Universidad Politécnica de Valencia
Departamento de Informática de Sistemas y Computadores

Marco de Referencia para la Definición de Planificadores a Nivel de Usuario en Sistemas de Tiempo-Real

Tesis Doctoral presentada por
Arnoldo Díaz Ramírez

Dirigida por
José Ismael Ripoll Ripoll

Valencia, Marzo de 2006

A Armida y Arnoldo
por enseñarme a forjar mis sueños

A Rosy, Paulina y Alan
por hacerlos realidad

Agradecimientos

Quiero expresar mi agradecimiento a algunas de las personas que han sido importantes en el desarrollo de esta tesis.

En primer lugar a Ismael Ripoll, mi asesor, por la confianza, el apoyo y el tiempo dedicado a la consecución de este trabajo. Su calidad profesional y humana han contribuido a que esta experiencia haya sido enriquecedora. También a Alfons Crespo, coordinador del Grupo de Informática Industrial y Sistemas de Tiempo Real de DISCA, por todas las atenciones brindadas. De igual manera a Mercedes Salazar por el apoyo que siempre me ha dado, y por tener siempre una sonrisa y una palabra amable. También a Pedro José Garcia Gil, quien ha estado a nuestro lado desde el primer día, y a Josep Vidal, por su tiempo, buena disposición y paciencia para atender mis dudas siempre que fue necesario.

Agradezco también a mis entrañables amigos de *la pecera*: a Johan, Carlino, Carlos Miguel, Juliana y Tito, Pepe, Guillermo, Jordi, Danilo, Abelardo, Luis y demás compañeros. Su amistad y solidaridad hizo más agradable nuestra estancia en Valencia.

También a los amigos de México, por estar siempre dispuestos a ayudarnos y darnos su cariño: a Luis Eduardo y Chuyita, Lucy Hugues, César Sanz, Jaime Olvera, Luzhelia Llamas, Javier Balderrama, Abif y Rosario, Mily y David, Lourdes y Pancho, Paco y Olga, así como a Eliezer y Queta.

Quiero agradecer de manera especial el apoyo de mi familia: a mi Madre, a Paty y Sergio, Jorge y Ofelia, Víctor y Susana, Niny y Rodolfo. Gracias por hacernos sentir siempre cobijados por su amor.

A las autoridades de la Dirección General de Educación Superior Tecnológica (DGEST), de la Asociación Nacional de Universidades e Instituciones de Educación Superior (ANUIES) y del Instituto Tecnológico de Mexicali, de México, así como del Proyecto de la Unión Europea IST-2001-35102 (Proyecto OCERA), por aportar los medios administrativos y económicos para el desarrollo de esta tesis.

Finalmente, agradezco profundamente a Rosy, Paulina y Alan, por la valentía y alegría con la que afrontaron esta aventura a mi lado, y por hacerlo llenándome de su amor y comprensión. Sin ustedes no hubiera podido.

Índice general

Resumen

Después de más de 25 años de intensa investigación, la planificación de sistemas de tiempo-real ha mostrado una transición que va desde una infraestructura basada en ejecutivos cíclicos, a modelos de planificación más flexibles, tales como planificación basada en prioridades estáticas y dinámicas, planificación de tareas no críticas, y planificación con retroalimentación, por nombrar algunas. A pesar de lo anterior, actualmente tan sólo unas cuantas políticas de planificación están disponibles para la implementación de sistemas de tiempo-real. Por ejemplo, la mayoría de los sistemas operativos de tiempo-real existentes proporcionan únicamente planificación basada en prioridades fijas. Sin embargo, no todos los requerimientos de las aplicaciones de tiempo-real pueden ser satisfactoriamente atendidos utilizando exclusivamente planificación estática. Existen sistemas constituidos por tareas críticas y no críticas que son planificados de mejor manera utilizando planificación basada en prioridades dinámicas. Además, se ha demostrado que la planificación dinámica permite una mayor utilización de los recursos del sistema.

En años recientes, algunos autores han publicado diferentes esquemas para integrar nuevas políticas de planificación a un sistema operativo. Algunos de ellos proponen que los nuevos servicios de planificación se implementen a nivel de usuario, evitando así que la estructura interna del sistema operativo tenga que ser modificada, y ofreciendo la oportunidad de implementar y probar muchos de los resultados generados por el trabajo de investigación en el área de planificación de sistemas de tiempo-real.

De entre los trabajos relacionados publicados a la fecha, destaca el *Modelo para la Planificación Definida por el Usuario*, propuesto por Mario Aldea y Michael González-Harbour. El modelo presenta una Interfaz para Programas de Aplicación (API) que permite crear y utilizar planificadores a nivel de usuario de manera compatible con el modelo de planificación propuesto por POSIX.

Esta tesis se centra en el estudio de la definición de planificadores a nivel de usuario en sistemas de tiempo-real, y en particular en el estudio del modelo para la planificación definida por el usuario compatible con POSIX, para identificar los problemas no resueltos y proponer algunas extensiones al modelo. Además de

proponer una estrategia de implementación que minimice la sobrecarga, en esta tesis se propone una extensión a la interfaz del modelo que simplifique la implementación de aplicaciones de tiempo-real, proporcione portabilidad y sirva como marco de referencia para la creación y uso de planificadores a nivel de usuario.

Resum

Després de més de 25 anys d'intensa investigació, la planificació de sistemes de temps-real ha mostrat una transició que va des d'una infraestructura basada en executius cíclics fins a models de planificació més flexibles, com ara la planificació basada en prioritats estàtiques i dinàmiques, la planificació de tasques no crítiques i la planificació amb retroalimentació, per nomenar algunes. Malgrat això, actualment només unes quantes polítiques de planificació estan disponibles per a la implementació de sistemes de temps-real. Per exemple, la majoria dels sistemes operatius de temps-real existents proporcionen únicament planificació basada en prioritats fixes. Tanmateix, no tots els requeriments de les aplicacions de temps-real poden ser satisfactòriament atesos utilitzant exclusivament planificació estàtica. Existeixen sