similar a los datos específicos de los threads para que, de esta forma, fuera posible asignar y obtener la información asociada a un mutex mediante su identificador (puntero a un objeto de tipo `pthread_mutex_t`).

Con este objetivo, en la interfaz introducimos una nueva funcionalidad no existente en el estándar POSIX, la cual permite asociar datos específicos a los mutexes. A diferencia de lo que ocurre en la funcionalidad equivalente para los threads, en la definida para los mutexes sólo se permite asociar un dato con cada mutex, cuyo valor es gestionado utilizando las funciones:

```
#include <pthread.h>
int posix_appsched_mutex_setspecific(
                    pthread_mutex_t *mutex,
                    const void *data);
int posix_appsched_mutex_getspecific(
                  const pthread_mutex_t *mutex,
                  void **data);
```

5.4. Descripción de la interfaz Ada

En este apartado se presenta una interfaz Ada con la misma funcionalidad que la interfaz C descrita en el apartado anterior. Esta interfaz Ada extiende la funcionalidad descrita en el apartado 4.4 para la definición por la aplicación de políticas de planificación, ampliando las capacidades de las tareas planificadoras en dos aspectos:

- Permitiendo que tengan noticia de la utilización de los mutexes del sistema realizada por parte de sus tareas planificadas.
- Permitiéndolas definir los protocolos de sincronización de los mutexes con ellas vinculados.

La interfaz se integra en el estándar POSIX.5b (“binding” en lenguaje Ada del POSIX.1), añadiéndose a los paquetes `POSIX_Application_Scheduling` y `POSIX_Mutexes` nuevos servicios que permiten a las aplicaciones:

- *Crear mutexes con protocolo definido por la aplicación*: la interfaz define el nuevo protocolo de sincronización `APPSCHED_PROTOCOL`. También se añaden dos nuevos atributos de los mutexes que permiten especificar su planificador y sus parámetros de planificación específicos del protocolo definido por la aplicación.
- *Añadir a una lista de acciones la acción de entrega de un mutex*. Además se amplía el significado de la acción de activación, usándose también para indicar que una tarea ha fracasado en una operación de intento de toma de un mutex.
- *Obtener la información asociada a los eventos de planificación*: la interfaz define nuevos tipos de eventos relacionados con el uso de mutexes de sistema y de protocolo definido por la aplicación. También añade funciones que permiten acceder a la información asociada con los nuevos tipos de eventos introducidos.

- *Asignar y obtener los datos específicos de un mutex*: se añade una nueva funcionalidad similar a la proporcionada para las tareas por el paquete estándar `Ada.Task_Attributes` pero aplicable a los mutexes.

La interfaz, al estar definida como una extensión del estándar POSIX-Ada, no permite crear objetos protegidos cuyo protocolo esté definido por la aplicación. Para que esto fuera posible, la interfaz debería incluirse como una extensión del propio lenguaje de forma similar a como el manual de referencia define los anexos especializados (anexo de tiempo real, anexo de sistemas distribuidos, etc.). Con esta segunda opción se podrían ofrecer un conjunto de directivas al compilador ("pragmas") que permitieran indicar el planificador y demás propiedades de un objeto protegido con protocolo definido por la aplicación. La modificación de la librería de tiempo de ejecución y del compilador Ada quedan fuera de los objetivos de esta tesis, constituyendo una línea de trabajo futuro que plantearemos en el capítulo 6, "Conclusiones y trabajo futuro".

En los siguientes apartados procederemos a describir en detalle cada uno de los servicios definidos por la interfaz. Cuando una funcionalidad extiende alguna de las definidas en la interfaz para la definición de algoritmos de planificación descrita en el apartado 4.4, en este capítulo únicamente se procederá a describir las extensiones relacionadas con la gestión de los mutexes, debiendo referirse al citado apartado para una descripción de la parte común de ambas interfaces.

5.4.1. Creación de mutexes con protocolo definido por la aplicación

En el "binding" Ada del estándar POSIX, la forma de especificar las características de un mutex en el momento de su creación es mediante un objeto de atributos de tipo `Attributes`. Entre otras características, en este objeto se especifica el protocolo de sincronización del mutex, pudiéndose elegir entre los protocolos estándar: `NO_PRIORITY_INHERITANCE`, `HIGHEST_CEILING_PRIORITY` y `HIGHEST_BLOCKED_TASK`. La nueva interfaz añade la constante `APPSCHED_PROTOCOL` para permitir la creación de mutexes con protocolo definido por la aplicación.

Para los mutexes de protocolo `APPSCHED_PROTOCOL` es necesario además indicar la tarea planificadora que implementa el protocolo de sincronización del mutex. Para ello se define el nuevo atributo "Application Scheduler" cuyo valor es asignado y obtenido de un objeto de atributos mediante las nuevas operaciones `POSIX_Mutexes.Set_App_Scheduler` y `POSIX_Mutexes.Get_App_Scheduler`. La interfaz de estas operaciones se muestra a continuación:

```
procedure Set_App_Scheduler
   (Attr      : in out Attributes;
    Scheduler : in     Ada.Task_Identification.Task_Id);

function Get_App_Scheduler (Attr : in Attributes)
                   return Ada.Task_Identification.Task_Id;
```

Como se ha comentado en la descripción del modelo, los mutexes de aplicación pueden crearse con unos parámetros de planificación específicos de su protocolo y que serán interpretados por su tarea planificadora. Con este propósito, la interfaz define el paquete genérico `POSIX_Mutexes.Application_Defined_Parameters`:

```
generic
   type Parameters is private;
package Application_Defined_Parameters is
   procedure Set_Parameters (Attr  : in out Attributes;
                             Param : in     Parameters);

   procedure Get_Parameters (Attr  : in  Attributes;
                             Param : out Parameters);

   procedure Set_Parameters (M     : in Mutex_Descriptor;
                             Param : in Parameters);

   procedure Get_Parameters (M     : in  Mutex_Descriptor;
                             Param : out Parameters);
end Application_Defined_Parameters;
```

En este paquete el parámetro genérico `Parameters` es el objeto de parámetros de planificación específicos del protocolo. El paquete proporciona procedimientos que permiten asignar y obtener su valor en un objeto de atributos y otros que permiten cambiar estos parámetros de forma dinámica una vez creado el mutex.

El estándar POSIX no proporciona operaciones que permitan cambiar dinámicamente el protocolo de sincronización con el que fue creado un mutex, ya que se considera que ese dato es conocido y determinado en el momento de su creación. Por la misma razón en la interfaz presentada no se han definido operaciones para convertir un mutex con uno de los protocolos estándares en un mutex de protocolo de aplicación ni para cambiar el planificador de un mutex de este último tipo. Únicamente se ha definido una función en el paquete `POSIX_Mutexes` que permite obtener el valor del atributo “Application Scheduler” de un mutex en un momento posterior a su creación:

```
function Get_App_Scheduler (M : in Mutex_Descriptor)
                 return Ada.Task_Identification.Task_Id;
```

5.4.2. Acciones de planificación

Las acciones de planificación constituyen el mecanismo mediante el cual una tarea planificadora comunica al sistema operativo las operaciones que desea realizar sobre las tareas y mutexes por él planificados. En el apartado 4.4.2, “Gestión y ejecución de las acciones de planificación”, se explica como se añaden las acciones a una lista de acciones (un objeto de tipo `Scheduling_Actions`) para posteriormente ser ejecutadas mediante los procedimientos de las familias `Execute_Actions` y `Execute_Actions_With_Timeout`.

Con la extensión de la interfaz para permitir la definición de protocolos de sincronización, se añaden nuevas acciones a las descritas en el citado apartado 4.4.2. Estas nuevas acciones son:

- Aceptación de un mutex creado para ser planificado por el planificador.
- Rechazo de un mutex creado para ser planificado por el planificador.
- Entrega de un mutex de aplicación a una de las tareas en él bloqueadas.

Los procedimientos proporcionados por el paquete `POSIX_Application_Scheduling` para añadir las acciones anteriormente citadas a un objeto de tipo `Scheduling_Actions` son:

```
procedure Add_Accept_Mutex
      (Sched_Actions : in out Scheduling_Actions;
       M             : in     Posix_Mutexes.Mutex_Descriptor);
procedure Add_Reject_Mutex
      (Sched_Actions : in out Scheduling_Actions;
       M             : in     Posix_Mutexes.Mutex_Descriptor);
procedure Add_Lock_Mutex
      (Sched_Actions : in out Scheduling_Actions;
       T             : in     Ada.Task_Identification.Task_Id;
       M             : in     Posix_Mutexes.Mutex_Descriptor);
```

Además de añadir estas nuevas acciones, con la introducción de los mutexes de protocolo de aplicación se da un nuevo significado a la acción de activación de una tarea: la ejecución de esta acción sobre una tarea que ha intentado tomar un mutex de aplicación invocando la función `Try_Lock`, significa que ha fracasado en su intento. Como consecuencia de la ejecución de esta acción, la tarea será eliminada de la cola del mutex pasando a estado ejecutable. Una vez le sea concedida la CPU finalizará la ejecución de la función `Try_Lock`, retornando el valor `FALSE` como consecuencia del fallo en la toma del recurso.

5.4.3. Eventos de planificación

La interfaz define nuevos eventos relacionados con la utilización que las tareas planificadas hacen de los mutexes de protocolo definido por el sistema y de protocolo definido por la aplicación. Las constantes definidas en el paquete `POSIX_Application_Scheduling` para los nuevos tipos de eventos relacionados con el uso de mutexes del sistema se muestran en la tabla 5.6. En la tabla también se indican las causas que provocan la generación de tales eventos.

Tabla 5.6: Eventos relacionados con el uso de mutexes del sistema.

Tipo de Evento	Causa del Evento
`PRIORITY_INHERIT`	Una tarea de aplicación ha heredado una prioridad debido al uso de un mutex de sistema.
`PRIORITY_UNINHERIT`	Una tarea de aplicación ha perdido una prioridad heredada como consecuencia del uso de un mutex de sistema.

Estos eventos tienen un carácter meramente informativo puesto que el planificador no puede influir en la decisión de entrega del mutex a una determinada tarea, decisión que es tomada por el sistema operativo en base a las políticas de planificación y protocolos de sincronización estándares. A pesar de ese carácter informativo estos eventos pueden resultar de gran utilidad, ya que permiten al planificador dar preferencia a la tarea en posesión del recurso sobre las otras tareas por él planificadas, para de esta forma minimizar el efecto de inversión de prioridad sobre las demás tareas del sistema que comparten el recurso global.

Es importante notar que, en aquellas librerías de tiempo de ejecución que utilizan mutexes para implementar los objetos protegidos, los eventos mostrados en la tabla 5.6 proporcionan información sobre su utilización por parte de las tareas planificadas. Este es el caso de la librería de tiempo de ejecución del compilador GNAT [MIR02].

La interfaz también añade un conjunto de nuevos eventos relacionados con el uso de los mutexes de protocolo definido por la aplicación. La constantes definidas en el paquete

POSIX_Application_Scheduling para estos nuevos tipos de eventos junto con la causa que provoca la generación de cada uno de ellos son las que aparecen en la tabla 5.7.

Tabla 5.7: Eventos relacionados con el uso de mutexes de aplicación

Tipo de Evento	Causa del Evento
INIT_MUTEX	Una tarea de aplicación solicita la creación de un mutex de protocolo definido por la aplicación.
DESTROY_MUTEX	Una tarea de aplicación ha destruido un mutex de protocolo definido por la aplicación.
LOCK_MUTEX	Una tarea de aplicación ha invocado la operación Lock sobre un mutex de aplicación libre.
TRY_LOCK_MUTEX	Una tarea de aplicación ha invocado la operación Try_Lock sobre un mutex de aplicación libre.
UNLOCK MUTEX	Una tarea de aplicación ha invocado la operación Unlock sobre un mutex de aplicación.
BLOCK_AT_MUTEX	Una tarea de aplicación ha invocado la operación Lock sobre un mutex de aplicación tomado.
CHANGE_MUTEX_SCHED_PARAM	Una tarea de aplicación ha cambiado los parámetros de planificación de un mutex de aplicación.

Como consecuencia de la generación de un evento el sistema operativo encola un objeto del tipo privado Scheduling_Event en la cola FIFO de eventos de planificación asociada con la tarea planificada.

En el capítulo 4 se presentaron las funciones Get_Event_Code, Get_User_Event_Code, Get_Task, Get_Signal_Info y Explicit_Scheduler_Invocation.Get_Message, que permitían obtener la información asociada con los objetos de tipo Scheduling_Event. Con la incorporación de los nuevos tipos de eventos relacionados con los mutexes de sistema y de aplicación, en el paquete POSIX_Application_Scheduling se definen las funciones Get_Sched_Priority y Get_Mutex cuyas interfaces son las mostradas a continuación:

```
function Get_Sched_Priority (Event : in Scheduling_Event)
                             return System.Any_Priority;
function Get_Mutex (Event : in Scheduling_Event)
                    return POSIX_Mutexes.Mutex_Descriptor;
```

La función Get_Sched_Priority permite obtener la prioridad de sistema heredada o perdida por una tarea planificada debido al uso de un mutex del sistema. La invocación de esta función para un Scheduling_Event cuyo tipo no sea PRIORITY_INHERIT o PRIORITY_UNINHERIT provocará que se eleve la excepción POSIX_Error.

Por su parte, mediante la función Get_Mutex es posible obtener el mutex de protocolo de aplicación sobre el que una tarea planificada ha realizado una operación. Esta función sólo debe ser utilizada con objetos de tipo Scheduling_Event que representen un evento de los que aparecen en la tabla 5.7; en caso contrario se elevará la excepción POSIX_Error.

5.4.4. Datos específicos de los mutexes

Al igual que ocurría con las tareas planificadas, lo normal será que los planificadores asocien a cada uno de sus mutexes una estructura de control con información relativa a su estado. Esta información posiblemente contendrá entre otros datos el identificador de la tarea propietaria y la lista de tareas encoladas. Como se ha expuesto en el apartado anterior, el planificador conoce el mutex sobre el que se ha realizado una operación mediante la función `Get_Mutex` que retorna un descriptor de mutex (`POSIX_Mutexes.Mutex_Descriptor`). Resultaría muy útil disponer para los mutexes de una funcionalidad similar a la proporcionada para las tareas por el paquete estándar `Ada.Task_Attributes`, que nos permitiera acceder a la información asociada con un mutex utilizando su descriptor.

Con este objetivo, la interfaz introduce el paquete genérico `POSIX_Mutexes.Mutex_Attributes` que permite asociar datos específicos a los mutexes. La interfaz de dicho paquete se muestra a continuación:

```
generic
   type Attribute is private;
package Mutex_Attributes is
   type Attribute_Handle is access all Attribute;

   function Get (M : Mutex_Descriptor) return Attribute_Handle;

   procedure Set (Val : in Attribute_Handle;
                  M   : in Mutex_Descriptor);
end Mutex_Attributes;
```

5.5. Ejemplos de uso de la interfaz

5.5.1. Protocolo de techo de prioridad

En este ejemplo mostraremos la implementación del protocolo de sincronización de techo de prioridad, también conocido como PCP (“Priority Ceilling Protocol”), el cual fue descrito en el apartado 1.2 de la introducción de esta memoria. Para su realización utilizaremos la versión C de nuestra interfaz.

El protocolo de techo de prioridad es compatible con las políticas de planificación descritas en el estándar POSIX. En consecuencia, nuestro planificador únicamente implementa el protocolo de sincronización, concediendo un mutex cuando sea posible y modificando la prioridad de los threads según las reglas que dicta este protocolo. La planificación de los threads se deja en manos del sistema operativo, el cual elegirá el thread más prioritario de entre todos los que se encuentren listos para ejecutar en un determinado momento.

Cuando el planificador acepta un thread, le asocia mediante la función `pthread_setspecific_for()` una estructura `datos_thread` como la mostrada a continuación. Esta estructura contiene un campo que permite identificar el mutex en el que se encuentra bloqueado el thread.

```
struct datos_thread {
  struct datos_mutex *bloqueado_en_mutex;
};
```

También se asocia una estructura, en este caso de tipo `datos_mutex`, con cada mutex aceptado por el planificador. Esta asociación se realiza mediante la función `posix_appsched_mutex_setspecific()` descrita en el apartado 5.3.4. El planificador gestiona una lista que contiene las estructuras asociadas con todos los mutexes de protocolo de aplicación (`lista_mutexes_pcp`).

```
struct datos_mutex {
  lista_t threads_bloqueados;
  struct datos_thread *propietario;
  int techo;
};
```

La implementación del protocolo de techo de prioridad propiamente dicho se encuentra oculta en varias funciones llamadas por el thread planificador:

- `bloqueo_por_techo_de_prioridad_global()`: detecta si se produce un bloqueo de techo de prioridad global.
- `eleva_prio_de_todos_los_threads_bloqueantes()`: llamada cuando un thread se bloquea en un mutex. Sirve para elevar las prioridades de todos los threads bloqueantes por bloqueo de techo global, esto es, todos aquellos que están en posesión de mutexes con techo mayor o igual que la prioridad del thread recién bloqueado.
- `actualiza_la_prioridad_del_thread()`: es llamada cuando un thread toma un mutex en el que estaba bloqueado o cuando se libera un mutex. Encuentra la nueva prioridad activa del thread de acuerdo con su prioridad base y todas las prioridades heredadas.
- `hay_un_nuevo_thread_que_puede_tomar_mutex()`: tras la liberación de un mutex el techo global cambia. Esta función detecta si con la nueva situación algún thread se encuentra en disposición de tomar un mutex.

El pseudocódigo correspondiente al thread planificador se muestra a continuación:

```
void * planificador_pcp (void * arg)
{
  ...
  // Lazo de planificación
  while (1) {
    // Ejecuta acciones de planificación y espera un nuevo evento
    posix_appsched_execute_actions (&acciones, NULL, NULL, NULL,
                                    &evento);

    switch (evento.event_code) {
    case POSIX_APPSCHED_NEW:
      // Asocia su estructura 'datos_thread'
      d_thread = ...;
      pthread_setspecific_for (key_pcp, evento.thread, d_thread);
      // Acepta y activa el thread
      posix_appsched_actions_addaccept (&acciones, evento.thread);
      posix_appsched_actions_addactivate (&acciones, evento.thread);
      break;

    case POSIX_APPSCHED_TERMINATE:
      elimina su estructura de datos asociada;
```

```
    break;

  case POSIX_APPSCHED_INIT_MUTEX:
    // Obtiene sus parámetros de planificación específicos (techo
    // de prioridad)
    pthread_mutex_getappschedparam (...);
    // Asocia su estructura 'datos_mutex' y la añade a la lista
    d_mutex = ...;
    posix_appsched_mutex_setspecific (evento.event_info.mutex,
                                      d_mutex);
    encola (d_mutex, lista_mutexes_pcp);
    // Acepta el mutex
    posix_appsched_actions_addacceptmutex
                             (&acciones, evento.event_info.mutex);
    break;

  case POSIX_APPSCHED_DESTROY_MUTEX:
    elimina su estructura de datos asociada;
    desencola (d_mutex, lista_mutexes_pcp);
    break;

  case POSIX_APPSCHED_LOCK_MUTEX:
    // Obtiene los datos asociados al thread y al mutex
    pthread_getspecific_from (key_pcp, evento.thread, &d_thread);
    posix_appsched_mutex_getspecific (evento.event_info.mutex,
                                      &d_mutex);
    // Una tarea de aplicación trata de tomar un mutex libre
    if (bloqueo_por_techo_de_prioridad_global (d_thread)) {
      // Bloquea thread en el mutex
      d_thread.bloqueado_en_mutex = d_mutex;
      encola (d_thread, d_mutex.threads_bloqueados);
      eleva_prio_de_todos_los_threads_bloqueantes (d_thread);
    } else {
      // El thread puede tomar el mutex
      d_mutex.propietario = d_thread;
      posix_appsched_actions_addlockmutex
           (&acciones, evento.thread, evento.event_info.mutex);
    }
    break;

  case POSIX_APPSCHED_TRY_LOCK_MUTEX:
    // Obtiene los datos asociados al thread y al mutex
    pthread_getspecific_from (key_pcp, evento.thread, &d_thread);
    posix_appsched_mutex_getspecific (evento.event_info.mutex,
                                      &d_mutex);
    if (bloqueo_por_techo_de_prioridad_global (d_thread)) {
      // El thread no puede tomar el mutex
      posix_appsched_actions_addactivate (&acciones,
                                          evento.thread);
    } else {
      // El thread puede tomar el mutex
      d_mutex.propietario = d_thread;
      posix_appsched_actions_addlockmutex
           (&acciones, evento.thread, evento.event_info.mutex);
```

```
      }
      break;

    case POSIX_APPSCHED_UNLOCK_MUTEX:
      // Obtiene los datos asociados al thread y al mutex
      pthread_getspecific_from (key_pcp, evento.thread, &d_thread);
      posix_appsched_mutex_getspecific (evento.event_info.mutex,
                                        &d_mutex);
      // El thread libera el mutex
      d_mutex.propietario = NULL;
      actualiza_la_prioridad_del_thread (d_thread);
      if (hay_un_nuevo_thread_que_puede_tomar_mutex()) {
        // El thread deja de estar bloqueado en el mutex
        desencola (nuevo_d_thread,
                   nuevo_d_mutex.threads_bloqueados);
        nuevo_d_thread.bloqueado_en_mutex = NULL;
        // El thread toma el mutex
        nuevo_d_mutex.propietario = nuevo_d_thread;
        actualiza_la_prioridad_del_thread (nuevo_d_thread);
        posix_appsched_actions_addlockmutex
                              (&acciones, nuevo_thread,
                               nuevo_thread.mutex_where_blocked);
      }
      break;

    case POSIX_APPSCHED_BLOCK_AT_MUTEX:
      // Obtiene los datos asociados al thread y al mutex
      pthread_getspecific_from (key_pcp, evento.thread, &d_thread);
      posix_appsched_mutex_getspecific (evento.event_info.mutex,
                                        &d_mutex);
      // Bloquea thread en el mutex
      d_thread.bloqueado_en_mutex = d_mutex;
      encola (d_thread, d_mutex.threads_bloqueados);
      eleva_prio_de_todos_los_threads_bloqueantes (d_thread);
      break;

    } // switch
  } // while (1)
  ...
  return NULL;
} // planificador_pcp
```

A continuación mostramos el código correspondiente a la creación del thread planificador, de un mutex de protocolo definido por la aplicación y de un thread planificado que hará uso de los mutexes de protocolo PCP.

```
int main ()
{
  ...
  // Creación del thread planificador de aplicación
  pthread_attr_init (&attr);
  pthread_setinheritsched (&attr, PTHREAD_EXPLICIT_SCHED);
  pthread_attr_setappschedulerstate (&attr, PTHREAD_APPSCHEDULER);
  pthread_attr_setschedpolicy (&attr, SCHED_FIFO);
```

```
  param.sched_priority = 8;
  pthread_attr_setschedparam (&attr, &param);
  pthread_create (&planificador_id, &attr, planificador_pcp, NULL);

  // Creación de un mutex con protocolo definido por la aplicación
  pthread_mutexattr_init (&mutex_attr);
  pthread_mutexattr_setprotocol (&mutex_attr,
                                 PTHREAD_APPSCHED_PROTOCOL);
  pthread_mutexattr_setappscheduler (&mutex_attr, planificador_id);
  techo_mutex = 6;
  pthread_mutexattr_setappschedparam (&mutex_attr,
                                      &techo_mutex,
                                      sizeof (techo_mutex));
  pthread_mutex_init (&mutex_pcp, &mutex_attr);

  // Creación de un thread de política definida por la aplicación
  pthread_attr_init (&attr);
  pthread_setinheritsched (&attr, PTHREAD_EXPLICIT_SCHED);
  pthread_attr_setschedpolicy (&attr, SCHED_APP);
  pthread_attr_setappscheduler (&attr, planificador_id);
  param.sched_priority = 8;
  pthread_attr_setschedparam (&attr, &param);
  pthread_create (&t1, &attr, pthread_planificado, NULL);
  ...
} // main
```

La forma en que los threads planificados por la aplicación utilizan los mutexes de protocolo PCP no se diferencia del uso normal de un mutex descrito por el estándar POSIX. Así, un extracto del código de un thread planificado podría ser el mostrado a continuación:

```
// Cuerpo de un thread planificado por la aplicación
void * pthread_planificado (void * arg)
{
  ...
  pthread_mutex_lock (&mutex_pcp);
  uso_del_recurso ();
  pthread_mutex_unlock (&mutex_pcp);
  ...
  return NULL;
}
```

5.5.2. Protocolo de techo de prioridad dinámico

En este ejemplo presentaremos el pseudocódigo correspondiente a un thread planificador que implementa el protocolo de sincronización de techo de prioridad dinámico. Este protocolo, descrito en el apartado 1.3, posibilita la sincronización entre threads de política EDF. A diferencia del ejemplo anterior en el que se dejaban las decisiones de planificación en manos del sistema operativo, en este caso el planificador implementa tanto la política de planificación (EDF) como el propio protocolo de sincronización.

El algoritmo implementado es el descrito por Chen y Lin en el artículo en el que presentan el protocolo de techo de prioridad dinámico [CHE90]. Para la implementación se utilizan tres colas ordenadas en base al plazo de ejecución de los threads que contienen:

- `cola_threads_activos`: en ella se encuentran todos los threads en disposición de ejecutar. Además, en esta cola se inserta una entrada "heredada" cada vez que se produce un bloqueo en un mutex. Esta entrada, que contiene el thread y el mutex causantes del bloqueo, es ordenada en base al menor de los plazos de los threads bloqueados en el mutex y eliminada una vez que el thread bloqueante ha liberado el recurso.
- `cola_threads_suspendidos`: en esta cola se encuentran todos los threads que han finalizado el trabajo correspondiente a su activación actual y se encuentran suspendidos a la espera de la llegada de su próxima activación.
- `cola_recursos_tomados`: es una cola que almacena todos los mutexes tomados junto con su thread propietario. Es utilizada para calcular el techo de prioridad global.

Además, para la implementación del algoritmo es necesario asociar con cada mutex una estructura de tipo `datos_mutex` como la mostrada a continuación:

```
struct datos_mutex {
  struct datos_thread * propietario;
  cola_t cola_threads_registrados;
};
```

El campo `cola_threads_registrados` mantiene, ordenados en función de su plazo de ejecución, todos los threads que en algún momento van a utilizar el mutex. Esta cola es reordenada cada vez que un thread finaliza su ejecución correspondiente a la activación actual. Por su parte, el campo `propietario` permite identificar el thread que tiene tomado el mutex en cada momento. El planificador asocia una estructura de este tipo a cada nuevo mutex mediante la función `posix_appsched_mutex_setspecific()` y la obtiene a partir del identificador del mutex utilizando `posix_appsched_mutex_getspecific()`.

También es necesario asociar con cada thread una estructura de datos que contenga la información requerida por el algoritmo. Esta información, representada mediante una estructura `datos_thread`, contiene los parámetros de planificación del thread, como son su periodo, plazo e instante de activación, y una lista con todos los mutexes que pretende utilizar en algún momento.

```
struct datos_thread {
  struct timespec periodo, plazo, instante_activación;
  lista_t mutexes_utilizados;
};
```

Para asignar y obtener la estructura `datos_thread` asociada con cada thread el planificador utiliza las funciones `pthread_setspecific_for()` y `pthread_getspecific_from()`.

A continuación mostramos el pseudocódigo del thread planificador. Con el fin de clarificar el ejemplo, procederemos a ocultar algunos detalles de la utilización de la interfaz que ya fueron mostrados en el ejemplo correspondiente al protocolo PCP del apartado anterior.

```
void *planificador_edf_tpd (void *arg)
{
  ...;
  while (1) {
    /* Acciones de activación y suspensión de threads */
    thread_siguiente = cabeza (cola_threads_activos);
    if (thread_siguiente != thread_actual) {
      if (thread_siguiente != NULL)
```

```
      Añade la acción de activación de thread_siguiente;
    if (thread_actual != NULL)
      Añade la acción de suspensión de thread_actual);
    thread_actual = thread_siguiente;
  }
  /* Ejecuta acciones de planificación */
  posix_appsched_execute_actions
     (&acciones, NULL,
      cabeza (cola_threads_suspendidos).instante_activación,
      &hora_actual, &evento);
  /* Procesa eventos de planificación */
  switch (evento.event_code) {
  case POSIX_APPSCHED_NEW:
    Obtiene los parámetros de planificación específicos;
    Crea y asocia la estructura de control ("d_thread");
    inserta (d_thread, cola_threads_activos);
    Añade la acción de aceptación en la lista de acciones;
    break;

  case POSIX_APPSCHED_EXPLICIT_CALL:
    // Un thread ha finalizado el trabajo correspondiente
    // a la activación actual
    Obtiene los datos asociados con el thread ("d_thread");
    extrae (d_thread, cola_threads_activos)
    inserta (d_thread, cola_threads_suspendidos);
    Calcula siguiente instante de activación y plazo de ejecución;
    Para cada mutex en d_thread.mutexes_utilizados:
      Reordena (mutex.cola_threads_registrados);
    break;

  case POSIX_APPSCHED_TIMEOUT:
    // Se ha alcanzado el instante de activación de un thread
    Obtiene los datos asociados con el thread ("d_thread");
    extrae (d_thread, cola_threads_suspendidos)
    inserta (d_thread, cola_threads_activos);
    break;

  case POSIX_APPSCHED_INIT_MUTEX:
    Obtiene los parámetros de planificación específicos;
    Crea y asocia la estructura de control del mutex ("d_mutex");
    Para cada thread registrado:
      inserta (d_mutex, thread.mutexes_utilizados);
    Añade acción de aceptación del mutex a la lista de acciones;
    break;

  case POSIX_APPSCHED_LOCK_MUTEX:
    // Una thread de aplicación trata de tomar un mutex libre
    Obtiene los datos asociados con el thread y
                                  el mutex (d_thread y d_mutex);
    if (bloqueo_por_techo_global (d_thread, &m_bloqueante)) {
      // Bloquea thread en el mutex
      inserta_heredada ({m_bloqueante.propietario, m_bloqueante},
                        cola_threads_activos);
    } else {
```

```
        // El thread puede tomar el mutex
        inserta ({d_mutex, d_thread}, cola_recursos_tomados);
        Añade acción de toma de mutex;
      }
      break;

    case POSIX_APPSCHED_TRY_LOCK_MUTEX:
      Obtiene los datos asociados con el thread y
                                    el mutex (d_thread y d_mutex);
      if (bloqueo_por_techo_global (d_thread)) {
        // El thread no puede tomar el mutex
        Añade acción de activación del thread;
      } else {
        // El thread puede tomar el mutex
        inserta ({d_mutex, d_thread}, cola_recursos_tomados);
        Añade acción de toma de mutex;
      }
      break;

    case POSIX_APPSCHED_UNLOCK_MUTEX:
      // El thread libera el mutex
      Obtiene los datos asociados con el thread y
                                    el mutex (d_thread y d_mutex);
      extrae ({d_mutex, d_thread}, cola_recursos_tomados);
      if (cabeza (cola_threads_activos) es una entrada heredada)
        extrae_cabeza (cola_threads_activos);
      break;

    case POSIX_APPSCHED_BLOCK_AT_MUTEX:
      // El mutex ya se encuentra en posesión de otro thread
      Obtiene los datos asociados con el thread y
                                    el mutex (d_thread y d_mutex);
      inserta_heredada ({d_mutex.propietario, d_mutex},
                        cola_threads_activos);
      break;
    } // switch
  } // while (1)
}
```

En este ejemplo, tanto para los threads planificados como para los mutexes es necesario utilizar unos parámetros de planificación específicos del algoritmo de planificación. En el caso de los threads, el único dato que precisa conocer su planificador es su periodo (por simplicidad se suponen plazos iguales a periodos). En el caso de los mutexes, será necesario informar al planificador de la lista de threads que les van a utilizar, para que pueda configurar las estructuras necesarias para la gestión del algoritmo.

Para un funcionamiento correcto, este algoritmo exige conocer desde el principio el número de threads y mutexes así como toda la información referente a la relación entre ellos. Sin embargo, dicha información no estará disponible para el planificador hasta que todos los mutexes hayan sido creados por el thread principal. Este problema se puede resolver sin más que asignar a este thread una prioridad mayor que la de los threads planificador y planificados. De esta forma, una vez que el planificador se encuentre en disposición de ejecutar tendrá pendientes todos los eventos correspondientes a la creación de threads y mutexes. El planificador procesará todos los

eventos pendientes de forma consecutiva, sin que pueda comenzar la ejecución de ningún thread planificado antes de que haya realizado la configuración de todas sus estructuras internas.

A continuación mostramos el pseudocódigo correspondiente a la creación del thread planificador, de un thread planificado y de un mutex de protocolo de techo de prioridad dinámico:

```
int main ()
{
  ...
  // Creación del thread planificador de aplicación
  Asigna atributos: SCHED_FIFO y PTHREAD_APPSCHEDULER;
  pthread_create (planificador_edf_tpd);

  // Creación de un thread de política definida por la aplicación
  Asigna atributos: SCHED_APP, planificador_id y periodo;
  Asigna parámetros específicos del algoritmo de planificación;
  pthread_create (pthread_planificado);

  Creación de otros threads planificados...;
  ...

  // Creación de un mutex con protocolo definido por la aplicación
  Asigna atributos: PTHREAD_APPSCHED_PROTOCOL y planificadord_id;
  Asigna parámetros específicos del protocolo de sincronización;
  pthread_mutex_init (&mutex_tpd);

  Creación de otros mutexes de protocolo de aplicación...;
  ...

  El thread main se suspende;
} // main
```

Un extracto del código de un thread EDF que utiliza un mutex de protocolo de techo de prioridad dinámico podría ser el mostrado a continuación:

```
// Cuerpo de un thread EDF
void * pthread_planificado (void * arg)
{
  while (1) {
    ...
    pthread_mutex_lock (&mutex_tpd);
    uso_del_recurso ();
    pthread_mutex_unlock (&mutex_tpd);
    ...
    // Informa a su planificador que ha terminado el
    //                trabajo correspondiente a su activación actual
    pthread_appsched_invoke_scheduler();
  }
}
```

5.5.3. Protocolo de no expulsión

En este ejemplo implementaremos el protocolo de "No expulsión". Se trata de un protocolo muy sencillo con el que pretendemos ilustrar el modo de utilización de la versión Ada de la interfaz. Con este protocolo, el acceso mutuamente exclusivo a los recursos se consigue evitando que las tareas sean expulsadas del procesador mientras se encuentran ejecutando alguna sección crítica. En el ejemplo, la forma elegida para implementar la no expulsión será elevando la prioridad de las tareas en posesión de uno o más mutexes a un nivel mayor que el de todas las tareas planificadas por el planificador.

Además del comportamiento básico descrito, nuestro protocolo permitirá indicar si se desea que las interrupciones permanezcan deshabilitadas durante el tiempo en que un mutex se encuentra tomado por una tarea. Si se elige esta opción, el mutex podrá ser utilizado para sincronizarse con un manejador de interrupción. El comportamiento elegido para el mutex se especifica en el momento de su creación en base a sus parámetros de planificación específicos. Para ello se crea el tipo `Clase_Mutex` y se instancia para ese tipo el paquete genérico `POSIX_Mutexes.Application_Defined_Parameters`:

```
type Clase_Mutex is (Interrumpible, No_Interrumpible);

package Parametros_Mutexes is
  new POSIX_Mutexes.Application_Defined_Parameters
                                        (Parameters => Clase_Mutex);
```

Para la implementación del protocolo, será necesario asociar con cada tarea planificada una estructura de control que permita almacenar su prioridad base y el número de mutexes, tanto interrumpibles como no interrumpibles, que se encuentran en su poder. La asociación se realiza utilizando el paquete estándar `Ada.Task_Attributes`.

```
type Datos_Tarea is record
  Prioridad_Base : System.Any_Priority;
  Mutexes_No_Interrumpibles_Tomados : Natural;
  Mutexes_Interrumpibles_Tomados : Natural;
end record;

package Atributos_Tarea is
  new Ada.Task_Attributes (Attribute => Datos_Tarea);
```

Por simplicidad no se permite la suspensión de las tareas mientras se encuentran ejecutando una sección crítica. Con ello se asegura que siempre que una tarea trate de tomar un mutex le encontrará libre. Esta restricción hace que no sea preciso asociar ninguna estructura de control con cada mutex, puesto que no es necesario saber cuando un mutex está tomado ni gestionar colas de tareas bloqueadas asociadas a los mutexes. También debido a esta restricción, nuestro planificador no necesita tratar eventos de tipo `BLOCK_AT_MUTEX` ya que nunca podrán producirse.

En nuestro ejemplo el planificador es capaz de comprobar que las tareas cumplen la condición de no suspensión dentro de las secciones críticas. La violación de esta restricción se detecta al comprobar que se ha producido un evento de tipo `BLOCK` para una tarea que tiene algún mutex en su poder. En el sencillo protocolo implementado, esta situación constituye un error fatal que implicaría la terminación del planificador.

El pseudocódigo detallado de la tarea planificadora se muestra a continuación:

```
task body Tarea_Planificadora is
begin
  --  Lazo de atención de eventos
  loop
    --  Ejecuta acciones de planificación y espera siguiente evento
    Execute_Actions (Acciones, Evento);
    case Get_Event_Code (Evento) is
      when NEW_TASK =>
        --  Asigna los datos asociados a la tarea
        Nueva_Tarea := ...;
        Atributos_Tarea.Set_Value (Get_Task (Evento), Nueva_Tarea);
        --  Acepta y activa tarea
        Add_Accept (Acciones, Get_Task (Evento));
        Add_Activate (Acciones, Get_Task (Evento));

      when READY =>
        -- Únicamente es necesario activar la tarea
        Add_Activate (Acciones, Get_Task (Evento));

      when BLOCK =>
        -- Obtiene los datos asociados a la tarea
        D_Tarea := Atributos_Tarea.Value (Get_Task (Evento));
        -- Detección de suspensión en sección crítica
        if la tarea tiene mutexes en su poder then
          Error fatal: una tarea se ha bloqueado mientras se
                       encuentra ejecutando una sección crítica.
        end if;

      when INIT_MUTEX =>
        -- Siempre se acepta el mutex
        Add_Accept_Mutex (Acciones, Get_Mutex (Evento));

      when LOCK_MUTEX | TRY_LOCK_MUTEX =>
        -- Obtiene la clase del mutex
        Parametros_Mutexes.Get_Parameters (Get_Mutex (Evento),
                                           Clase);
        -- Obtiene los datos asociados a la tarea
        D_Tarea := Atributos_Tarea.Value (Get_Task (Evento));
        if Clase := No_Interrumpible then
          D_Tarea.Mutexes_No_Interrumpibles_Tomados :=
                    D_Tarea.Mutexes_No_Interrumpibles_Tomados + 1;
          -- Deshabilita interrupciones
          Hardware_Interrupts.Disable_All;
        else
          D_Tarea.Mutexes_Interrumpibles_Tomados :=
                        D_Tarea.Mutexes_Interrumpibles_Tomados + 1;
        end if;
        -- Eleva la prioridad de la tarea
        Ada.Dynamic_Priorities.Set_Priority (Max_Priority,
                                             Get_Task (Evento));
        -- Concede el mutex a la tarea
        Add_Lock_Mutex (Acciones,
                        Get_Task (Evento),
                        Get_Mutex (Evento));
```

```
      when UNLOCK_MUTEX =>
        -- Obtiene la clase del mutex
        Parametros_Mutexes.Get_Parameters (Get_Mutex (Evento),
                                           Clase);
        -- Obtiene los datos asociados a la tarea
        D_Tarea := Atributos_Tarea.Value (Get_Task (Evento));
        -- Decrementa el número de mutexes tomados
        if Clase := No_Interrumpible then
          D_Tarea.Mutexes_No_Interrumpibles_Tomados :=
                    D_Tarea.Mutexes_No_Interrumpibles_Tomados - 1;
          if D_Tarea.Mutexes_No_Interrumpibles_Tomados = 0 then
            -- Habilita interrupciones
            Hardware_Interrupts.Disable_All;
          end if;
        else
          D_Tarea.Mutexes_Interrumpibles_Tomados :=
                        D_Tarea.Mutexes_Interrumpibles_Tomados - 1;
        end if;
        -- Si a la tarea no le quedan más mutexes en su poder,
        -- se rebaja su prioridad al valor original
        if Tarea no tiene mutexes en su poder then
          Ada.Dynamic_Priorities.Set_Priority
                        (D_Tarea.Prioridad_Base, Get_Task (Evento));
        end if;

      when others => null;
    end case;
  end loop;
end Tarea_Planificadora;
```

Los pasos necesarios para crear un mutex de protocolo definido por la aplicación se muestran a continuación. En el caso de que el planificador rechazara el mutex, se elevaría una excepción `POSIX_Error` en la llamada al procedimiento `POSIX_Mutexes.Initialize`.

```
begin
  ...;
  POSIX_Mutexes.Initialize (Attr);
  POSIX_Mutexes.Set_Locking_Policy
    (Attr, POSIX_Mutexes.APPSCHED_PROTOCOL);
  POSIX_Mutexes.Set_App_Scheduler
    (Attr, Tarea_Planificadora'Identity);
  Parámetros_Mutexes.Set_Parameters (Attr, Interrumpible);
  POSIX_Mutexes.Initialize (Mutex, Attr);
  Mutex_De_Aplicación := POSIX_Mutexes.Descriptor_Of (Mutex);
  ...;

exception
  when POSIX_Error =>
    Error: mutex no aceptado por el planificador;
end;
```

A continuación mostramos el pseudocódigo correspondiente a una tarea planificada por la aplicación que utiliza un mutex de protocolo de no expulsión. Como puede apreciarse, se trata de una tarea Ada normal que requiere ser planificada por el planificador de aplicación mediante

la llamada al procedimiento `Change_Task_Policy_To_App_Sched`. En el caso de que fuera rechazada se elevaría la excepción `POSIX_Error`. Posteriormente, la tarea hace uso del mutex de protocolo definido por la aplicación de igual manera que si se tratara de un mutex de protocolo estándar.

```
task body Tarea_De_Aplicación is
begin
  --  Requiere ser planificada por la tarea planificadora
  Change_Task_Policy_To_App_Sched (Tarea_Planificadora'Identity);
  loop
    ...;
    POSIX_Mutexes.Lock (Mutex_De_Aplicación);
    -- Utiliza el recurso
    ...;
    POSIX_Mutexes.Unlock (Mutex_De_Aplicación);
    ...;
  end loop;

exception
  when POSIX_Error =>
    Error: tarea no aceptada por el planificador;
end Tarea_De_Aplicación;
```

5.6. Implementación en el núcleo

La implementación en el núcleo de MaRTE OS del soporte para la parte de la interfaz descrita en este capítulo resulta bastante sencilla, ya que se apoya en gran medida en el soporte general para la gestión de eventos descrito en el apartado 4.6. El soporte implementado puede dividirse en dos partes fundamentales: la generación de eventos de herencia y pérdida de prioridades debido al uso de mutexes del sistema y la gestión de los mutexes de protocolo definido por la aplicación. Los cambios realizados únicamente han afectado a los paquetes `Kernel.Mutexes` y `Kernel.Mutexes.Internals`.

Como expusimos en la descripción del modelo, la gestión de los mutexes de protocolo definido por la aplicación descansa en gran medida en el planificador de aplicación. Él es el encargado de decidir cuál, de entre todas las tareas bloqueadas en un mutex, será la siguiente en tomarlo y en qué momento debe hacerlo. En la gestión de este tipo de mutexes, el sistema operativo únicamente se encarga de asegurar la exclusión mutua y de comprobar que el mutex no sea entregado a una tarea que no se encuentre bloqueada en él. Para desempeñar esta labor, el sistema operativo solamente utiliza el campo `Owner` (propietario) de la estructura de control de los mutexes y el campo `AppSched_Mutex_Where_Waiting` (mutex de protocolo de aplicación en el que una tarea se encuentra bloqueada) de la estructura de control de las tareas. El pseudocódigo correspondiente a la operación `Lock` sobre un mutex de protocolo definido por la aplicación se muestra a continuación:

```
  Lock (M : Mutex; T : Tarea):
    if M.Protocol = APPSCHED_PROTOCOL then
      T.AppSched_Mutex_Where_Waiting := M;
      if M.Owner = null then
        Genera_Evento_De_Planificación (APPSCHED_LOCK_MUTEX);
      else
        Genera_Evento_De_Planificación (APPSCHED_BLOCK_AT_MUTEX);
```

```
      end if;
      Bloquea_Tarea (T);
    else -- M.Protocol /= APPSCHED_PROTOCOL
      Acciones normales de toma del mutex...;
```

Tras la ejecución de la operación `Lock`, la tarea se queda bloqueada en el mutex identificado por su campo `AppSched_Mutex_Where_Waiting`. En un instante posterior, el planificador puede decidir entregar el mutex mediante la ejecución de una acción de entrega de mutex. Ante esa decisión por parte del planificador, el sistema operativo únicamente deberá verificar que la tarea se encontraba bloqueada en el mutex y que éste se encuentra libre. El pseudocódigo correspondiente a la ejecución de una de estas acciones por parte del sistema operativo es el siguiente:

```
Ejecuta acción de entrega de mutex (M : Mutex; T : Tarea)
  if T.AppSched_Mutex_Where_Waiting /= M  or  M.Owner /= null then
    Error;
  else
    T.AppSched_Mutex_Where_Waiting := null;
    M.Owner := T;
    Activa_Tarea (T)
  end if;
```

La ejecución de la operación `Unlock` de un mutex planificado únicamente requiere del sistema operativo la generación del evento correspondiente y la modificación del campo `Owner`:

```
Unlock (M : Mutex; T : Tarea):
  if M.Protocol = APPSCHED_PROTOCOL then
    M.Owner := null;
    Genera_Evento_De_Planificación (APPSCHED_UNLOCK_MUTEX);
  else -- M.Protocol /= APPSCHED_PROTOCOL
    Acciones normales de liberación del mutex...;
```

Es importante resaltar que con mutexes de protocolo definido por la aplicación no se utilizan la mayoría de las estructuras relacionadas con los mutexes de sistema. Así, al no provocar herencia de prioridad, los mutexes de aplicación no se encolan en la cola de mutexes tomados por una tarea, ya que la función de la citada cola es precisamente el cálculo de la prioridad heredada. Tampoco se utiliza la cola de tareas bloqueadas asociada a cada mutex, puesto que esta cola es utilizada por el sistema operativo para calcular el techo de prioridad de los mutexes de protocolo `PTHREAD_PRIO_INHERIT` y para obtener la siguiente tarea que tomará el mutex, decisión que en estos mutexes es tomada por el planificador de aplicación.

En lo referente a la utilización de mutexes del sistema por parte de tareas planificadas por la aplicación, la implementación del comportamiento descrito en el modelo ha resultado extremadamente sencilla. En este caso no existe ningún comportamiento particular en las operaciones de toma y liberación de un mutex, siendo iguales independientemente de que la tarea que las realiza esté planificada por la aplicación o por el propio sistema operativo. La única modificación ha consistido en generar los eventos de herencia y pérdida de prioridad cuando tales operaciones son realizadas por una tarea de política definida por la aplicación.

5.7. Prestaciones

En la tabla 5.8 mostramos los tiempos de toma y liberación de un mutex de protocolo definido por la aplicación desde una aplicación C. Las medidas han sido realizadas en un computador con procesador Pentium III a 1.1GHz.

Tabla 5.8: Tiempo de toma y liberación de mutexes de protocolo definido por la aplicación

Descripción Medida	Tiempo (μs)
Toma de un mutex	1.25
Liberación de un mutex	1.1

La primera medida corresponde al intervalo de tiempo transcurrido desde que un thread invoca la operación `pthread_mutex_lock()` para tratar de tomar el mutex, hasta que se encuentra de nuevo en ejecución con el mutex en su poder. Como mostrábamos en la figura 5.2, la toma del mutex es una operación compleja en la que interviene el thread planificador de aplicación. En primer lugar, la invocación de `pthread_mutex_lock()` sobre un mutex libre provoca la generación del evento `POSIX_APPSCHED_LOCK_MUTEX`, con la consiguiente activación del thread planificador y el bloqueo del thread planificado. El planificador puede, mediante una acción de entrega del mutex, entregar el mutex al thread bloqueado en el momento que lo desee. En el experimento realizado para nuestra medida, la entrega se produce de forma inmediata tras la llegada del evento. La ejecución de la citada acción supone la suspensión del planificador y la reactivación del thread planificado ya con el mutex en su poder. En resumen, la toma del mutex consiste en la generación de un evento de planificación, un cambio de contexto del thread planificado al planificador, la ejecución de una acción de planificación y un nuevo cambio de contexto ahora del thread planificador al planificado.

Por su parte, la operación de liberación del mutex (`pthread_mutex_unlock()`) presenta una complejidad similar a la anterior, incluyendo también dos cambios de contexto, la generación de un evento y la ejecución de una acción de planificación.

La complejidad anteriormente descrita para las operaciones de toma y liberación de mutexes de protocolo definido por la aplicación, es mucho mayor que la de esas mismas operaciones cuando se realizan sobre mutexes de alguno de los protocolos definidos en el estándar POSIX. Así, tal y como exponíamos en el apartado 3.8.1, el tiempo empleado en la toma y posterior liberación de un mutex de protocolo de techo de prioridad inmediato es sólo de 0.34μs (0.53μs en el caso de que haya que realizar el cambio de prioridad diferido).

Sin embargo, esta misma complejidad, principalmente debida a la necesidad de generar un evento cada vez que una tarea realiza una operación sobre un mutex de aplicación, es lo que proporciona generalidad a nuestra solución, permitiendo implementar protocolos que no son implementables con otras soluciones existentes en la bibliografía. Pensamos que para algunos tipos de protocolos parte de esta complejidad podría evitarse, con la consiguiente mejora en las prestaciones finales. Esta mejora constituye una línea de trabajo futura que será esbozada en el capítulo 6, “Conclusiones y trabajo futuro”.

5.8. Conclusiones

Se ha diseñado una interfaz que amplía y completa la expuesta en el capítulo 4, “Interfaz de usuario para la definición de algoritmos de planificación” permitiendo a las aplicaciones definir sus propios protocolos de sincronización. Se han presentado dos versiones equivalentes de la citada interfaz, una escrita en lenguaje C y otra en lenguaje Ada, ambas integradas en el conjunto de interfaces POSIX. Con el fin de proceder a su validación, la interfaz ha sido implementada en el sistema operativo MaRTE OS (presentado en el capítulo 3 de esta memoria). Además se han desarrollado varios protocolos de sincronización que han permitido comprobar la validez y generalidad de la interfaz a la hora de implementar protocolos de sincronización.

La interfaz presentada en este capítulo completa la descrita en el capítulo anterior, en el sentido de que con ella se resuelve una de las principales limitaciones presentadas por la mayoría de las soluciones para planificación flexible existentes en la bibliografía: la inexistencia de un mecanismo general que permitiera definir protocolos de sincronización. Por el contrario, el esquema utilizado en nuestro caso, basado en mutexes especiales que provocan la invocación del planificador cada vez que son utilizados por una tarea, constituye un mecanismo general que permite implementar gran variedad de protocolos[1]. En este aspecto, nuestra propuesta supera otras soluciones como las basadas en la herencia de CPU ya que permite la implementación de todo tipo de algoritmos no sólo aquellos basados en la herencia de prioridad.

La generalidad de la solución adoptada se ve incrementada por el hecho de que con nuestra solución un mutex no es entregado automáticamente a la tarea que trata de tomarlo, aún en el caso de que el recurso se encuentre libre. En lugar de eso el sistema operativo invoca al planificador, de forma que sea éste el que decida conceder el recurso o bloquear a la tarea en él. Esta característica permite implementar algoritmos como el de techo de prioridad global, en los que la concesión de los recursos no depende únicamente del estado del mutex sino que lo hace también de algún otro parámetro global del sistema. En este sentido nuestra interfaz también supera a la herencia de CPU, con la que no sería posible implementar este tipo de protocolos puesto que no permite imponer ninguna limitación a la toma de un mutex libre.

Otra ventaja fundamental que nuestra interfaz presenta sobre las demás soluciones existentes en la bibliografía radica en su compatibilidad e integración con el estándar POSIX. El modelo en el que se basa es compatible con los protocolos de sincronización definidos en ese estándar, permitiéndose que tareas planificadas por la aplicación utilicen mutexes de protocolos estándar. Esta característica permite la sincronización de tales tareas con otras planificadas bajo alguna de las políticas estándar, y además, las permite utilizar librerías de propósito general que hagan uso ese tipo de mutexes. Se definen eventos de planificación que informan a las tareas planificadoras sobre la utilización de tales mutexes por sus tareas planificadas. Con esa información, los planificadores pueden tomar las acciones de planificación oportunas para reducir los efectos de inversión de prioridad sufridos por las demás tareas del sistema que comparten el recurso global.

1. A fecha de escritura de esta memoria, no se ha encontrado ninguna clase de protocolos de sincronización que no sea implementable con la interfaz presentada.

6. Conclusiones y trabajo futuro

6.1. Conclusiones

Esta tesis se ha centrado en la mejora de los mecanismos proporcionados por los sistemas operativos para la planificación de tareas en sistemas empotrados de tiempo real. Las principales aportaciones de este trabajo son las enumeradas a continuación:

- Se ha desarrollado una interfaz que permite a las aplicaciones definir los algoritmos de planificación bajo los cuales pretenden que sean planificadas sus tareas. La interfaz se integra en el conjunto de interfaces POSIX.
- Se ha extendido la interfaz para que englobe la gestión de recursos compartidos, permitiendo a las aplicaciones definir los protocolos de sincronización que mejor se adaptan a cada política de planificación.
- Se ha diseñado e implementado el sistema operativo MaRTE OS, escrito en lenguaje Ada y conforme con el perfil mínimo de sistema de tiempo real definido en el estándar POSIX.13. La existencia de este sistema operativo nos ha permitido:
  - Evaluar el tamaño y prestaciones del perfil mínimo de sistema de tiempo real.
  - Evaluar la complejidad que supone añadir al citado perfil nuevos servicios que consideramos interesantes para los sistemas empotrados de tiempo real.
  - Implementar y probar las interfaces para planificación flexible desarrolladas.
  - Evaluar la posibilidad y posibles ventajas de utilizar Ada 95 para implementar un sistema operativo sobre el que puedan ejecutar aplicaciones escritas en Ada, C o una mezcla de ambos.

En los siguientes apartados procederemos a comentar de forma detallada cada uno de los resultados obtenidos.

Diseño e implementación de un sistema operativo de tiempo real

Se ha desarrollado el sistema operativo MaRTE OS (“Minimal Real-Time Operating System for Embedded Applications”), un núcleo de tiempo real para sistemas empotrados cuyas principales características son:

- Implementa la funcionalidad descrita en el perfil de sistema de tiempo real mínimo definido en el estándar POSIX.13.
- Permite ejecutar aplicaciones escritas tanto en Ada como en C proporcionando las interfaces POSIX para ambos lenguajes.
- Está escrito utilizando el lenguaje Ada 95.
- Sobre él se ha adaptado la librería de tiempo de ejecución de compilador GNAT, lo que permite a las aplicaciones Ada utilizar toda la semántica de tareas proporcionada por este lenguaje.

Junto con el sistema operativo se ha desarrollado un entorno de desarrollo cruzado basado en Linux, en los compiladores GCC y GNAT y en un conjunto de archivos de órdenes que automatizan el proceso de creación, carga y depuración de las aplicaciones.

Se han realizado medidas para obtener datos sobre el tamaño y prestaciones del núcleo, obteniéndose valores similares a los ofrecidos por otros sistemas operativos. Los resultados obtenidos resultan más destacables aún al considerar que muchas de las comparaciones se han realizado con sistemas operativos comerciales, en los que seguramente sus fabricantes hayan realizado un gran esfuerzo de optimización a lo largo de muchos años.

MaRTE OS constituye una de las primeras implementaciones del perfil mínimo de sistema de tiempo real. Por consiguiente, y a la vista de los resultados obtenidos, representa una de las primeras pruebas de que un núcleo con la funcionalidad descrita por el perfil de sistema de tiempo real mínimo definido en el estándar POSIX.13 resulta apropiado para sistemas empotrados. De la misma forma, al ser uno de los pocos sistemas operativos escritos en Ada, MaRTE OS también demuestra que es posible implementar un sistema operativo de esas características utilizando el lenguaje de programación Ada 95, para así aprovechar sus importantes ventajas, sobre todo en lo referente a la fiabilidad.

Otro importante resultado obtenido tiene que ver con la implementación en MaRTE OS de algunos de los nuevos servicios que, dentro del actual proceso de revisión del estándar POSIX.13, han sido propuestos para su incorporación en el perfil mínimo de sistema de tiempo real. Los servicios implementados han sido el reloj monótono, la operación para suspensión absoluta de alta resolución, la política de planificación de servidor esporádico y los relojes y temporizadores de tiempo de ejecución. Esto nos ha permitido obtener información sobre el impacto que su incorporación supone en el tamaño, complejidad y prestaciones de un núcleo conforme con el subconjunto mínimo. Tanto en el caso de la implementación del reloj monótono, como en el de la operación de suspensión absoluta de alta resolución, el impacto es mínimo. Por su parte, la implementación de la política de planificación de servidor esporádico y de los relojes y temporizadores de tiempo de ejecución ha supuesto un aumento apreciable, aunque no excesivo, de la complejidad del núcleo, el cual creemos totalmente asumible comparado con el importante aumento de funcionalidad que supone su incorporación.

A fecha de escritura de esta memoria, MaRTE OS ya constituye un producto totalmente operativo, que se distribuye como software libre bajo licencia “GNU General Public License” (GPL), encontrándose disponible en `http://marte.unican.es`. En la actualidad, MaRTE OS está siendo utilizado en proyectos de docencia, investigación y desarrollo realizados tanto en el grupo de “Computadores y Tiempo Real” de la Universidad de Cantabria, como en otras universidades de España y del resto del mundo, entre los que cabría citar: “Departamento Sistemas Informáticos y Computación”, “Departamento Informática de sistemas y computadores” y “Departamento Ingeniería de sistemas y automática” (Universidad Politécnica de Valencia), “Departamento de Ciencias Matemáticas e Informática” (Universidad de las Islas Baleares), “Departamento de Lenguajes y Ciencias de la Computación” (Universidad de Málaga), “Departamento de Ingeniería Telemática” (Universidad de Vigo), “Instituto Universitario de Microelectrónica Aplicada” (Universidad de Las Palmas de Gran Canaria), “Department of Computer Science” (University of York), “Department of Computer Science” (University of Northern Iowa) y “École d'ingénieurs de Genève” (Université de Genève).

Interfaz para la planificación definida por la aplicación

Se ha diseñado una interfaz que permite a las aplicaciones definir los algoritmos de planificación bajo los cuales desean planificar sus tareas. Se han desarrollado dos versiones

equivalentes de la citada interfaz, una escrita en lenguaje C y otra en lenguaje Ada, ambas integradas en el conjunto de interfaces POSIX.

El modelo en el que se apoya la interfaz está basado en la existencia de un tipo especial de tareas que son las encargadas de implementar los algoritmos de planificación definidos por la aplicación. Estas tareas, denominadas planificadoras, tienen el poder de activar o suspender una o varias de sus tareas planificadas. Además, las tareas planificadoras son activadas por el sistema operativo cada vez que ocurre cualquier circunstancia que pudiera requerir la toma de alguna decisión de planificación.

La solución desarrollada aúna las mejores cualidades de las soluciones existentes en la bibliografía, como son la ejecución de los algoritmos en una tarea de usuario o la invocación del algoritmo en cada punto de planificación. A la vez, nuestra solución supera sus principales carencias, como la falta de generalidad, la dificultad de implementación en sistemas multiprocesadores o la falta de aislamiento entre planificadores y sistema operativo. Las principales características de nuestra solución son las siguientes:

- *Constituye una solución general*. Permitiendo la implementación de una amplísima variedad de algoritmos de planificación[1].
- *Permite la implementación de algoritmos de planificación para sistemas multiprocesadores*. Gracias principalmente a la habilidad de los planificadores de activar o suspender varias de sus tareas planificadas de forma simultánea.
- *Permite proteger el sistema operativo ante fallos en un algoritmo de planificación*. Permitiendo que las tareas planificadoras sean ejecutadas fuera del núcleo del sistema operativo, compartiendo espacio de direcciones que el resto de las tareas de la aplicación.
- *Persigue su integración en el estándar POSIX*. El modelo en el que se basa es compatible con las políticas de planificación definidas en los estándares POSIX y Ada 95.

Con el fin de proceder a la validación de la interfaz, se ha procedido a su implementación en el sistema operativo MaRTE OS, desarrollándose con ellas varios algoritmos de planificación que han servido para su prueba.

Como se desprende de los ejemplos realizados, la utilización de la interfaz no complica de forma significativa el código del algoritmo de planificación, permitiendo la escritura de planificadores con una estructura clara y fácil de seguir. Su uso tampoco supone un aumento considerable del número de líneas de código necesarias para la escritura de un algoritmo de planificación.

Midiendo las prestaciones de algunos de los algoritmos de planificación implementados se ha calculado que la penalización provocada por la interfaz es aproximadamente 3 veces superior que la correspondiente al planificador interno de MaRTE OS. Este aumento resulta claramente asumible si se considera que la sobrecarga rondaría el 0.5% del periodo de ejecución de una tarea periódica de 1KHz. La comparación con los resultados proporcionados por otras soluciones también resulta favorable, sobre todo si se considera la mayor generalidad de nuestra solución.

1. A fecha de escritura de esta memoria, no se ha encontrado ningún tipo de algoritmos de planificación que no sea implementable con la interfaz presentada.

Extensión de la interfaz: protocolos de sincronización definidos por la aplicación

La utilización conjunta de una política de planificación con un protocolo de sincronización inadecuado puede producir efectos perjudiciales e incluso fatales para la planificabilidad del sistema. Por consiguiente, una solución general al problema de la planificación flexible debe permitir definir, junto a cada política de planificación, el protocolo de sincronización que mejor se adapte a ella. Con ese propósito, se ha extendido la interfaz para planificación definida por la aplicación de forma que permita a las aplicaciones definir sus propios protocolos de sincronización.

La solución adoptada se basa en la existencia de un tipo especial de mutexes para los que el sistema operativo no decide la tarea ni el momento de su entrega, sino que deja esta decisión en manos de las tareas planificadoras de aplicación. Cualquier acción realizada sobre un mutex de este tipo provoca la generación de un evento para su tarea planificadora, lo que la permite tomar las decisiones de planificación pertinentes en los momentos oportunos. Esta característica hace que nuestra solución constituya un mecanismo general que permite implementar gran variedad de protocolos[1]. Precisamente era la falta de generalidad la principal limitación que presentaban la mayoría de las soluciones existentes en la bibliografía, como es el caso de las basadas en la herencia de CPU.

La generalidad de la solución adoptada se ve incrementada por el hecho de que con nuestra solución un mutex no es entregado automáticamente a la tarea que trata de tomarlo, aún en el caso de que el recurso se encuentre libre. En lugar de eso el sistema operativo invoca al planificador, de forma que sea éste el que decida conceder el recurso o bloquear a la tarea en él. Esta característica permite implementar protocolos en los que la concesión de los recursos no depende únicamente del estado del mutex sino que lo hace también de algún otro parámetro global del sistema. En este sentido nuestra interfaz también supera a otras soluciones como las basadas en la herencia de CPU.

Otra ventaja fundamental de nuestra interfaz sobre las demás soluciones existentes en la bibliografía radica en su compatibilidad e integración con el estándar POSIX. El modelo en el que se basa es compatible con los protocolos de sincronización definidos en el POSIX, permitiendo que tareas planificadas por la aplicación utilicen mutexes de protocolos estándar. Esta característica permite la sincronización de tales tareas con otras planificadas bajo alguna de las políticas definidas por el estándar POSIX, y además, las permite utilizar librerías de propósito general que hagan uso ese tipo de mutexes. Se definen eventos de planificación que informan a las tareas planificadoras sobre la utilización de tales mutexes por sus tareas planificadas. Con esa información, los planificadores pueden tomar las acciones de planificación necesarias para reducir la inversión de prioridad sufrida por las demás tareas del sistema que comparten el recurso global.

Se han desarrollado dos versiones equivalentes de la citada interfaz, una escrita en lenguaje C y otra en lenguaje Ada, ambas integradas en el conjunto de interfaces POSIX. Con el fin comprobar su validez y generalidad, la interfaz ha sido implementada en el sistema operativo MaRTE OS y se han escrito varios de protocolos de sincronización para su prueba.

La interfaz extendida, esto es, incluyendo tanto la parte de definición de algoritmos de planificación como la de protocolos de sincronización, esta siendo utilizada por el “Grupo de Computadores y Tiempo Real” de la Universidad de Cantabria. La investigación realizada se

1. A fecha de escritura de esta memoria, no se ha encontrado ninguna clase de protocolos de sincronización que no sea implementable con la interfaz presentada.

encuadra dentro del proyecto europeo FIRST [FIR02] y persigue la planificación conjunta de tareas planificadas bajo políticas basadas en prioridades, ya sean estáticas o dinámicas, con otras tareas planificadas por un ejecutivo cíclico [DOB01]. Asimismo, la interfaz está siendo implementada en el sistema operativo RT-Linux por el "Departamento de Ingeniería de Sistemas y Automática" de la Universidad Politécnica de Valencia.

En cuanto a su proceso de inclusión en el estándar POSIX, la interfaz se ha presentado en el grupo encargado de las extensiones de tiempo real del estándar, encontrándose en la actualidad en fase de estudio por los miembros del citado grupo.

6.2. Trabajo futuro

En los tres resultados principales de esta memoria existen líneas de trabajo abiertas. A continuación procedemos a esbozar las más importantes.

Sistema operativo MaRTE OS

Con la libre distribución de MaRTE OS hemos adquirido un compromiso de mantenimiento y mejora continuada del sistema con todos aquellos usuarios que han decidido utilizarlo para sus proyectos. Son innumerables los aspectos en los que un sistema operativo como MaRTE OS esta abierto a la ampliación o a la mejora, aunque quizá los que merecerían ser contemplados en primer[1] lugar podrían ser su adaptación a sistemas multiprocesadores, el desarrollo de un entorno que facilite la incorporación de manejadores ("drivers") de dispositivos, su portado a distintas plataformas (principalmente a microcontroladores) y el desarrollo de una librería gráfica.

En lo que es más propiamente trabajo de investigación, pretendemos seguir utilizando MaRTE OS como plataforma sobre la que evaluar los servicios que los sistemas operativos ofrecen a las aplicaciones empotradas de tiempo real. En particular resultaría muy interesante la utilización del sistema operativo en un buen número de aplicaciones industriales ya que esto nos permitiría evaluar que servicios, además de los incluidos en el perfil mínimo de sistema de tiempo real, resultarían útiles en los sistemas empotrados, o por el contrario, cuales de los incluidos podrían ser eliminados en posteriores revisiones del estándar.

Con esa idea de utilizar MaRTE OS en aplicaciones prácticas reales, pretendemos utilizar nuestro sistema operativo en el desarrollo de controladores de robots dentro del proyecto europeo FIRST [FIR02]. Asimismo, vamos a utilizar MaRTE OS para implementar el controlador de un robot industrial para la soldadura de los alojamientos de las barras de control en vasijas de centrales nucleares, dentro de un proyecto conjunto con la empresa Equipos Nucleares S.A.

Interfaz para la planificación definida por la aplicación

La interfaz desarrollada se integra en el conjunto de interfaces POSIX en sus versiones Ada y C, por lo que puede ser directamente utilizada por aplicaciones escritas en ambos lenguajes. Sin embargo, en el caso de las aplicaciones Ada, parece que su utilización resultaría más cómoda si se incluyera como una extensión del propio lenguaje, de forma similar a como el manual de referencia define los anexos especializados (anexo de tiempo real, anexo de sistemas distribuidos, etc.). Con esta segunda opción se podría ofrecer un conjunto de directivas al

1. Algunos de estos proyectos ya se están desarrollando a fecha de escritura de esta memoria.

compilador (“pragmas”) que, aunque no supondrían ningún aumento de funcionalidad con respecto a la interfaz presentada, sí que podrían simplificar su uso.

Otro aspecto en el que parece interesante profundizar es en el desarrollo de algoritmos de planificación para sistemas multiprocesadores. Este es un campo en el que queda mucho trabajo por realizar y en el que se podría sacar mucho partido a la interfaz desarrollada.

Protocolos de sincronización definidos por la aplicación

Sería muy interesante que las aplicaciones Ada pudieran utilizar objetos protegidos cuyo protocolo estuviese definido por la aplicación. Para ello debería desarrollarse una extensión del propio lenguaje que proporcionara directivas al compilador (“pragmas”) que permitieran indicar el planificador y demás propiedades de un objeto protegido con protocolo definido por la aplicación. La modificación de la librería de tiempo de ejecución podría resultar más o menos complicada dependiendo de la forma en que ésta implemente los objetos protegidos. En el caso de la librería de tiempo de ejecución del compilador GNAT, la modificación podría resultar sencilla puesto que únicamente utiliza un mutex para implementar cada objeto protegido [MIR02]. En consecuencia, parece que bastaría únicamente con hacer que dicho mutex fuera de protocolo `APPSCHED_PROTOCOL`.

La eficiencia en la gestión de los mutexes de protocolo definido por la aplicación es otro aspecto que, al menos en algunos casos, podría ser mejorado. Con el modelo actual, el planificador debe indicar que tarea toma el mutex. Esto es necesario para implementar protocolos en los que la toma de un mutex no depende únicamente de su estado, sino también de otros factores globales del sistema. Sin embargo, existen protocolos en los un mutex libre siempre se entrega a la tarea que trata de tomarlo. En esos caso, la entrega podría ser directamente realizada por el sistema operativo, sin que fuera necesaria la intervención de la tarea planificadora. Resultaría útil que la interfaz permitiera diferenciar entre ambos tipos de protocolos, de forma que fuera posible mejorar la eficiencia en los que permiten la entrega directa del mutex.

También sería interesante incluir en la interfaz la gestión de variables condicionales. Debería proporcionase a los planificadores algún mecanismo que les permitiera determinar el orden de activación de las tareas bloqueadas en una variable condicional que es señalizada.

Por último, al igual que ocurría para las políticas de planificación, también resultaría muy interesante desarrollar algunos protocolos de sincronización para sistemas multiprocesadores.

Proceso de estandarización de la interfaz

En la actualidad la interfaz constituye una propuesta que está siendo estudiada por el grupo encargado de las extensiones de tiempo real del estándar POSIX. Una vez superada esta fase debemos continuar con el proceso de estandarización, con el objetivo final de que el trabajo propuesto en esta tesis, con las modificaciones que necesariamente surgirán durante ese proceso, se incorpore como una funcionalidad opcional en este importante estándar internacional.

Anexo A. Propuesta de incorporación al estándar POSIX

Como anexo a esta memoria de tesis doctoral, incluimos el documento presentado al grupo encargado de las extensiones de tiempo real del POSIX en el que se describe nuestra interfaz para planificación y sincronización definidas por la aplicación. La propuesta se realizó en julio de 2002 y en la actualidad se encuentra en fase de estudio por parte del citado grupo.

Este documento constituye una definición más formal y acorde con el modelo de los documentos que componen los estándares POSIX de lo presentado en los capítulos 4 y 5 de esta memoria. En particular en este documento se describe el modelo en el que se basa nuestra solución, descrito en los apartados 4.2 y 5.2, y la versión C de la interfaz, que presentábamos en los apartados 4.3 y 5.3.

A.1. Model for Application-Defined Scheduling

In the proposed approach for application-defined scheduling, shown in Figure 1, each application scheduler is a special kind of thread, that is responsible of scheduling a set of threads that have been attached to it. This leads to two classes of threads in this context:

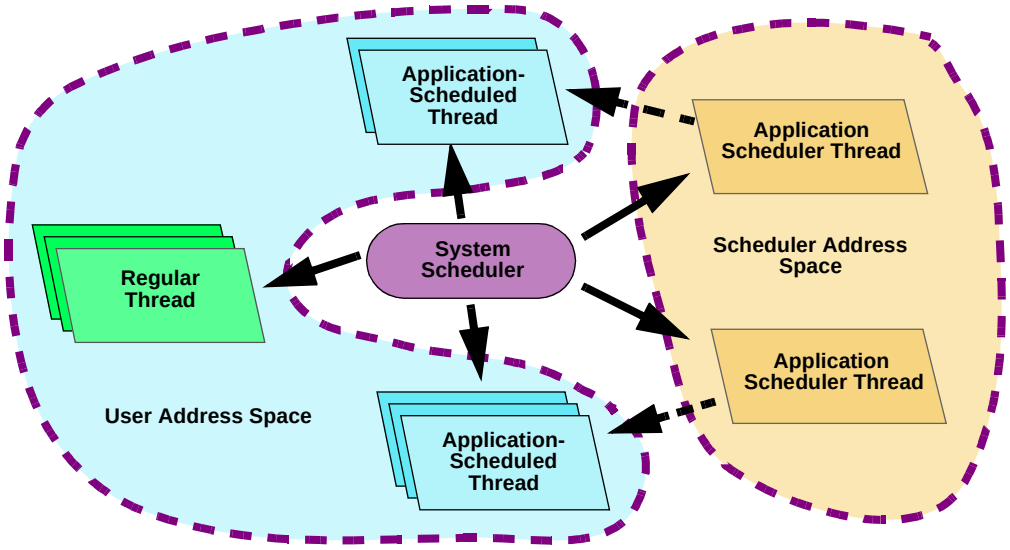

Figure 1. Model for application-defined scheduling

- *Application scheduler threads*: special threads used to run application schedulers.
- *Regular threads*: regular application threads

The application schedulers can run in the context of the kernel or in the context of the application. This allows implementations in which application threads are not trusted, and

therefore their schedulers run in the context of the application, as well as implementations for trusted application schedulers, which can run more efficiently inside the kernel. Because of this duality, the scheduler threads are modeled as if they run in a separate context, which is called the scheduler space. The main implication of this separate space is that for portability purposes the application schedulers cannot directly share information with the kernel, nor with regular threads. Application schedulers belonging to the same process can share data among them. This is useful for building a multithreaded application scheduler, for a multiprocessor platform.

According to the way a thread is scheduled, we can categorize the threads as:

- *System-scheduled threads*: these threads are scheduled directly by the operating system, without intervention of a scheduler thread.
- *Application-scheduled threads*: these threads are also scheduled by the operating system, but before they can be scheduled they need to be activated by their application-defined scheduler.

It is unspecified whether application scheduler threads can themselves be application scheduled. They can always be system scheduled.

There are two ways in which a thread can be scheduled by an application scheduler:

- At thread creation. In this case, thread creation attributes specify the scheduler to be used, the regular and application scheduling parameters, and the scheduling policy (as application scheduled). The *pthread_create*() function creates the thread in a suspended state and waits until the scheduler either accept or rejects the new thread. If rejected, *pthread_create* fails and the thread is destroyed. To avoid this creation and destruction process, the parent thread could explicitly invoke the scheduler to reserve resources for the new thread, before creating it.
- By dynamically changing the scheduling policy to “application scheduled”. Before the change the thread must be under a policy that is not application-scheduled, and must set the scheduler thread and the application scheduling parameters attributes to the desired values. Then, the *pthread_setschedparam*() function is invoked to dynamically change the scheduling policy into application-scheduled. If the thread is rejected, the function fails.

Direct change of the application scheduler is not allowed because if the new scheduler rejects the thread after the old scheduler has detached it, the thread would be left in an uncertain state, with no scheduler. Therefore, to change the scheduler a thread would first have to switch to another policy, SCHED_FIFO for example, and from there request attachment to the new scheduler. If rejected, the thread would continue with the SCHED_FIFO policy in that case.

Because the use of mutexes may cause priority inversions or similar delay effects, it is necessary that the scheduler thread knows about their use, to establish its own protocols adapted to the particular thread scheduling policy. As is shown in Figure 2, two kinds of mutexes will be considered:

- *System-scheduled mutexes*. Those created with the current POSIX protocols: no priority inheritance (PTHREAD_PRIO_NONE), immediate priority ceiling (PTHREAD_PRIO_PROTECT), or basic priority inheritance (PTHREAD_PRIO_INHERIT). They can be used to access resources shared between application schedulers, or between sets of application-scheduled threads attached to different schedulers.
- *Application-scheduled mutexes*: Those created with PTHREAD_APPSCHED_PROTOCOL. The behavior of the protocol itself is defined by the application scheduler. The kernel notifies the scheduler about the request to lock one such mutex, the execution of an unlock operation, or when a thread blocks on one of these mutexes. After the lock

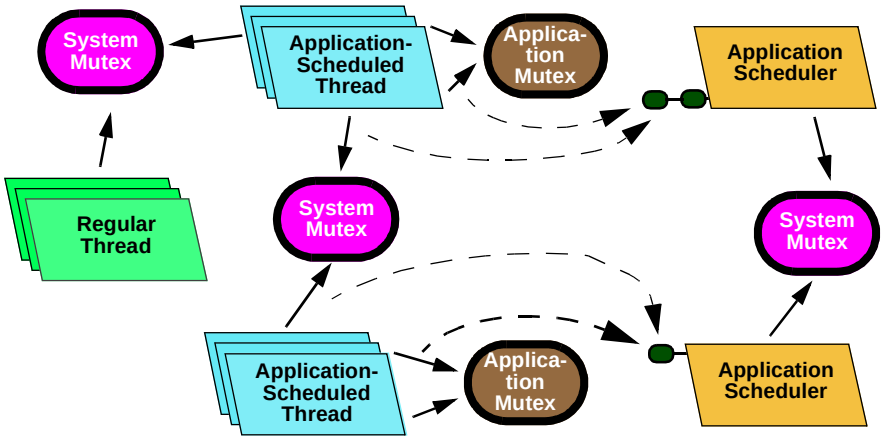

Figure 2. . Model for Application-Defined Synchronization

request operation the application scheduler can chose to grant or not the mutex to the requesting thread. The block event might not be necessary in some schedulers that implement non-blocking synchronization protocols.

A.1.1. Relations with Other Threads

Each thread in the system, whether application- or system-scheduled, has a system priority:

- For system-scheduled threads, the system priority is the priority defined in its scheduling parameters (*sched_priority* field of its *sched_param* structure), possibly modified by the inheritance of other priorities through the use of mutexes.
- For application-scheduled threads, the system priority is lower than or equal to the system priority of their scheduler thread. The system priority of an application-scheduled thread may change because of the inheritance of other system priorities through the use of mutexes. In that case, its scheduler also inherits the same system priority (but this priority is not inherited by the rest of the threads scheduled by that scheduler). In addition to the system priority, application-scheduled threads have *application scheduling parameters* that are used to schedule that thread contending with the other threads attached to the same application scheduler. The system priority always takes precedence over any application scheduling parameters. Therefore, application-scheduled threads and their scheduler take precedence over threads with lower system priority, and they are always preempted by threads with higher system priority that become ready. The scheduler always takes precedence over its scheduled threads.

If application-scheduled threads coexist at the same priority level with other system-scheduled threads, then POSIX scheduling rules apply as if the application-scheduled threads were scheduled under the FIFO within priorities policy (SCHED_FIFO); so a thread runs until completion, until blocked, or until preempted, whatever happens earlier. A thread running under the round-robin within priorities policy (SCHED_RR) runs until completion, until blocked, until preempted, or until its round robin quantum has been consumed, whatever happens earlier. Of course, in that case the interactions between the different policies may be difficult to analyze, and thus the normal use will be to have the scheduler thread and its scheduled threads running at an exclusive range of system priorities.

In the presence of priority inheritance, the scheduler inherits the same priorities as its scheduled tasks, to prevent priority inversions from occurring. This means that high priority tasks that share mutexes with lower system priority application threads must take into account the scheduler overhead when accounting for their blocking times.

A.1.2. Relations Between the Scheduler and its Attached Threads

When an application-defined thread is attached to its application scheduler, the latter has to either accept it or reject it, based upon the current state and the scheduling attributes of the candidate thread. Rejection of a thread causes the thread creation function to return an error.

Each application-defined scheduler may activate many application-scheduled threads to run concurrently. The scheduler may also block previously activated threads. Among themselves, concurrently scheduled threads are activated like SCHED_FIFO threads. As mentioned previously, the scheduler always takes precedence over its scheduled threads.

For an application-scheduled thread to become ready it is necessary that its scheduler activates it. When the application thread executes one of the following actions or experiences one of the following situations, a scheduling event is generated for the scheduler, unless the scheduling event to be generated is being filtered out (discarded).

- when a thread requests attachment to the scheduler
- when a thread terminates or requests de-attachment from the scheduler
- when a thread blocks (except at an application-scheduled mutex)
- when a thread is unblocked by the system and would become ready
- when a thread changes its scheduling parameters
- when a thread invokes the *pthread_yield()* operation
- when a thread explicitly invokes the scheduler
- when a thread inherits or uninherits a priority, due to the use of a system mutex
- when a thread does any operation on a application-scheduled mutex.

The application scheduler is a special thread whose code is usually a loop where it waits for a scheduling event to be notified to it by the system, and then determines the next application threads to be activated. The scheduling events are stored in a FIFO queue until processed by the scheduler. For most scheduling events, unless the event is masked, after the event is generated the associated thread is suspended, to allow the scheduler to make a decision; for the thread to become active again it has to be explicitly activated by the scheduler, via the “execute actions” operation. For other events, the thread continues in the same state as before. The specific events that may be notified, and the state of the associated thread after the event are shown in Table 1.

Table 1. Application scheduling events and the state of the associated thread

Application Scheduling Events	State of associated thread after the event
POSIX_APPSCHED_NEW	Suspended
POSIX_APPSCHED_TERMINATE	No change
POSIX_APPSCHED_READY	Suspended
POSIX_APPSCHED_BLOCK	No Change
POSIX_APPSCHED_YIELD	Suspended

Table 1. Application scheduling events and the state of the associated thread (Continued)

POSIX_APPSCHED_SIGNAL	Not applicable (no associated thread)
POSIX_APPSCHED_CHANGE_SCHED_PARAM	No change
POSIX_APPSCHED_EXPLICIT_CALL	Suspended
POSIX_APPSCHED_EXPLICIT_CALL_WITH_DATA	Suspended
POSIX_APPSCHED_TIMEOUT	Not applicable (no associated thread)
POSIX_APPSCHED_PRIORITY_INHERIT	Suspended
POSIX_APPSCHED_PRIORITY_UNINHERIT	Suspended
POSIX_APPSCHED_INIT_MUTEX	Suspended
POSIX_APPSCHED_DESTROY_MUTEX	Suspended
POSIX_APPSCHED_LOCK_MUTEX	Suspended
POSIX_APPSCHED_TRY_LOCK_MUTEX	Suspended
POSIX_APPSCHED_UNLOCK_MUTEX	Suspended
POSIX_APPSCHED_BLOCK_AT_MUTEX	Suspended
POSIX_APPSCHED_CHANGE_MUTEX_SCHED_PARAM	Suspended

The description of the different events appears next:

- POSIX_APPSCHED_NEW. A thread has requested attachment to this scheduler; this can be a newly created thread (via *pthread_create*()), or an existing thread that was not running under this scheduler (via *pthread_setschedparam*()).
- POSIX_APPSCHED_TERMINATE. A thread attached to this scheduler has been terminated (via an explicit or implicit *pthread_exit*(), or by cancellation via *pthread_cancel*()), or it has changed its scheduling parameters and should no longer run under this scheduler (via *pthread_setschedparam*()). The thread is not suspended, because it is no longer going to run under the present scheduler.
- POSIX_APPSCHED_READY. A thread attached to this scheduler that was blocked has become unblocked by the system.
- POSIX_APPSCHED_BLOCK. A thread attached to this scheduler has blocked (except at an application-scheduled mutex). The thread is not suspended because it is already blocked by the system. Once it is unblocked, a POSIX_APPSCHED_READY event will be generated.
- POSIX_APPSCHED_YIELD. A thread attached to this scheduler has invoked the *sched_yield*() operation.
- POSIX_APPSCHED_SIGNAL. A blocked signal belonging to the set of signals for which the scheduler is waiting has been accepted by the scheduler thread.
- POSIX_APPSCHED_CHANGE_SCHED_PARAM. The scheduling parameters of a thread attached to this scheduler have been changed, but the thread continues to run under this scheduler. The change includes either the regular scheduling parameters (`sched_policy` and `schedparam` attributes, via *pthread_setschedparam*()) or the application-defined scheduling parameters. (`appsched_param`, via *pthread_setappschedparam*()). Because this operation may be invoked asynchronously by some thread different than the one changing its parameters, there is no change to the activated/suspended state of the involved threads.

- POSIX_APPSCHED_EXPLICIT_CALL. A thread attached to this scheduler has explicitly invoked the scheduler via *posix_appsched_invoke_scheduler*().
- POSIX_APPSCHED_EXPLICIT_CALL_WITH_DATA. A thread attached to this scheduler has explicitly invoked the scheduler, with a message containing scheduling information, and possibly requesting a reply message, via *posix_appsched_invoke_withdata*().
- POSIX_APPSCHED_TIMEOUT. A timeout requested by the scheduler has expired.
- POSIX_APPSCHED_PRIORITY_INHERIT. A thread attached to this scheduler has inherited a new system priority due to the use of system mutexes.
- POSIX_APPSCHED_PRIORITY_UNINHERIT. A thread attached to this scheduler has finished the inheritance of a system priority that was inherited due to the use of system mutexes.
- POSIX_APPSCHED_INIT_MUTEX. A thread attached to this scheduler has requested initialization of an application-scheduled mutex.
- POSIX_APPSCHED_DESTROY_MUTEX. A thread attached to this scheduler has destroyed an application-scheduled mutex.
- POSIX_APPSCHED_LOCK_MUTEX. A thread attached to this scheduler has invoked the “lock” operation on an available application- scheduled mutex.
- POSIX_APPSCHED_TRYLOCK_MUTEX. A thread attached to this scheduler has invoked the “try lock” operation on an available application- scheduled mutex.
- POSIX_APPSCHED_UNLOCK_MUTEX. A thread attached to this scheduler has released the lock of an application-scheduled mutex.
- POSIX_APPSCHED_BLOCK_AT_MUTEX. A thread attached to this scheduler has blocked at an unavailable application-scheduled mutex.
- POSIX_APPSCHED_CHANGE_MUTEX_SCHED_PARAM. A thread attached to this scheduler and currently holding the lock on an application-scheduled mutex has changed the scheduling parameters of that mutex.

The application scheduler is a special thread whose code is usually a loop where it waits for a scheduling event to be notified to it by the system, and then determines the next application thread or threads to be activated.

Although the scheduler can activate many threads at once, it is a single thread and therefore its actions are all sequential. For multiprocessor systems this may seem to be a limitation, but for these systems several schedulers could be running simultaneously on different processors, cooperating with each other by synchronizing through regular mutexes and condition variables. For single processor systems the sequential nature of the scheduler should be no problem.

A.1.3. Sharing Information Between the Schedulers and Their Scheduled Threads

There is an explicit “invoke scheduler” family of operations that can be used by an application-scheduled thread to directly invoke its scheduler, pass information to it, and obtain information back. Generally, the information to be shared by the scheduler and its associated threads is small, and therefore this mechanism does not introduce much overhead.

Because the scheduler may run in a context different than its scheduled threads, possibly with a different address space, there is no mechanism to directly share memory among them. The mechanism used in POSIX to share memory between processes that are placed in different address spaces, shared memory objects is not useful for this case because it is designed for different processes, and in this case the scheduler and its threads are in the same process.

A.2. Interface for Application-Defined Scheduling

All the interfaces defined in this standard are mandatory if the standard is supported. The implementation shall conform to at least one of the POSIX.13 realtime profiles.

A.2.1. Data Definitions

A.2.1.1. Errors

The following symbolic name representing an error number shall be defined in <errno.h>:

[EREJECT] The thread requesting attachment to an application-defined scheduler has been rejected by that scheduler.

[EPOLICY] The scheduling policy or the scheduler state attribute of the calling thread is not valid for this operation.

[EMASKED] The operation cannot be executed because the associated scheduling event is currently masked by the application scheduler.

A.2.1.2. Minimum and Configurable Values

The constants specified in the Table 1-1 shall be defined in <limits.h> with the values shown. These are symbolic names for the most restrictive value for certain features in this Standard. A conforming implementation shall provide values at least this large. A portable application shall not require a larger value for correct operation:

Table 1-1: Minimum Values

Constant	Description	Value
_POSIX_APPSCHEDPARAM_MAX	The minimum size in bytes for the bound on the size of the application scheduling parameters (see A.2.1.3)	32
_POSIX_APPSCHEDINFO_MAX	The minimum size in bytes for the bound on the size of the data exchanged between an application-scheduled thread and its scheduler (see A.2.10)	32
_POSIX_APPMUTEXPARAM_MAX	The minimum size in bytes for the bound on the size of the mutex scheduling parameters (see A.5.1.2)	16

The constants defined in Table 1-2 are related to the constants that appear in Table 1-1. If the actual value is determinate, it shall be represented by the constant, defined in <limits.h>. If the actual bound is indeterminate, it shall be provided by the *sysconf*() function using the corresponding name value, defined in <unistd.h> and specified in Table 1-3.

Table 1-2: Run-time Invariant Values (possibly Indeterminate)

Constant	Description	Minimum Value
POSIX_APPSCHEDPARAM_MAX	The bound in bytes for the size of the application scheduling parameters (see A.2.1.3)	_POSIX_APPSCHEDPARAM_MAX
POSIX_APPSCHEDINFO_MAX	The bound in bytes for the size of the data exchanged between an application-scheduled thread and its scheduler (see A.2.10)	_POSIX_APPSCHEDINFO_MAX
POSIX_APPMUTEXPARAM_MAX	The bound in bytes for the size of the mutex scheduling parameters (see A.5.1.2)	_POSIX_APPMUTEXPARAM_MAX

Table 1-3: Configurable System Variables

Variable	Description	name Value
POSIX_APPSCHEDPARAM_MAX	The bound in bytes for the size of the application scheduling parameters (see A.2.1.3)	_SC_POSIX_APPSCHEDPARAM_MAX
POSIX_APPSCHEDINFO_MAX	The bound in bytes for the size of the data exchanged between an application-scheduled thread and its scheduler (see A.2.10)	_SC_POSIX_APPSCHEDINFO_MAX
POSIX_APPMUTEXPARAM_MAX	The bound in bytes for the size of the mutex scheduling parameters (see A.5.1.2)	_SC_POSIX_APPMUTEXPARAM_MAX

A.2.1.3. Scheduling Policy and Attributes

The following scheduling policy shall be defined in <sched.h>. This policy shall not be used to schedule processes, i.e., it shall not be used in a call to *sched_setscheduler*():

Symbol	Description
SCHED_APP	Application-defined scheduling policy

The following new thread attributes are defined for specifying application scheduling parameters:

Attribute type	Attribute Name	Description
pthread_t	appscheduler	Scheduler thread to which the application-defined scheduler is attached
n/a	appsched_param	Application-defined scheduling parameters

For a thread to be created as a application-scheduled thread under a specified scheduler, the scheduler must have been created beforehand. Then, at the thread creation time, the thread creation policy is set to the value SCHED_APP, and the scheduling parameters are set with the appropriate values in the appscheduler and appsched_param attributes. These attributes can also be set or queried dynamically, after the thread has been created.

If the scheduling policy of the thread is not SCHED_APP, these attributes have no effect.

The default value for the appscheduler attribute is unspecified. If at the time of the thread creation the scheduling policy is SCHED_APP and the appscheduler attribute does not refer to a valid application scheduler thread, the corresponding *pthread_create*() operation shall fail with an error of [EINVAL].

The appsched_param attribute is a variable-size buffer containing application-defined scheduling parameters. The maximum size of this attribute shall be represented by the variable POSIX_APPSCHEDPARAM_MAX (see A.2.1.2). The default value for the appsched_param attribute shall be a buffer of zero bytes.

A.2.1.4. Scheduler Thread State Attribute

A new attribute, appscheduler_state, is defined to represent whether a thread is an application-scheduler thread or a regular thread. This is a thread-creation attribute that can be queried dynamically, but cannot be modified dynamically.

A value of PTHREAD_APPSCHEDULER for the appscheduler_state attribute shall cause threads created with that attribute to be created as application scheduler threads. Application scheduler threads shall not share variables or memory with other regular application threads, and shall not synchronize with such threads using mutexes or condition variables; otherwise the results are undefined. Application scheduler threads may share variables and cooperate via mutexes and condition variables with other application scheduler threads in the same process.

A value of PTHREAD_REGULAR for the appscheduler_state attribute shall cause the thread to be created as a normal application thread as specified elsewhere in this standard.

These symbols shall be defined in <pthread.h>. The default value shall be PTHREAD_REGULAR.

A.2.1.5. Scheduling Events

The following datatypes shall be defined in <sched.h>:

```
union posix_appsched_eventinfo {
      int sched_priority;
      siginfo_t siginfo;
      pthread_mutex_t *mutex;
      void *info;
      int user_event_code;
};

struct posix_appsched_event {
      int event_code;
      pthread_t thread;
      union posix_appsched_eventinfo event_info;
      size_t info_size;
};
```

The structure *posix_appsched_event* represents a scheduling event that the system notifies to the application-defined scheduler. It contains the code of the specific event that has occurred (see Table 2) in the *event_code* member; the thread identifier of the thread that caused the event in the *thread* member; the information associated with that event in the *event_info* member; and the size of the information, if necessary, in the *info_size* member.

The union *posix_appsched_eventinfo* represents the different kinds of information that may be attached to a scheduling event. Table 2 shows the information associated with each scheduling event. All the symbols shall be defined in <sched.h>.

Table 2. Application scheduling events and their associated information

Application Scheduling Events (event_code**)**	**Additional Information (**event_info**)**
POSIX_APPSCHED_NEW	NULL pointer
POSIX_APPSCHED_TERMINATE	NULL pointer
POSIX_APPSCHED_READY	NULL pointer
POSIX_APPSCHED_BLOCK	NULL pointer
POSIX_APPSCHED_YIELD	NULL pointer
POSIX_APPSCHED_SIGNAL	*siginfo_t* value delivered with the signal
POSIX_APPSCHED_CHANGE_SCHED_PARAM	NULL pointer
POSIX_APPSCHED_EXPLICIT_CALL	User event code
POSIX_APPSCHED_EXPLICIT_CALL_WITH_DATA	Pointer to the message set by the thread[a]
POSIX_APPSCHED_TIMEOUT	NULL pointer
POSIX_APPSCHED_PRIORITY_INHERIT	Inherited system priority

Table 2. Application scheduling events and their associated information (Continued)

POSIX_APPSCHED_PRIORITY_UNINHERIT	Uninherited system priority
POSIX_APPSCHED_INIT_MUTEX	Pointer to the app. scheduled mutex
POSIX_APPSCHED_DESTROY_MUTEX	Pointer to the app. scheduled mutex
POSIX_APPSCHED_LOCK_MUTEX	Pointer to the app. scheduled mutex
POSIX_APPSCHED_TRY_LOCK_MUTEX	Pointer to the app. scheduled mutex
POSIX_APPSCHED_UNLOCK_MUTEX	Pointer to the app. scheduled mutex
POSIX_APPSCHED_BLOCK_AT_MUTEX	Pointer to the app. scheduled mutex
POSIX_APPSCHED_CHANGE_MUTEX_SCHED_PARAM	Pointer to the app. scheduled mutex

[a.] In this case, *info_size* shall be the size of the information; in other cases, *info_size* shall be zero

A.2.1.6. Scheduling Events Sets

The *posix_appsched_eventset_t* type shall be defined in <sched.h>. Values of this type represent sets of scheduling event codes. Several functions used to manipulate objects of this type are defined in A.2.8. No comparison or assignment operators are defined for the type *posix_appsched_eventset_t*.

A.2.1.7. Scheduling Actions

The *posix_appsched_actions_t* datatype shall be defined in <sched.h> to represent a list of scheduling actions that the scheduler will later request to be executed by the system. The possible actions are of the following kinds:

- accept or reject a thread that has requested attachment to this scheduler
- activate or suspend an application-scheduled thread
- accept or reject initialization of an application-scheduled mutex
- grant the lock of an application-scheduled mutex

No comparison or assignment operators are defined for the type *posix_appsched_actions_t*.

A.2.1.8. Rationale for Data Definitions

The new thread attributes for application scheduling, appscheduler and appsched_param, were initially proposed as additional fields of the *sched_param* structure that is commonly used to store the scheduling parameters of a thread. However, this solution has the disadvantage that code requiring use of that structure needs recompilation. In particular, because the widely used C library (libc) uses that structure it was considered that, to ease backwards compatibility, new thread attributes should be used instead.

The minimum size for the scheduling parameters attribute appsched_param is set to 32 because there are many scheduling algorithms that require specifying up to four time parameters (for example, period, execution time, soft deadline, and hard deadline). Each of these parameters requires eight bytes if stored in a *struct timespec* datatype.

An event for notifying preemption of a scheduled thread is not included. Although such event might seem to be useful to measure execution times from an application scheduler, it would be difficult for the scheduler to know when a scheduled thread actually started executing. If measuring execution time is required, it is much simpler to use a POSIX execution time clock for that purpose.

A.2.2. Interface for the Creation of the Scheduler

A.2.2.1. Synopsis

```
#include <pthread.h>

int pthread_attr_setappschedulerstate
        (pthread_attr_t *attr,
         int appschedstate);

int pthread_attr_getappschedulerstate
        (const pthread_attr_t *attr,
         int *appschedstate);
```

A.2.2.2. Description

The *pthread_attr_setschedulerstate*() and *pthread_attr_getschedulerstate*() functions are used to set and get the `appscheduler_state` attribute in the object pointed to by *attr*. The *pthread_attr_setschedulerstate*() function shall set this attribute to the value specified by *appschedstate*, which shall be either PTHREAD_REGULAR or PTHREAD_APPSCHEDULER. These symbols are described in A.2.1.4.

An application-defined scheduler itself shall be scheduled by the system as a thread under the SCHED_FIFO policy, with the system priority determined by the *sched_priority* member of its `schedparam` attribute.

A.2.2.3. Returns

Upon successful completion, *pthread_attr_setschedulerstate*() and *pthread_attr_getschedulerstate*() shall return a value of 0. Otherwise an error number shall be returned to indicate the error.

The *pthread_attr_getschedulerstate*() function stores the `appscheduler_state` attribute in the variable pointed to by *appschedstate* if successful.

A.2.2.4. Errors

If any of the following conditions occur, the *pthread_attr_setschedulerstate*() function shall return the corresponding error number:

[EINVAL] The value of *appschedstate* was not valid.

A.2.2.5. Cross-References

pthread_create(), *pthread_getappscheduler*().

A.2.3. Dynamically Getting the `Appscheduler_State` Attribute

A.2.3.1. Synopsis

```
#include <pthread.h>

int pthread_getappschedulerstate
       (pthread_t thread,
         int *appschedstate);
```

A.2.3.2. Description

This function allows the `appscheduler_state` attribute to be retrieved dynamically, after the thread has been created. The value of the attribute shall be returned in the variable pointed to by *appschedstate*.

A.2.3.3. Returns

The *pthread_getappschedulerstate*() function stores the `appscheduler_state` attribute of the thread specified by *thread* in the variable pointed to by *appschedstate* and returns zero, if successful.

A.2.3.4. Errors

If any of the following conditions are detected, the *pthread_getappschedulerstate*() function shall return the corresponding error number:

[ESRCH] The value specified by *thread* does not refer to an existing thread.

A.2.3.5. Cross-References

pthread_attr_setschedulerstate().

A.2.4. Interfaces for Creating Application-Scheduled Threads

A.2.4.1. Synopsis

```
#include <pthread.h>

int pthread_attr_setappscheduler
        (pthread_attr_t *attr,
         pthread_t scheduler);

int pthread_attr_getappscheduler
        (const pthread_attr_t *attr,
         pthread_t *scheduler);
```

```
int pthread_attr_setappschedparam
        (pthread_attr_t *attr,
         const void *param,
         size_t paramsize);
int pthread_attr_getappschedparam
        (const pthread_attr_t *attr,
         void *param,
         size_t *paramsize);
```

A.2.4.2. Description

The *pthread_attr_setappscheduler*() and *pthread_attr_getappscheduler*() functions are used to set and get the appscheduler attribute in the object pointed to by *attr*. For those threads with the scheduling policy SCHED_APP, this attribute represents the identifier of its scheduler thread. The attribute is described in A.2.1.3. If successful, the *pthread_attr_setappscheduler*() function shall set this attribute to the value specified by *scheduler*, which shall be a valid application scheduler thread.

The *pthread_attr_setappschedparam*() and *pthread_attr_getappschedparam*() functions are used to set and get the appsched_param attribute in the object pointed to by *attr*. For those threads with the scheduling policy SCHED_APP, this attribute represents the application-specific scheduling parameters. The attribute is described in A.2.1.3. If successful, the *pthread_attr_setappschedparam*() function shall set the size of the appsched_param attribute to the value specified by *paramsize*, and shall copy the scheduling parameters occupying *paramsize* bytes and pointed to by *param* into that attribute.

The *pthread_attr_getappschedparam*() function shall copy the contents of the appsched_param attribute into the memory area pointed to by *param*. This memory area shall be capable of storing at least a number of bytes equal to the size of the appsched_param attribute; otherwise, the results are undefined.

A.2.4.3. Returns

Upon successful completion, *pthread_attr_setappscheduler*(), *pthread_attr_setappschedparam*(), *pthread_attr_getappscheduler*(), and *pthread_attr_getappschedparam*() shall return a value of 0. Otherwise an error number shall be returned to indicate the error.

The *pthread_attr_getappscheduler*() function stores the appscheduler attribute in the variable pointed to by *scheduler* if successful.

The *pthread_attr_getappschedparam*() function stores the size of the appsched_param attribute in the variable pointed to by *paramsize*, and copies the contents of the appsched_param attribute into the memory area pointed to by *param*.

A.2.4.4. Errors

If any of the following conditions occur, the *pthread_attr_setappscheduler*() function shall return the corresponding error number:

[EINVAL] The value of *scheduler* was not valid.

If any of the following conditions occur, the *pthread_attr_setappschedparam*() function shall return the corresponding error number:

[EINVAL] The value of *paramsize* was smaller than zero, or was larger than POSIX_APPSCHEDPARAM_MAX.

If any of the following conditions are detected, the *pthread_attr_setappschedparam*() function shall return the corresponding error number:

[EINVAL] The value of *param* was invalid.

A.2.4.5. Cross-References

pthread_create().

A.2.5. Interfaces for Dynamic Access to Application Scheduling Parameters

A.2.5.1. Synopsis

```
#include <pthread.h>

int pthread_setappscheduler
        (pthread_t thread,
         pthread_t scheduler);

int pthread_setappschedparam
        (pthread_t thread,
         const void *param,
         size_t paramsize);

int pthread_getappscheduler
        (pthread_t thread,
         pthread_t *scheduler);

int pthread_getappschedparam
        (pthread_t thread,
         void *param,
         size_t *paramsize);
```

A.2.5.2. Description

The *pthread_setappscheduler*() and *pthread_getappscheduler*() functions are used to dynamically set and get the `appscheduler` attribute of the thread identified by *thread*. For a thread with the scheduling policy SCHED_APP, this attribute represents the identifier of its scheduler thread. This attribute is described in A.2.1.3.

If successful, the *pthread_setappscheduler*() function shall set the `appscheduler` attribute to the value specified by *scheduler*. For this function to succeed, the scheduling policy of the calling thread shall not be SCDHED_APP, because directly changing the scheduler is not permitted.

The *pthread_setappschedparam*() and *pthread_getappschedparam*() functions are used to dynamically set and get the `appsched_param` attribute of the thread identified by *thread*.

For a thread with the scheduling policy SCHED_APP, this attribute represents its application-defined scheduling parameters. This attribute is described in A.2.1.3.

If successful, the *pthread_setappschedparam*() function shall set the size of the appsched_param attribute to the value specified by *paramsize*, and it shall copy the scheduling parameters occupying *paramsize* bytes and pointed to by *param* into that attribute. In addition, if the scheduling policy of *thread* is SCHED_APP, it shall generate a POSIX_APPSCHED_CHANGE_SCHED_PARAM event for the application scheduler of that thread, unless that event is masked in that application scheduler.

The memory area represented by *param* in the call to *pthread_getappschedparam*() shall be capable of storing at least a number of bytes equal to the size of the appsched_param attribute; otherwise, the results are undefined.

A.2.5.3. Returns

Upon successful completion, *pthread_setappscheduler*(), *pthread_getappscheduler*(), *pthread_setappschedparam*(), and *pthread_getappschedparam*() shall return a value of 0. Otherwise an error number shall be returned to indicate the error.

The *pthread_getappscheduler*() function stores the appscheduler attribute in the variable pointed to by *appsched*, if successful.

The *pthread_getappschedparam*() function stores the size of the appsched_param attribute in the variable pointed to by paramsize, and copies the contents of the appsched_param attribute into the memory area pointed to by *param*, if successful.

A.2.5.4. Errors

If any of the following conditions occur, the *pthread_setappscheduler*() function shall return the corresponding error number:

[EINVAL] The value of *scheduler* was not valid.

[EPOLICY] The scheduling policy of the calling thread is SCHED_APP

If any of the following conditions occur, the *pthread_setappschedparam*() function shall return the corresponding error number:

[EINVAL] The value of *paramsize* was less than zero, or was larger than POSIX_APPSCHEDPARAM_MAX.

If any of the following conditions are detected, the *pthread_setappschedparam*() function shall return the corresponding error number:

[EINVAL] The value of *param* was invalid.

If any of the following conditions are detected, the *pthread_setappscheduler*(), *pthread_getappscheduler*(), *pthread_setappschedparam*(), and *pthread_getappschedparam*() functions shall return the corresponding error number:

[ESRCH] The value specified by *thread* does not refer to an existing thread.

A.2.5.5. Cross-References

pthread_create(), *pthread_attr_setappscheduler*, *pthread_attr_setappschedparam*().

A.2.6. Interfaces for the Scheduler Thread: Scheduling Actions

A.2.6.1. Synopsis

```
#include <sched.h>

int posix_appsched_actions_init(
        posix_appsched_actions_t *sched_actions);

int posix_appsched_actions_destroy(
        posix_appsched_actions_t *sched_actions);

int posix_appsched_actions_addaccept(
        posix_appsched_actions_t *sched_actions,
        pthread_t thread);

int posix_appsched_actions_addreject(
        posix_appsched_actions_t *sched_actions,
        pthread_t thread);

int posix_appsched_actions_addactivate(
        posix_appsched_actions_t *sched_actions,
        pthread_t thread);

int posix_appsched_actions_addsuspend(
        posix_appsched_actions_t *sched_actions,
        pthread_t thread);

int posix_appsched_actions_addacceptmutex(
        posix_appsched_actions_t *sched_actions,
        const pthread_mutex_t *mutex);

int posix_appsched_actions_addrejectmutex(
        posix_appsched_actions_t *sched_actions,
        const pthread_mutex_t *mutex);

int posix_appsched_actions_addlockmutex(
        posix_appsched_actions_t *sched_actions,
        pthread_t thread,
        const pthread_mutex_t *mutex);
```

A.2.6.2. Description

A scheduling actions object is of type *posix_appsched_actions_t* (defined in <sched.h>) and is used to specify a series of actions to be performed by the *posix_appsched_execute_actions*() function. The order of the actions added to the object shall be preserved.

The *posix_appsched_actions_init*() function initializes the object referenced by *sched_actions* to contain no scheduling actions to perform. After successful initialization, the number of actions that may be successfully added to the actions object shall be at least equal to _POSIX_THREAD_THREADS_MAX.

The effect of initializing an already initialized actions object is undefined.

The *posix_appsched_actions_destroy*() function destroys the object referenced by *sched_actions*; the object becomes, in effect, uninitialized. An implementation may cause *posix_appsched_actions_destroy*() to set the object referenced by *sched_actions* to an in-

valid value. A destroyed scheduling actions object can be reinitialized using *posix_appsched_actions_init*(); the results of otherwise referencing the object after it has been destroyed are undefined.

The *posix_appsched_actions_addaccept*() function adds a thread-accept action to the object referenced by *sched_actions,* that will serve to notify that the thread identified by *thread* has been accepted by the scheduler thread to be scheduled by it. When the *posix_appsched_execute_actions*() function is invoked with this scheduling actions object, any thread waiting for such notification either on a *pthread_create*() or a *pthread_setschedparam*() function shall successfully complete the function. If no thread was waiting for such notification, the action has no effects.

The *posix_appsched_actions_addreject*() function adds a thread-reject action to the object referenced by *sched_actions,* that will serve to notify that the thread identified by *thread* has not been accepted by the scheduler thread to be scheduled by it, possibly because the thread contained invalid application scheduling attributes, or because there are not enough resources for the new thread. When the *posix_appsched_execute_actions*() function is invoked with this scheduling actions object, any thread waiting for such notification either on a *pthread_create*() or a *pthread_setschedparam*() function shall complete the function with an error code of [EREJECT]. If no thread was waiting for such notification, the action has no effects.

The *posix_appsched_actions_addactivate*() function adds a thread-activate action to the object referenced by *sched_actions,* that will cause the thread identified by *thread* to be activated when the *posix_appsched_execute_actions*() function is invoked with this scheduling actions object. If the thread was already active at the time the thread-activate action is executed, then the thread shall continue to be active. If the thread was suspended at a *posix_mutex_trylock*() operation then this action shall be an indication that the corresponding mutex is not available.

The *posix_appsched_actions_addsuspend*() function adds a thread-suspend action to the object referenced by *sched_actions,* that will cause the thread identified by *thread* to be suspended when the *posix_appsched_execute_actions*() function is invoked with this scheduling actions object. If the thread was already suspended at the time the thread-suspend action is executed, then the thread shall continue to be suspended.

The *posix_appsched_actions_addacceptmutex*() function adds a mutex-accept action to the object referenced by *sched_actions,* that will serve to notify that the mutex identified by *mutex* has been accepted by the scheduler thread to be scheduled by it. When the *posix_appsched_execute_actions*() function is invoked with this scheduling actions object, any thread waiting for such notification on a *pthread_mutex_init*() function shall successfully complete the function. If no thread was waiting for such notification, the action has no effects.

The *posix_appsched_actions_addrejectmutex*() function adds a mutex-reject action to the object referenced by *sched_actions,* that will serve to notify that the mutex identified by *mutex* has not been accepted by the scheduler thread to be scheduled by it, possibly because the mutex contained invalid scheduling attributes, or because there are not enough resources for the new mutex. When the *posix_appsched_execute_actions*() function is invoked with this scheduling actions object, any thread waiting for such notification on a *pthread_mutex_init*() function shall complete the call with an error code of [EREJECT]. If no thread was waiting for such notification, the action has no effects.

The *posix_appsched_actions_addlockmutex*() function adds a mutex-lock action to the object referenced by *sched_actions,* that will cause the lock on the mutex identified by *mutex*

to be granted to the thread indicated by *thread* when the *posix_appsched_execute_actions*() function is invoked with this scheduling actions object.

A.2.6.3. Returns

Upon successful completion, *pthread_appsched_actions_init*(), *pthread_appsched_actions_destroy*(), *pthread_appsched_actions_addaccept*(), *pthread_appsched_actions_addreject*(), *pthread_appsched_actions_addactivate*(), *pthread_appsched_actions_addsuspend*(), *posix_appsched_actions_addacceptmutex*(), *posix_appsched_actions_addrejectmutex*(), and *posix_appsched_actions_addlockmutex*() shall return a value of 0. Otherwise an error number shall be returned to indicate the error.

A.2.6.4. Errors

If any of the following conditions occur, the *pthread_appsched_actions_init*() function shall return the corresponding error number:

[ENOMEM] There is insufficient memory to initialize the actions object.

If any of the following conditions occur, the *pthread_appsched_actions_addaccept*(), *pthread_appsched_actions_addreject*(), *pthread_appsched_actions_addactivate*(), *pthread_appsched_actions_addsuspend*(), *posix_appsched_actions_addacceptmutex*(), *posix_appsched_actions_addrejectmutex*(), and *posix_appsched_actions_addlockmutex*() functions shall return the corresponding error number:

[ENOMEM] There is insufficient memory to add a new action to the actions object.

If any of the following conditions is detected, the *pthread_appsched_actions_destroy*(), *pthread_appsched_actions_addaccept*(), *pthread_appsched_actions_addreject*(), *pthread_appsched_actions_addactivate*(), *pthread_appsched_actions_addsuspend*(), *posix_appsched_actions_addacceptmutex*(), *posix_appsched_actions_addrejectmutex*(), and *posix_appsched_actions_addlockmutex*() functions shall return the corresponding error number:

[EINVAL] The value specified by *sched_actions* is invalid.

A.2.6.5. Cross-References

pthread_create(), *pthread_setschedparam*(), *posix_appsched_execute_actions*(), *pthread_mutex_init*(), *pthread_mutex_lock*(), *pthread_mutex_timedlock*(), *pthread_mutex_trylock*().

A.2.7. Interfaces for the Scheduler Thread: Execute Scheduling Actions

A.2.7.1. Synopsis

```
#include <signal.h>
#include <sched.h>
```

```
int posix_appsched_execute_actions
     (const posix_appsched_actions_t *sched_actions,
      const sigset_t *set,
      const struct timespec *timeout,
      struct timespec *current_time,
      struct posix_appsched_event *event);
```

A.2.7.2. Description

The *posix_appsched_execute_actions*() shall execute the scheduling actions pointed to by *sched_actions* (as described in A.2.1.7 and A.2.6) in the same order as they were added to the actions object, and then it shall cause the application scheduler thread calling the function to wait for the next scheduling event notified by the system. If no event is available in the scheduling event queue of the scheduler thread, then the scheduler thread shall block. If *sched_actions* is **NULL**, then no scheduling actions shall be executed, but the function shall wait for the next scheduling event notified by the system, as in the case in which scheduling actions were executed.

The threads referenced in the actions object pointed to by *sched_actions* shall refer to threads scheduled by the calling application scheduler for the function to succeed. If mutex-lock action is specified in *sched_actions* and the associated thread is not waiting in a *pthread_mutex_lock*() or *pthread_mutex_timedlock*() for the specified mutex at the time of the call, the function shall fail. Detection of one of these error conditions shall be made as the action is performed; thus, if an error is detected, previous actions will have been executed, and none of the further actions shall be executed.

If *set* is not **NULL**, the *posix_appsched_execute_actions*() function shall enable an additional scheduling event which will occur when there is no scheduling event available in its queue and one of the blocked signals belonging to *set* is accepted by the calling thread; the event generated shall have the code POSIX_APPSCHED_SIGNAL, its *thread* member shall be unspecified, and the information associated with the signal shall be returned in the *siginfo* member of the *event_info* member of *event*. If *set* is **NULL**, there shall be no acceptance of signals during the call.

If *timeout* is not **NULL**, the *posix_appsched_execute_actions*() function shall enable an additional scheduling event which will occur when there is no scheduling event available in its queue but the timeout specified by *timeout* expires; the event generated shall have the code POSIX_APPSCHED_TIMEOUT, and its *thread* member shall be unspecified. The value of *timeout* shall be interpreted according to the way in which the timeouts have been configured, as described for the scheduler attributes (see A.2.9). If *timeout* is **NULL**, there shall be no pending timeout for the call.

If *current_time* is not **NULL**, the *posix_appsched_execute_actions*() function shall return in the variable pointed to by that argument the value of the clock specified by the `clockid` attribute of the application scheduler (see A.2.9), as measured immediately before the function returns.[1]

If the thread calling *posix_appsched_execute_actions*() is not an application scheduler thread, then the function shall fail.

1. Notice that the value returned may not represent the time at which it is used by the scheduler, since preemptions may occur in between.

A.2.7.3. Returns

If successful, *posix_appsched_execute_actions*() shall return zero. Otherwise an error number shall be returned to indicate the error.

A.2.7.4. Errors

If any of the following conditions occur, the corresponding error number shall be returned by, *posix_appsched_execute_actions*():

[EINVAL] One or more of the threads specified by the *sched_actions* scheduling actions object does not refer to a thread associated with this scheduler at the time of the *posix_appsched_execute_actions*() call.

A mutex-lock action was specified in *sched_actions* and the associated thread was not waiting for the specified mutex at the time of the call.

[ESRCH] One or more of the threads identifiers specified by the *sched_actions* scheduling actions object did not correspond to a thread that existed at the time of the *posix_appsched_execute_actions*() call.

[EPOLICY] The calling thread is not an application scheduler.

If any of the following conditions is detected, the corresponding error number shall be returned by, *posix_appsched_execute_actions*():

[EINVAL] The value of one or more of the arguments is invalid.

A.2.7.5. Cross-References

posix_appsched_actions_init(), *pthread_create*(), *pthread_setschedparam*(), *pthread_mutex_lock*(), *pthread_mutex_timedlock*(), *pthread_mutex_trylock*().

A.2.8. Interfaces for the Scheduler Thread: Scheduling Events Set Manipulation

A.2.8.1. Synopsis

```
#include <sched.h>

int posix_appsched_emptyset
   (posix_appsched_eventset_t *set);

int posix_appsched_fillset
   (posix_appsched_eventset_t *set);

int posix_appsched_addset
   (posix_appsched_eventset_t *set, int appsched_event);

int posix_appsched_delset
   (posix_appsched_eventset_t *set, int appsched_event);

int posix_appsched_ismember
   (const posix_appsched_eventset_t *set, int appsched_event);
```

A.2.8.2. Description

The *posix_appsched_emptyset*() function shall eliminate all the scheduling event codes from the set pointed to by *set*.

The *posix_appsched_fillset*() function shall add all the scheduling event codes to the set pointed to by *set*.

The *posix_appsched_addset*() function shall add the scheduling event code represented by *appsched_event* to the set pointed to by *set*.

The *posix_appsched_delset*() function shall eliminate the scheduling event code represented by *appsched_event* from the set pointed to by *set*.

The *posix_appsched_ismember*() function shall return one if the scheduling event code represented by *appsched_event* is a member of the set pointed to by *set*. Otherwise, it shall return zero.

A.2.8.3. Returns

Upon successful completion, the *posix_appsched_ismember*() function shall return a value of one if the specified event code is a member of the specified set, or a value of zero if it is not. Upon successful completion, the other functions shall return a value of zero. For all of the above functions, if an error is detected, an error number is returned.

A.2.8.4. Errors

If any of the following conditions is detected, the *posix_appsched_addset*(), *posix_appsched_addset*(), and *posix_appsched_ismember*() functions shall return the following error number:

[EINVAL] The value of *appsched_event* is invalid.

A.2.8.5. Cross-References

posix_appschedattr_seteventmask(), *posix_appschedattr_geteventmask*().

A.2.9. Interfaces for the Scheduler Thread: Scheduler attributes

A.2.9.1. Synopsis

```
#include <sched.h>
int posix_appschedattr_setclock (clockid_t clockid);
int posix_appschedattr_getclock (clockid_t *clockid);
int posix_appschedattr_setflags (int flags);
int posix_appschedattr_getflags (int *flags);
int posix_appschedattr_seteventmask
   (const posix_appsched_eventset_t *set);
```

```
int posix_appschedattr_geteventmask
   (posix_appsched_eventset_t *set);
int posix_appschedattr_setreplyinfo
   (const void *reply, int reply_size);
int posix_appschedattr_getreplyinfo (void *reply, int *reply_size);
```

A.2.9.2. Description

The *posix_appschedattr_setclock*() and *posix_appschedattr_getclock*() functions respectively set and get the `clockid` attribute of the scheduler thread that calls the function. This is the clock that shall be used to report the current time in the *posix_appsched_execute_actions*() function, if requested, and for the timeout of the same function. The default value of the `clockid` attribute shall be CLOCK_REALTIME. If successful, the *posix_appschedattr_setclock*() function shall set the value of the attribute to *clockid*.

The *posix_appschedattr_setflags*() and *posix_appschedattr_getflags*() functions respectively set and get the `flags` attribute of the scheduler thread that calls the function. The default value of the `flags` attribute is with no flags set. If successful, the *posix_appschedattr_setflags*() function shall set the value of the attribute to *flags*. The following flags shall be defined in `<sched.h>`:

POSIX_APPSCHED_ABSTIMEOUT:
: If this flag is set in `flags`, the value of the *timeout* parameter in a call to *posix_appsched_execute_actions*() shall represent an absolute time at which the timeout expires, according to the clock specified by the `clockid` attribute. If the time specified by *timeout* has already passed at the time of the call and there are no scheduling events to be reported, the timeout expires immediately and thus the function does not block. If this flag is not set in `flags`, the *timeout* parameter shall represent a relative time interval, after which the timeout expires. This interval shall be measured using the clock specified by the `clockid` attribute.

The *posix_appschedattr_seteventmask*() and *posix_appschedattr_geteventmask*() functions respectively set and get the `eventmask` attribute of the scheduler thread that calls the function. This is the set of scheduling events that shall be masked, i.e., not reported to this scheduler thread. The default value of the `eventmask` attribute is with an empty set, i.e., all events are reported to the scheduler. If successful, the *posix_appschedattr_seteventmask*() function shall set the value of the attribute to *set*.

The *posix_appschedattr_setreplyinfo*() and *posix_appschedattr_getreplyinfo*() functions respectively set and get the `replyinfo` attribute of the scheduler thread that calls the function. This is a variable-size memory area containing information that shall be used to return information to an application scheduled thread that has requested it via a *posix_appsched_invoke_withdata*() function. The default value of the `replyinfo` attribute shall be a memory area of zero bytes. The maximum size of the `replyinfo` attribute is specified by the POSIX_APPSCHEDINFO_MAX variable (see A.2.1.2). If successful, the *posix_appschedattr_setreplyinfo*() function shall set the value of the attribute equal to the memory area starting at *reply* and of size *reply_size*.

The *posix_appschedattr_getreplyinfo*() functions has undefined results if the memory area pointed to by *reply* is smaller than the size of the `replyinfo` attribute.

If the thread calling any of these functions is not a scheduler thread, then the function shall fail.

A.2.9.3. Returns

Upon successful completion, *posix_appschedattr_setclock*(), *posix_appschedattr_setflags*(), and *posix_appschedattr_seteventmask*() shall return zero. Otherwise, an error number shall be returned to indicate the error.

Upon successful completion, *posix_appschedattr_getclock*(), shall set the value of the `clockid` attribute in the variable pointed to by *clockid* and shall return zero. Otherwise, an error number shall be returned to indicate the error.

Upon successful completion, *posix_appschedattr_getflags*(), shall set the value of the `flags` attribute in the variable pointed to by *flags* and shall return zero. Otherwise, an error number shall be returned to indicate the error.

Upon successful completion, *posix_appschedattr_geteventmask*(), shall set the value of the `eventmask` attribute in the variable pointed to by *set* and shall return zero. Otherwise, an error number shall be returned to indicate the error.

Upon successful completion, *posix_appschedattr_getreplyinfo*(), shall set the value of the variable pointed to by *reply_size* to the size of the `replyinfo` attribute, shall copy the contents of the attribute into the memory area pointed to by *reply*, and shall return zero. Otherwise, an error number shall be returned to indicate the error.

A.2.9.4. Errors

If any of the following conditions is detected, the *posix_appschedattr_setclock*(), *posix_appschedattr_setflags*(), *posix_appschedattr_seteventmask*(), and *posix_appschedattr_setreplyinfo*() functions shall return the following error number:

[EINVAL] The value of one of the arguments is invalid.

If any of the following conditions occurs, the *posix_appschedattr_setclock*(), *posix_appschedattr_setflags*(), *posix_appschedattr_seteventmask*(), *posix_appschedattr_setreplyinfo*(), *posix_appschedattr_getclock*(), *posix_appschedattr_getflags*(), *posix_appschedattr_geteventmask*(), and *posix_appschedattr_getreplyinfo*() functions shall return the following error number:

[EPOLICY] The calling thread is not an application scheduler.

A.2.9.5. Cross-References

posix_appsched_emptyset(), *posix_appsched_fillset*(), *posix_appsched_addset*(), *posix_appsched_delset*(), *posix_appsched_ismember*(), *posix_appsched_execute_actions*(), *posix_appsched_invoke_withdata*().

A.2.10.Interfaces for the Scheduled Thread: Explicit Scheduler Invocation

A.2.10.1.Synopsis

```
#include <sched.h>

int posix_appsched_invoke_scheduler (int user_event_code);

int posix_appsched_invoke_withdata
  (const void *msg, size_t msg_size, void *reply, size_t *reply_size);
```

A.2.10.2.Description

The *posix_appsched_invoke_scheduler*() or *posix_appsched_invoke_withdata*() functions are used by an application-scheduled thread to explicitly invoke its application scheduler.

If successful, the *posix_appsched_invoke_scheduler*() function shall generate a scheduling event with code POSIX_APPSCHED_EXPLICIT_CALL, a *thread* member equal to the thread id of the calling thread, and an *event_info* member with its *user_event_code* member equal to *user_event_code*. This event shall be inserted in the scheduling events queue of the scheduler thread of the calling thread, and then the calling thread shall become suspended. The function call will return after the scheduler thread activates the calling thread via a *posix_appsched_execute_actions*() call.

If successful, the *posix_appsched_invoke_withdata*() function shall generate a scheduling event with code POSIX_APPSCHED_EXPLICIT_CALL_WITH_DATA, a *thread* member equal to the thread Id of the calling thread, and an *info_size* member equal to *msg_size*. In addition, if *msg_size* is larger than zero, the function shall make available to the scheduler thread a memory area whose contents are identical to the memory area pointed to by *msg* and of size *msg_size*, and shall set the *event_info* member of the event with its *info* member pointing to that area of memory[1]. This event shall be inserted in the scheduling events queue of the scheduler thread of the calling thread, and then the calling thread shall become suspended. When the scheduler thread activates the calling thread via a *posix_appsched_execute_actions*() call, if the *reply* argument is non NULL, the `replyinfo` attribute of the scheduler thread is copied into the memory area pointed to by *reply*, and its size is copied into the variable pointed to by *reply_size*. If the size of that memory area is smaller than the size of the `replyinfo` attribute, results are undefined. The `replyinfo` attribute is set by the scheduler thread via a call to *posix_appsched_setreplyinfo*(). Its size is limited to the variable POSIX_APPSCHEDINFO_MAX (see A.2.1.2).

The *posix_appsched_invoke_withdata*() function shall fail if the size specified by *msg_size* is larger than the variable POSIX_APPSCHEDINFO_MAX (see A.2.1.2).

The calling thread shall be an application-scheduled thread for these functions to succeed. The scheduler thread shall have the corresponding scheduling event unmasked, for these functions to succeed.

1. In an implementation in which the scheduler is in the same address space as its threads, copying the information is not needed, and the function may just copy the pointer and size arguments.

A.2.10.3.Returns

Upon successful completion, *posix_appsched_invoke_scheduler*() and *posix_appsched_invoke_withdata*() shall return zero. Otherwise, an error number shall be returned to indicate the error.

A.2.10.4.Errors

If any of the following conditions occurs, the *posix_appsched_invoke_scheduler*() and *posix_appsched_invoke_withdata*() functions shall return the following error number:

[EPOLICY] The calling thread is not an application-scheduled thread.

[EMASKED] The operation cannot be executed because the associated scheduling event is currently masked by the application scheduler.

If any of the following conditions occurs, the *posix_appsched_invoke_withdata*() function shall return the following error number:

[EINVAL] The value of *msg_size* is less than zero or is larger than POSIX_APPSCHEDINFO_MAX.

If any of the following conditions is detected, the *posix_appsched_invoke_withdata*() function shall return the following error number:

[EINVAL] The value of *msg* is invalid.

A.2.10.5.Cross-References

posix_appsched_setreplyinfo(), *posix_appsched_execute_actions*().

A.2.11. Access to Specific Data of Other Threads

A.2.11.1.Synopsis

```
#include <pthread.h>

int pthread_setspecific_for
   (pthread_key_t key, pthread_t thread, const void *value)

int pthread_getspecific_from
   (pthread_key_t key, pthread_t thread, void **value)
```

A.2.11.2.Description

If successful, the *pthread_setspecific_for*() function shall associate a thread-specific *value* with a *key* obtained via a previous call to *pthread_key_create*(), on behalf of the thread specified by *thread*. These values are typically pointers to blocks of dynamically allocated memory that have been reserved for use by the calling thread[1].

1. Because sharing data between a scheduler thread and a regular thread has undefined results, the thread-specific data interface should not be used to exchange or share information between a scheduler thread and its scheduled threads.

If successful, the *pthread_getspecific_from*() function shall store in the variable pointed to by *value* the value currently bound to the specified *key* on behalf of the thread specified by *thread*.

The effect of calling *pthread_setspecific_for*() or *pthread_getspecific_from*() with a *key* value not obtained from *pthread_key_create*() or after *key* has been deleted with *pthread_key_delete*() is undefined.

Both *pthread_setspecific_for*() or *pthread_getspecific_from*() may be called from a thread-specific data destructor function. However, calling *pthread_setspecific_for*() from a destructor may result in lost storage or infinite loops.

A.2.11.3.Returns

If successful, the *pthread_getspecific_from*() function stores in the variable pointed to by *value* the thread-specific data value associated with the given key; if no thread-specific data value is associated with key, then the value **NULL** is returned. In both cases, the function shall return zero. If the function fails , an error number shall be returned to indicate the error

If successful, the *pthread_setspecific_for*() function shall return zero. Otherwise, an error number shall be returned to indicate the error.

A.2.11.4.Errors

If any of the following conditions occur, the *pthread_setspecific_for*() function shall return the corresponding error number:

[ENOMEM] Insufficient memory exists to associate the value with the key.

If any of the following conditions are detected, the *pthread_setspecific_for*() function shall return the corresponding error number:

[EINVAL] The key value is invalid.

If any of the following conditions are detected, the *pthread_setspecific_for*() and *pthread_getspecific_from*() functions shall return the corresponding error number:

[ESRCH] No thread could be found corresponding to that specified by the given thread Id.

A.2.11.5.Cross-References

pthread_key_create(), *pthread_setspecific*(), *pthread_getspecific*(), *pthread_key_delete*().

A.2.11.6.Rationale

Thread-specific data is a very useful mechanism for attaching information to a particular thread, and in particular, to attach scheduling information. However, the information in the thread-specific data currently defined in POSIX.1 is only accessible to the thread to which it is attached. In the context of application-defined scheduling, it is useful for the application scheduler to access the thread-specific data of its scheduled threads. This data is not usually shared by different threads, but just set and used by the application scheduler itself.

It could be argued that an independent data structure could be created using the thread id as an index or with a hash function that would make access efficient. However, the pthread_t type is defined as an opaque type for which there are no comparison operators (just a *pthread_equal*() function for comparison), and therefore any data structure using a pthread_t index and built at the application level is necessarily very inefficient.

For these reasons the thread-specific data interface has been extended with the functions defined in this subclause.

A.3. Modifications to Existing Thread Functions

A.3.1. Thread Creation Scheduling Attributes

The SCHED_APP policy, described in A.2.1.3, shall be supported by *pthread_attr_setschedpolicy*().

A.3.2. Dynamic Thread Scheduling Parameters Access

If *pthread_setschedparam*() is called for a thread whose scheduling policy is SCHED_APP to set that policy to a different one, a POSIX_APPSCHED_TERMINATE event shall be generated for the current application scheduler of that thread, with a *thread* member equal to the *thread* argument; if that event is masked in that application scheduler it is ignored.

If *pthread_setschedparam*() is called for a thread whose scheduling policy is SCHED_APP and the *policy* argument continues to be SCHED_APP, a POSIX_APPSCHED_CHANGE_SCHED_PARAM event shall be generated for the current application scheduler of that thread, with a *thread* member equal to the *thread* argument; if that event is masked in that application scheduler it is ignored.

If *pthread_setschedparam*() is called for a thread whose scheduling policy is different than SCHED_APP to set that policy to SCHED_APP, a check shall be made to determine if the `appscheduler` attribute of that thread is a valid application scheduler. If not, the function shall fail. If it is valid, a POSIX_APPSCHED_NEW event shall be generated for that application scheduler with a *thread* member equal to the *thread* argument, and the calling thread will be suspended until the application scheduler of *thread* executes an “accept” or “reject” action on that thread, via *posix_appsched_execute_actions*(). If the action is “reject”, the *pthread_setschedparam*() function shall fail. The function shall also fail if the priority of *thread* is larger than the priority of the application scheduler specified in the `appscheduler` attribute of that thread. The *pthread_setschedparam*() function shall fail if the POSIX_APPSCHED_NEW scheduling event is masked by the corresponding application scheduler.

The following new error conditions are defined for *pthread_setschedparam*():

[EINVAL] The `appscheduler` attribute of *thread* does not refer to a valid application scheduler at the time of the call.

The priority of *thread* is larger than the priority of the application scheduler specified in the `appscheduler` attribute of that thread.

[EREJECT] The application scheduler has rejected attachment of the requested thread.

[EMASKED] The operation cannot be executed because the POSIX_APPSCHED_NEW scheduling event is currently masked by the application scheduler.

A.3.3. Thread Creation

If the attributes object used in a call to *pthread_create*() has its scheduling policy set to SCHED_APP, a check shall be made to determine if the `appscheduler` attribute of that attributes object is a valid application scheduler. If not, the function shall fail. If it is valid, after the thread is created, a POSIX_APPSCHED_NEW event shall be generated for that application scheduler with a *thread* member equal to the thread Id of the newly created thread, and the calling thread will be suspended until the application scheduler executes an “accept” or “reject” action on that thread, via *posix_appsched_execute_actions*(). If the action is “reject”, the *pthread_create*() function shall fail. The *pthread_create*() function shall fail if the POSIX_APPSCHED_NEW event is masked by the corresponding application scheduler. The function shall also fail if the priority of the new thread, stored in the *sched_priority* member of the `schedparam` attribute in *attr*, is larger than the priority of the application scheduler specified in the `appscheduler` attribute.

The following new error conditions are defined for *pthread_create*():

[EINVAL] The `appscheduler` attribute of *attr* does not refer to a valid application scheduler at the time of the call.

The priority of the *sched_priority* member of the `schedparam` attribute in *attr* is larger than the priority of the application scheduler specified in the `appscheduler` attribute.

[EREJECT] The application scheduler has rejected attachment of the requested thread.

[EMASKED] The operation cannot be executed because the POSIX_APPSCHED_NEW scheduling event is currently masked by the application scheduler.

A.3.4. Thread termination and cancellation

After execution of the cleanup handlers and of the destructor functions of any thread-specific data by an explicit or implicit call to *pthread_exit*() or by a call to *pthread_cancel*(), if the terminating thread has a scheduling policy of SCHED_APP, a POSIX_APPSCHED_TERMINATE event shall be generated for the application scheduler of the terminating thread with a *thread* member equal to its thread Id. The terminating thread will be permitted to finish its termination operation concurrently with the processing of the generated event by the application scheduler. If the event is masked in that application scheduler, it is just ignored.

The termination of a scheduler thread (either because it calls *pthread_exit*(), or because the execution of its body terminates, or because it is cancelled) which has scheduled threads attached to it has undefined results. Applications should make sure that a scheduler thread only terminates if it has no attached threads.

A.3.4.1. Rationale

Because thread-specific data is destroyed when the thread terminates, the application scheduler processing a POSIX_APPSCHED_TERMINATE event should not attempt using it. If it is necessary for the scheduler to perform operations on the terminating thread just before its termination, it could install a cancellation handler that could invoke the scheduler explicitly. Cancellation handlers are executed before destroying thread-specific data.

A.4. Use of Regular Mutexes by Application-Scheduled Threads

When an application-scheduled thread locks or unlocks a regular mutex (i.e., one that has not been created with the PTHREAD_APPSCHED_PROTOCOL protocol) its priority is changed according to the mutex protocol. In addition, if the mutex uses the priority inheritance or priority ceiling protocols, a scheduling event of type POSIX_APPSCHED_PRIORITY_INHERIT or POSIX_APPSCHED_PRIORITY_UNINHERIT is queued to the scheduler thread at the time a priority is inherited or uninherited. The scheduler thread shall inherit (or uninherit) the same active system priority or priorities of its scheduled threads.

If the application-scheduled thread tries to lock an already locked mutex, it is queued in the mutex queue in the same way as for any other thread. Besides this queuing, a scheduling event of type POSIX_APPSCHED_BLOCK is sent to the scheduler thread, as for any other blocking situation.

The events of type POSIX_APPSCHED_PRIORITY_INHERIT and POSIX_APPSCHED_PRIORITY_UNINHERIT can be used by the application scheduler to keep the thread that locked the mutex in execution. Otherwise, regular threads that use the same mutex could suffer priority inversion due to the execution of other application-scheduled threads.

A.5. Management of Application-Scheduled Mutexes

A.5.1. Data definitions

A.5.1.1. Application-Scheduled Mutex Protocol

The following mutex protocol constant shall be defined in <pthread.h>:

Table 3:

Symbol	Description
PTHREAD_APPSCHED_PROTOCOL	Application-defined protocol

When this protocol is set for a mutex, the mutex shall only be used by threads scheduled under the scheduler referenced by the appscheduler attribute of the mutex. In addition, scheduling events are notified to the scheduler thread under the following circumstances:

- when a thread invokes the lock operation of the mutex,
- when a thread invokes the trylock operation of the mutex,
- when a thread unlocks the mutex,

- when a thread blocks waiting for the mutex,
- when a thread changes the scheduling parameters of the mutex.

A.5.1.2. Application-Scheduled Mutex Attributes

All application-scheduled mutexes shall support the appscheduler attribute that represents the scheduler thread that will schedule threads using that mutex. The default value of this attribute is unspecified. If at the time of the mutex creation the protocol is PTHREAD_APPSCHED_PROTOCOL and the appscheduler attribute does not refer to a valid application scheduler thread, the corresponding *pthread_mutex_init*() operation shall fail with an error of [EINVAL].

In addition, all application-scheduled mutexes shall support the appschedparam attribute, which represents an area of memory containing the application-specific scheduling parameters used for that mutex. The default value of this attribute is a memory area of zero bytes. The maximum size of this attribute in bytes shall be represented by the variable POSIX_APPMUTEXPARAM_MAX (see A.2.1.2).

A.5.2. Application-Scheduled Mutex Attributes Manipulation

A.5.2.1. Synopsis

```
#include <pthread.h>

int pthread_mutexattr_setappscheduler
      (pthread_mutexattr_t *attr,
       pthread_t scheduler);

int pthread_mutexattr_getappscheduler
      (const pthread_mutexattr_t *attr,
       pthread_t *scheduler);

int pthread_mutexattr_setappschedparam
      (pthread_mutexattr_t *attr,
       const void *param,
       size_t param_size);

int pthread_mutexattr_getappschedparam
      (const pthread_mutexattr_t *attr,
       void *param,
       size_t *paramsize);
```

A.5.2.2. Description

The *pthread_mutexattr_setappscheduler*() and *pthread_mutexattr_getappscheduler*() functions shall respectively set and get the appscheduler attribute of the mutex attributes object specified by *attr*. This attribute is described in A.5.1.2. If successful, the *pthread_mutexattr_setappscheduler*() shall set the attribute equal to *scheduler*. For this function to succeed, *scheduler* must be a valid application scheduler at the time of the call.

The *pthread_mutexattr_setappschedparam*() and *pthread_mutexattr_getappschedparam*() functions shall respectively set and get the appschedparam attribute of the mutex attributes object specified by *attr*. This attribute is described in A.5.1.2. If successful, the

pthread_mutexattr_setappschedparam() function shall set the size of the attribute to *paramsize* and shall copy the memory area pointed to by `param` of length *paramsize* into the attribute. If *paramsize* is larger than POSIX_APPMUTEXPARAM_MAX, the function shall fail.

The *pthread_mutexattr_getappschedparam*() function shall copy the contents of the `appschedparam` attribute into the memory area pointed to by *param*. This memory area shall be capable of storing at least a number of bytes equal to the size of the `appschedparam` attribute; otherwise, the results are undefined.

A.5.2.3. Returns

Upon successful completion, *pthread_mutexattr_setappscheduler*(), *pthread_mutexattr_setappschedparam*(), *pthread_mutexattr_getappscheduler*(), and *pthread_mutexattr_getappschedparam*() shall return a value of 0. Otherwise an error number shall be returned to indicate the error.

The *pthread_mutexattr_getappscheduler*() function stores the `appscheduler` attribute in the variable pointed to by *scheduler* if successful.

The *pthread_attr_getappschedparam*() function stores the size of the `appschedparam` attribute in the variable pointed to by *paramsize*, and copies the contents of the `appschedparam` attribute into the memory area pointed to by *param*.

A.5.2.4. Errors

If any of the following conditions occur, the *pthread_mutexattr_setappscheduler*() function shall return the corresponding error number:

[EINVAL] The value of *scheduler* was not valid.

If any of the following conditions occur, the *pthread_mutexattr_setappschedparam*() function shall return the corresponding error number:

[EINVAL] The value of *paramsize* was smaller than zero, or was larger than POSIX_APPMUTEXPARAM_MAX.

If any of the following conditions are detected, the *pthread_mutexattr_setappschedparam*() function shall return the corresponding error number:

[EINVAL] The value of *param* was invalid.

A.5.2.5. Cross-References

pthread_mutex_init().

A.5.3. Dynamically Changing the Application-Scheduled Mutex Attributes

A.5.3.1. Synopsis

```
#include <pthread.h>
```

```
int pthread_mutex_getappscheduler
      (const pthread_mutex_t *mutex,
       pthread_t *scheduler);

int pthread_mutex_setappschedparam
      (pthread_mutex_t *mutex,
       const void *param,
       size_t param_size);

int pthread_mutex_getappschedparam
      (const pthread_mutex_t *mutex,
       void *param,
       size_t *paramsize);
```

A.5.3.2. Description

The *pthread_mutex_getappscheduler*() function is used to dynamically get the appscheduler attribute of the mutex specified by *mutex*. This attribute is described in A.5.1.2.

The *pthread_mutex_setappschedparam*() and *pthread_mutex_getappschedparam*() functions are used to dynamically set and get the appschedparam attribute of the mutex attributes object specified by *mutex*. This attribute is described in A.5.1.2.

If successful, the *pthread_mutex_setappschedparam*() function shall acquire the lock on the mutex. Then it shall set the size of the appschedparam attribute to the value specified by *paramsize*, and it shall copy the scheduling parameters occupying *paramsize* bytes and pointed to by *param* into that attribute. In addition, if the protocol of *mutex* is PTHREAD_APPSCHED_PROTOCOL and if the POSIX_APPSCHED_CHANGE_MUTEX_SCHED_PARAM event is not masked in the application scheduler of that mutex, the function shall generate a POSIX_APPSCHED_CHANGE_MUTEX_SCHED_PARAM event for that application scheduler and shall suspend until the scheduled thread activates it via a *posix_appsched_execute_actions*() call. Finally, the function shall release the lock on the mutex.

For the *pthread_mutex_setappschedparam*() function to succeed, the calling thread must be an application-scheduled thread scheduled by the same thread as the mutex.

The memory area represented by *param* in the call to *pthread_getappschedparam*() shall be capable of storing at least a number of bytes equal to the size of the appschedparam attribute; otherwise, the results are undefined.

A.5.3.3. Returns

Upon successful completion, *pthread_mutex_getappscheduler*(), *pthread_mutex_setappschedparam*(), and *pthread_mutex_getappschedparam*() shall return a value of 0. Otherwise an error number shall be returned to indicate the error.

The *pthread_mutex_getappscheduler*() function stores the appscheduler attribute in the variable pointed to by *scheduler*, if successful.

The *pthread_mutex_getappschedparam*() function stores the size of the appschedparam attribute in the variable pointed to by paramsize, and copies the contents of the appschedparam attribute into the memory area pointed to by *param*, if successful.

A.5.3.4. Errors

If any of the following conditions occur, the *pthread_mutex_setappschedparam*() function shall return the corresponding error number:

[EINVAL] The value of *paramsize* was less than zero, or was larger than POSIX_APPMUTEXPARAM_MAX.

[EPOLICY] The calling thread is not an application-scheduled thread scheduled by the same thread as the mutex.

If any of the following conditions are detected, the *pthread_mutex_setappschedparam*() function shall return the corresponding error number:

[EINVAL] The value of *param* was invalid.

If any of the following conditions are detected, the *pthread_mutex_getappscheduler*(), *pthread_mutex_setappschedparam*(), and *pthread_mutex_getappschedparam*() functions shall return the corresponding error number:

[EINVAL] The value specified by *mutex* does not refer to an initialized mutex object.

If any of the following conditions are detected, the *pthread_mutex_setappschedparam*() function shall return the corresponding error number:

[EDEADLK] The current thread already owns the mutex.

A.5.3.5. Cross-References

pthread_mutex_init(), *pthread_mutexattr_setappscheduler*, *pthread_mutexattr_setappschedparam*().

A.5.3.6. Rationale

The capability of switching a mutex from one application scheduler to another one is not provided because it can be implemented by destroying the mutex and initializing it again, with new attributes.

A.5.4. Mutex-Specific Data

A.5.4.1. Synopsis

```
int posix_appsched_mutex_setspecific
     (pthread_mutex_t *mutex,
      const void *value);
int posix_appsched_mutex_getspecific
     (const pthread_mutex_t *mutex,
      void **value);
```

A.5.4.2. Description

In successful, the *posix_appsched_mutex_setspecific*() function shall associate a mutex-specific *value* with the mutex specified by *mutex*. This value is typically a pointer to a block of dynamically allocated memory that has been reserved for use by the calling thread[1].

In successful, the *posix_appsched_mutex_getspecific*() function shall store in the variable pointed to by *value* the value currently bound to the specified *mutex*.

A.5.4.3. Returns

If successful, the *posix_appsched_mutex_getspecific*() function stores in the variable pointed to by *value* the mutex-specific data value associated with *mutex*; if no mutex-specific data value is associated with it, then the value **NULL** is returned. In both cases, the function shall return zero. If the function fails, an error number shall be returned to indicate the error.

If successful, the *posix_appsched_mutex_setspecific*() function shall return zero. Otherwise, an error number shall be returned to indicate the error.

A.5.4.4. Errors

If any of the following conditions are detected, the *posix_appsched_mutex_setspecific*() and *posix_appsched_mutex_getspecific*() functions shall return the corresponding error number:

[EINVAL] The value specified by *mutex* does not refer to an initialized mutex object.

A.5.4.5. Cross-References

pthread_setspecific(), *pthread_getspecific*(), *pthread_setspecific_for*(), *pthread_getspecific_from*().

A.5.4.6. Rationale

POSIX.1 defines a thread-specific data mechanism that is very useful for attaching information to a particular thread, and in particular, to attach scheduling information. However, the standard does not define a similar functionality for mutexes, which would be very useful in the context of writing application schedulers. For this reason the mutex-specific data interface has been introduced.

A.5.5. Modifications to existing functions: Initializing and Destroying a Mutex

If the mutex attributes object used in a call to *pthread_mutex_init*() has its `protocol` attribute set to POSIX_APPSCHED_PROTOCOL, a check shall be made to determine if the

1. Because sharing data between a scheduler thread and a regular thread has undefined results, the mutex-specific data interface should not be used to exchange or share information between a scheduler thread and its scheduled threads.

`appscheduler` attribute of that attributes object is a valid application scheduler. If not, the function shall fail. If it is valid, after the mutex is initialized, a POSIX_APPSCHED_INIT_MUTEX event shall be generated for that application scheduler with a *mutex* member equal to *mutex*, and the calling thread shall be suspended until the application scheduler executes an "mutex-accept" or "mutex-reject" action on that mutex, via *posix_appsched_execute_actions*(). If the action is "mutex-reject", the *pthread_mutex_init*() function shall fail. The *pthread_mutex_init*() function shall fail if the POSIX_APPSCHED_INIT_MUTEX event is masked by the corresponding application scheduler.

The following new error conditions are defined for *pthread_mutex_init*():

[EINVAL] The `appscheduler` attribute of *attr* does not refer to a valid application scheduler at the time of the call.

[EREJECT] The application scheduler has rejected attachment of the requested mutex.

[EMASKED] The operation cannot be executed because the POSIX_APPSCHED_INIT_MUTEX scheduling event is currently masked by the application scheduler.

If the mutex specified in a call to *pthread_mutex_destroy*() has its `protocol` attribute set to POSIX_APPSCHED_PROTOCOL and the POSIX_APPSCHED_DESTROY_MUTEX event is not masked in the application scheduler of that mutex, after destroying the mutex one such event shall be generated for that application scheduler with a *mutex* member equal to *mutex*, and the calling thread shall be suspended until the application scheduler activates it via a "thread-activate" action in a call to *posix_appsched_execute_actions*().

A.5.6. Modifications to existing functions: Locking and Unlocking a Mutex

If the mutex specified in a call to *pthread_mutex_lock*(), has its `protocol` attribute set to POSIX_APPSCHED_PROTOCOL, before the mutex is locked, a POSIX_APPSCHED_LOCK_MUTEX event shall be generated for the application scheduler of the mutex with a *mutex* member equal to *mutex*, and the calling thread shall be suspended until the application scheduler executes a "mutex-lock" action on that mutex, via *posix_appsched_execute_actions*(). After that action the function shall return with the mutex locked. The *pthread_mutex_lock*() function shall fail if the POSIX_APPSCHED_LOCK_MUTEX event is masked by the corresponding application scheduler.

The *pthread_mutex_timedlock*() function shall behave the same as *pthread_mutex_lock*(), with the additional requirement that if the timeout expires and the POSIX_APPSCHED_READY event is not masked by the application scheduler, the thread discontinues its wait on the mutex, a POSIX_APPSCHED_READY is generated for that scheduler, and the thread is suspended until activated via a "thread-activate" action executed by *posix_appsched_execute_actions*().

The following new error condition is defined for *pthread_mutex_lock*() and *pthread_mutex_timedlock*():

[EMASKED] The operation cannot be executed because the POSIX_APPSCHED_LOCK_MUTEX scheduling event is currently masked by the application scheduler.

If the mutex specified in a call to *pthread_mutex_trylock*(), has its `protocol` attribute set to POSIX_APPSCHED_PROTOCOL, before the mutex is locked, a POSIX_APPSCHED_TRYLOCK_MUTEX event shall be generated for the application scheduler of the mutex with a *mutex* member equal to *mutex*, and the calling thread shall be suspended until the application scheduler executes either a “mutex-lock” or a “thread-activate” action on that mutex, via *posix_appsched_execute_actions*(). If the action is a “mutex-lock”, the function shall return with the mutex locked; if it is a “thread-activate” the function shall return with an error of [EBUSY]. The *pthread_mutex_lock*() function shall fail if the POSIX_APPSCHED_TRYLOCK_MUTEX event is masked by the corresponding application scheduler.

The following new error condition is defined for *pthread_mutex_trylock*():

[EMASKED] The operation cannot be executed because the POSIX_APPSCHED_TRYLOCK_MUTEX scheduling event is currently masked by the application scheduler.

If the mutex specified in a call to *pthread_mutex_unlock*(), has its `protocol` attribute set to POSIX_APPSCHED_PROTOCOL, and the POSIX_APPSCHED_UNLOCK_MUTEX event is not masked in the application scheduler of that mutex, after releasing the mutex but before granting it to another thread one POSIX_APPSCHED_UNLOCK_MUTEX event shall be generated for the application scheduler with a *mutex* member equal to *mutex*, and the calling thread shall be suspended until the application scheduler activates it via a “thread-activate” action in a call to *posix_appsched_execute_actions*().

A.6. OS Implementation Considerations

Each scheduler thread needs a list of all its associated threads, which are scheduled by it. This is a dynamic list that grows when a new thread joins the scheduler, and shrinks when a scheduled thread is terminated or abandons the scheduler.

This list could be used to determine the active system priority of the scheduler thread, taking into account the priorities inherited by the scheduled threads.

Each scheduler thread needs a FIFO queue of scheduling events. There is no need to define a constant for the length of this queue, because it is bounded by two times the maximum number of threads in the system, plus one. This bound is originated because each application-scheduled thread in the system can cause at most one scheduling event before executing again, and it will only execute after the scheduler has processed the generated event. For system scheduled threads, the only events that are relevant are the priority inheritance or uninheritance, so there may be in the worst case two events per such task. And in addition, a single timeout or signal event could be generated.

Bibliografía

[ABE98] Abeni, L. y Buttazzo, G., “Integrating Multimedia Applications in Hard Real-Time Systems”. Proceedings of the IEEE Real-Time Systems Symposium. Madrid, España. Diciembre,1998.

[ADA83] “Reference Manual for the ADA Programming Language”. ANSI/MIL-STD 1815A.

[ADA95] International Standard ISO/IEC 8652:1995(E): “Information technology - Programming languajes - Ada. Ada Reference Manual”.1995.

[ADA00] International Standard ISO/IEC 8652:1995(E) with Technical Corrigendum: “Consolidated Ada Reference Manual. Language and Standard Libraries”. Springer, 2000.

[ALL95] Allen Weiss, M., “Estructuras de datos y algoritmos”, Addison-Wesley Iberoamericana, 1995.

[AUD90] Audsley, N.C. y Burns, A., “Real-Time System Scheduling”. Department of Computer Science, University of York, Technical Report YCS_134.

[AUD91A] Audsley, N.C., “Deadline Monotonic Scheduling”. Department of Computer Science, University of York, Technical Report YCS_146, September 1991.

[AUD91B] Audsley, N.C., “Optimal Priority Assignment and Feasibility of Static Priority Tasks With Arbitrary Start Times”. Department of Computer Science, University of York, Technical Report YCS_164, December 1991.

[BAC86] Bach, Maurice J., “The Design of the UNIX Operating System”. Prentice/Hall International, Inc., 1986.

[BAK88] Baker T.P., Shaw A., “The Cyclic Executive Model and Ada”. Proceedings of the IEEE Real-Time Systems Symposium, December 1988.

[BAK91] Baker T.P., “Stack-Based Scheduling of Realtime Processes”, Journal of Real-Time Systems, Volume 3 , Issue 1 (March 1991), pp. 67–99.

[BAR96] Barnes, J., "Programming in Ada 95". Addison-Wesley Publishing Company, 1996.

[BAR96A] Baruah, S.K., Cohen, N.K., Plaxton, C.G., Varvel, D.A., “Proportionate Progress: A Notion of Fairness in Resource Allocation”. Algorithmica 15(6), pp. 600-625. 1996.

[BAR97] Barabanov, M. y Yodaiken, V., “Introducing Real-Time Linux”. Linux Journal 34. pp. 19-23, 1997.

[BAR97A] Barnes, J., “High Integrity Ada - The SPARK Approach”. Addison-Wesley Professional, ISBN: 0-201-17517-7, 1997.

[BAR97B] Barabanov, M., “A Linux-based Real-Time Operating System”, Master's thesis, New Mexico Institute of Mining and Technology, 1997.

[BER01] Bernat, G. y Burns, A., “Implementing a Flexible Scheduler in Ada”, Lecture Notes in Computer Science 2043, Reliable Software Technologies. Ada-Europe’01, pp. 179-190.

[BOL00] Bollella, G., Gosling, J., Brosgol, B., Dibble, P., Furr, S., Hardin, D., y Turnbull, M., “The Real-Time Specification for Java”, Addison-Wesley, 2000.

[BRO01] Brosgol, B., Dobbing, B., “Can Java™ Meet Its Real-Time Deadlines?”, Lecture Notes in Computer Science 2043, Reliable Software Technologies. Ada-Europe’01, pp. 68-87.

[BUR99] Burns, A., “The Ravenscar Profile”. Technical report, University of York, 1999. Disponible en “http://www.cs.york.ac.uk/rts/papers/p.ps”.

[C99] International Standard ISO/IEC 9899:1999(E). “Programming Languages -- C”, ISO/IEC, 1999.

[C++98] International Standard ISO/IEC 14882:1998(E). “Programming languages - C++”, ISO/IEC, 1998.

[CAN98] Candea, George M., Jones, Michael B., “Vassal: Loadable Scheduler Support for Multi-Policy Scheduling”. Proceedings of the Second USENIX Windows NT Symposium, Seattle, Washington. Agosto, 1998.

[CHE90] Chen, M. y Lin, K., “Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems”. Real-Time Systems, Kluwer Academic Publishers, 2, pp. 325-346, 1990.

[COO00] Coopee, T., “Embedded intelligence”. Noviembre, 2000. (“http:// www.infoworld.com/articles/tc/xml/00/11/20/001120tcembed.xml”).

[DOB01] Dobrin, R., Fohler G. y Peter Puschner, P., “Translating Off-line Schedules into Task Attributes for Fixed Priority Scheduling”. Proceedings of Real-Time Systems Symposium London, UK, diciembre 2001.

[DSE02A] Dedicated Systems Experts, “The QNX NEUTRINO RTOS v6.2 - Full evaluation report, extended test suite”. Release Date: 13/08/2002 - Version 2.50. Disponible en “http://www.dedicated-systems.com/Encyc/BuyersGuide/RTOS/Evaluations/ docs.asp”.

[DSE02B] Dedicated Systems Experts, “The Windows CE.NET (CE 4.0) - Full evaluation report, extended test suite”, Release Date: 21/06/2002 - Version 2.52. Disponible en “http://www.dedicated-systems.com/Encyc/BuyersGuide/RTOS/Evaluations/ docs.asp”.

[DSE02C] Dedicated Systems Experts, “ The VxWorks AE v1.1 - Full evaluation report, extended test suite”. Release Date: 13/05/2002 - Version 2.00. Disponible en “http:/ /www.dedicated-systems.com/Encyc/BuyersGuide/RTOS/Evaluations/docs.asp”.

[EMBED] Embedded Company. “http://www.embedded.com.au/market.html”.

[ESP98] Espinosa, A., Julián, V., Carrascosa, C., Terrasa, A. y García-Fornes, A., “Programming Hard Real-Time Systems with Optional Components in Ada”.

Lecture Notes in Computer Science 1411, Reliable Software Technologies. Ada-Europe'98, pp. 102-111.

[ESP02] Espinosa Minguet, A., García Fornes, A., Crespo i Lorente, A., "An Ada Binding to the IEEE 1003.1q (POSIX Tracing) Standard". Lecture Notes in Computer Science 2361, Reliable Software Technologies - Ada-Europe 2002, pp. 321-333.

[FEL93] Feldman, Michael B., " Data structures with Ada". Addison-Wesley Publishing Company, Inc., 1993.

[FIR02] "Flexible Integrating Scheduling Technology - EU IST Project". "http:// www.idt.mdh.se/salsart/FIRST/".

[FOR96] Ford, B., Susarla, S., "CPU Inheritance Scheduling". Proceedings of OSDI. Octubre, 1996.

[FOR97] Ford, B., Back, G., Benson, G., Lepreau, J., Lin, A., y Shivers, O., "The Flux OSKit: a Substrate for OS and Language Research". Proceedings of the 16th ACM Symposium on Operating Systems Principles, Saint Malo, France, October 1997 (http://www.cs.utah.edu/flux/oskit)

[GAI01] Gai, P., Abeni, L., Giorgi, M. y Buttazzo, G., "A New Kernel Approach for Modular Real-Time Systems Development". IEEE Proceedings of the 13th Euromicro Conference on Real-Time Systems, Delft, Holanda. Junio, 2001.

[GDB98] Stallman, Richard M. y Pesch, Roland H., "Debugging with GDB-The GNU Source-Level Debugger", abril 1998.

[GIE94] Giering, E. W. y Baker, T. P.: "The GNU Ada Runtime Library (GNARL): Design and Implementation". Wadas'94, Proceedings, 1994.

[GNU] GNU's Not Unix - The GNU Project and the Free Software Foundation (FSF), "http://www.fsf.org/".

[GON91] González Harbour, M. y Sha, L.: "An Application-Level Implementation of the Sporadic Server". Technical Report CMU/SEI-91-TR-26, ESD-91-TR-26, September 1991.

[GON93] González Harbour, M., "Real-time POSIX: an Overview". VVConex 93 International Conference, Moscú. Junio 1993.

[GON97] González Harbour, M., Gutierrez Garcia, J.J. y Palencia Gutierrez, J.C.: "Implementing Application-Level Sporadic Server Schedulers in Ada 95". Lecture Notes in Computer Science 1251, Reliable Software Technologies. Ada-Europe'97, pp. 125-136.

[GON97A] González Harbour, M. and Locke, D., "Toasters and POSIX". Remitido para su publicación a Journal of Real-Time Systems, 1997.

[GOS00] Gosling, J., Joy, B., Steele, G., y Bracha, G., "The Java Language Specification, 2 nd Edition". Addison Wesley, 2000.

[GOY96] Goyal, P., Guo, X., Vin, H.M., "A Hierarchical CPU Scheduler for Multimedia Operating Systems". Usenix Association Second Symposium on Operating Systems Design and Implementation (OSDI'96), pp. 107-121. 1996.

[GUT95] Gutiérrez García, J.J. y González Harbour, M., “Increasing Schedulability in Distributed Hard Real-Time Systems”. Proceedings of 7th Euromicro Workshop on Real-Time Systems, IEEE Computer Society Press, pp. 99-106, Junio 1995.

[HPR94] Hewlett-Packard Company, “HP-RT Application Programming in the HP-RT Environment”, Hewlett-Packard, 1994

[INTE3] Intel. “Intel Architecture Software Developer’s Manual. Vol. 3. System Programming”. (ftp://download.intel.nl/design/pentiumii/manuals/24319202.pdf).

[JEF98] Jeffay, K., Smith, F.D., Moorthy, A., Anderson, J., “Proportional Share Scheduling of Operating System Services for Real-Time Applications”. Proceedings of the IEEE Real-Time Systems Symposium, pp. 480-491. Madrid, España. Diciembre,1998.

[JON97] Jones, M.B., Rosu, D., Rosu, M.C., “CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities”. In Proceedings of the 16th ACM Symposium on Operating Systems Principles. Saint-Malo, Francia, pp. 198-211. Octubre, 1997.

[JCONS] J Consortium. “http://www.j-consortium.org/”.

[KER84] Kernighan, B. W. y Pike, R. , “The Unix Programming Environment”. Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[KLE93] Klein, M., Ralya, T., Pollak, B., Obenza, R. y Gonzalez Harbour, M., “A Practitioner's handbook for Real-Time Analisys”. Kluwer Academic Pub., 1993.

[LEH92] Lehoczky, J.P., Ramos-Thuel, S., “An optimal algorithm for scheduling soft-aperiodic tasks in fixed preemptive systems”. Proceedings of the 13th Real-Time Systems Symposium, Phoenix, Arizona, pp. 110-123. Diciembre, 1992.

[LEU82] Leung, J. y Whitehead, J., “On Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks”. Performance Evaluation, 2, pp. 237-250, 1982.

[LNXa] The Linux Documentation Project, “http://www.tldp.org/”.

[LNXb]) The Linux Home Page at Linux Online, “http://www.linux.org/”.

[LIE95] Liedtke, J., “On micro-Kernel Construction”. Proceedings of the Fifteenth ACM Symposium on Operating System Principles, pp. 237-250, diciembre, 1995

[LIU73] Liu, C.L. y Layland, J.W.: “Scheduling Algorithms for Multi-Programming in a Hard Real-Time Environment”. Journal of the Association for Computing Machinery, vol. 20, no. 1, pp. 46-61, January 1973.

[LOC85] Locke, C.D., Jensen, E.D. y Tokuda, H., “A Time-Driven Scheduling Model for Real-Time Operating Systems”. Proceedings of Real-Time System Symposium, pp 112-122.

[LOC92] Locke, C.D., “Software Architecture for Hard Real-Time Applications: Cyclic Executives vs. Fixed Priority Executives”. The Journal of Real-Time Systems, 4, pp. 37-53, 1992.

[LYN96] “LynxOS 2.4. Writing Device Drivers for LynxOS”. Lynx Real-Time Systems, Inc., April 1996.

[LYNXO] LynxOS 4.0. “http://www.lynuxworks.com/products/lynxos/lynxos.php3”.

[MER94] Mercer, C., Savage, S., Tokuda, H., “Processor Capacity Reserves: Operating System Support for Multimedia Applications”. International Conference on Multimedia Computing and Systems, 1994.

[MIR02] Miranda, J., “A Detailed Description of the GNU Ada Run Time”. Disponible en “http://www.iuma.ulpgc.es/users/jmiranda/gnat-rts/main.htm”.

[MOK84] Mok, A. K., “The Design of Real-Time Programming Systems Based on Process Models”. Proceedings IEEE Real-Time Systems Symposium, pp. 5-17, 1984.

[MUE93] Mueller, F., “A library implementation of POSIX threads under UNIX”. Proceedings of the USENIX Conference, pp. 29-41, Enero 1993.

[MUE95] Mueller, F., Rustagi, V. y Baker, T. P., “MiThOS -- A Real-Time Micro-Kernel Threads Operating System”. Proceedings of the 16th IEEE Real-Time Systems Symposium, Pisa, Italy, pp. 49-53, December 4-7, 1995.

[OPENG] The Open Group, “http://www.opengroup.org/”.

[PAR93] Parekh, A.K., Gallager, R.G., “A generalized processor sharing approach to flow control in integrated services networks: the single-node case”. IEEE/ACM Transactions on Networking 1(3), pp. 344-357. 1993.

[PER96] Wall, L., Christiansen, T., Schwartz, R.L., “Programming Perl”, O’Reilly & Associates, 1996.

[PET91] James, L.P. y Silberschatz, A., “Sistemas Operativos. Conceptos fundamentales”. Editorial Reberté, S.A., 1991.

[pSOS] Wind River - pSOSystem 3. “http://www.windriver.com/products/psosystem_3/index.html”.

[PSX93] IEEE Standard 1003.1b:1993, “IEEE Standard for Information Technology - Portable Operating System Interface (POSIX)- Part 1: System Application Program Interface (API) - Amendment 1: Realtime Extension [C Language]”. The Institute of Electrical and Electronics Engineers, 1993.

[PSX96] ISO/IEC Standard 9945-1:1996, “Information Technology -Portable Operating System Interface (POSIX)- Part 1: System Application Program Interface (API) [C Language]”. Institute of Electrical and electronic Engineers, 1996.

[PSX98] IEEE Standard 1003.13:1998, “IEEE Standard for Information Technology - Standardized Application Environment Profile- POSIX Realtime Application Support (AEP)”. The Institute of Electrical and Electronics Engineers, 1998.

[PSX99d] IEEE Standard 1003.1d:1999, “Standard for Information Technology -Portable Operating System Interface (POSIX)- Part 1: System Application Program Interface (API) - Amendment d: Additional Realtime Extensions [C Language]”. The Institute of Electrical and Electronics Engineers, 1999.

[PSX99j] IEEE Standard 1003.1d:1999, “Standard for Information Technology -Portable Operating System Interface (POSIX)- Part 1: System Application Program Interface (API) - Amendment j: Advanced Realtime Extensions [C Language]”. The Institute of Electrical and Electronics Engineers, 1999.

[PSX00q] IEEE Standard 1003.1q:2000, “Draft Standard for Information Technology - Portable Operating System Interface (POSIX)- Part 1: System Application Program Interface (API) [C Language]*. Trace”. The Institute of Electrical and Electronics Engineers, 2000.

[PSX01] IEEE Standard 1003.1:2001, “Standard for Information Technology -Portable Operating System Interface (POSIX)- Part 1: System Application Program Interface (API) [C Language]”. Institute of Electrical and electronic Engineers, 2001.

[PSX02] IEEE Standard 1003.13:2002 Draft 1.2, “Draft Standard for Information Technology -Standardized Application Environment Profile- POSIX Realtime Application Support (AEP)”. The Institute of Electrical and Electronics Engineers, 2002.

[PUE00] De la Puente, Juan A., Ruiz, José F. y Zamorano, J., “An Open Ravenscar Real-Time Kernel for GNAT”. Proceedings 5th International Conference on Reliable Software Technologies - Ada-Europe 2000, Potsdam, Germany, June, 2000, Hubert B. Keller y Erhard Plödereder (Eds.), Lecture Notes in Computer Science, vol. 1845, Springer-Verlag, 2000, ISBN: 3-540-67669-4, pp. 5-15.

[QNX] QNX Software Systems. “http://www.qnx.com/”.

[QNXa] “QNX Neutrino OS - System Architecture”. QNX Software Systems Ltd. 2002. Disponible en “http://www.qnx.com/developer/docs/momentics_nc_docs/neutrino/”.

[QNX92] Hildebrand, D., (QNX Software Systems Ltd.). "An Architectural Overview of QNX". Procceedings of the Usenix Workshop on Micro-Kernels & Other Kernel Architecture", Seattle, April, 1992. ISBN 1-880446-42-1.

[RAJ89] Rajkumar, R., “Task Synchronization in Real-Time Systems”. PhD thesis, Carnegie Mellon University. Agosto, 1989.

[RAM98] Ramamritham, K., Shen(y), C., González, O., Sen, S. y Shirgurkar S., “Using Windows NT for Real-Time Applications: Experimental Observations and Recommendations”. IEEE Real-Time Technology and Applications Symposium, 1998.

[RIP96] Ripoll, I. y Crespo, A.(director), “Planificación con Prioridades Dinámicas en Sistemas de Tiempo Real Crítico”. Tesis doctoral, Valencia, 1996.

[RUB01] Alessandro Rubini, A. y & Jonathan Corbet, J., “Linux Device Drivers, 2nd Edition”. O’Reilly, 2nd Edition June 2001

[RTC01] OMG. “Real-Time CORBA 2.0: Dynamic Scheduling”. Joint Final Submission. OMG Document orbos/2001-06-09, junio, 2001.

[RTE96] "Real-Time Executive for Multiprocessor Systems: Reference Manual". U.S. Army Missile Command, Redstone Arsenal, Alabama, USA, Enero 1996.

[RTLIN] FSMLabs, Real-Time Linux Operating System Web Page, “http://fsmlabs.com/”.

[RUI99] Ruiz, José F., González-Barahona, Jesús M., “Implementing a New Low-Level Tasking Support for the GNAT Runtime System”. Proceedings 4th International Conference on Reliable Software Technologies - Ada-Europe 1999, Michael

González Harbour y Juan A. de la Puente (Eds.), Lecture Notes in Computer Science, vol. 1622, Springer-Verlag, pp. 298-307, 1999.

[RUS99] Rusling, D.A. (1999). "*The Linux Kernel*, Version 0.8-3" (http://www.linuxhq.com/guides/TLK/tlk.html).

[SCH94] Schonberg, E. y Bernard, B., "The GNAT Proyect : A GNU-Ada 9X Compiler". Proceedings of Tri-Ada '94, Baltimore, Maryland, 1994.

[SHA86] Sha, L., Lehoczky, J. P. y Rajkumar, R, "Solutions for Some Practical Problems in Prioritized Preemptive Scheduling". Proceedings of the Real-Time Systems Symposium, pp. 181-191, Diciembre 1986.

[SHA90] Sha, L., Rajkumar, R. y Lehoczky, J.P., "Priority Inheritance Protocols: An Approach to Real-Time Synchronization". IEEE Transaction on Computers, vol. 39, no. 9, pp. 1175-1185, September 1990.

[SHA91] Sha, L., Rajkumar, R., y Lehoczky, J.P., "Real-Time Computing with IEEE Futurebus+", IEEE Micro 11, junio 1991).

[SPR88] Sprunt, B., Lehoczky, J.P. y Sha, L., "Exploiting Unused Periodic Time for Aperiodic Service Using the Extended Priority Exchange Algorithm". Proceedings of the Real-Time System Symposium, pp. 251-258, Diciembre 1988.

[SPR89] Sprunt, B., Sha, L. y Lehoczky, J.P., "Aperiodic Task Scheduling for Hard-Real-Time Systems". The Journal of Real-Time Systems, Kluwer Academic Publishers, 1, pp. 27-60, 1989.

[STA88] Stankovic, J.A. y Ramamritham, K., "Hard Real-Time Systems". IEEE Computer Society, catalog no. EH0276-6, 1988.

[STE78] Department of Defense, "Requirements for High Order Computer Programming Languages STEELMAN", June 1978, AD-A059 444. Disponible en: "http://www.adahome.com/History/Steelman/steelman.htm".

[STR95] Strosnider, J.K., Lehoczky, J.P., y Sha, L. ,"The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-time Environments". IEEE Transactions on Computers, pages 73 - 91, Volume 44, Number 1, Enero 1995.

[TIM00] Timmerman, M. "RTOS Market Survey - Preliminary Results" y "RTOS Market Overview - A follow up", Published in the special edition of Dedicated Systems Magazine. Disponible en "http://www.dedicated-systems.com/encyc/buyersguide/rtos/evaluations/".

[TRI92] Triebel, W.A., "The 80386DX Microprocessor. Hardware, Software, and Interfacing". Prentice-Hall International Editions, 1992.

[VDC00] Venture Development Corporation, "http://www.vdc-corp.com/embedded/press/archives/00/pr00-06.html".

[VXW97] "VxWorks Programmer´s Guide", Wind River Systems, Inc., 1997.

[WAN99] Wang, Y.C. and Lin, K.J., "Implementing a General Real-Time Scheduling Framework in the RED-Linux Real-Time Kernel". Proceedings of IEEE Real-Time Systems Symposium, Phoenix. Diciembre 1999.

[WHE97] Wheeler, D. A., “Ada, C, C++, and Java vs. The Steelman”, Ada Letters, Julio/ Agosto, 1997. Disponible en “http://www.adahome.com/History/Steelman/ steeltab.htm”.

[WINCE] Windows CE .NET Home. “http://www.microsoft.com/windows/embedded/ ce.net/”.

[WINDR] Wind River. “http://www.windriver.com/”.

[WSTS] World Semiconductor Trade Statistics.”http://www.wsts.org/”.

[YOD99] Yodaiken, V., “An RT-Linux Manifesto”. Proceedings of the 5th Linux Expo, Raleigh, North Carolina, USA, May 1999.

[ZEI95] Zeigler, S. F., “Comparing Development Costs of C and Ada”, “http:// www.adauk.org.uk/pubs/zeigler.htm” ,marzo, 1995.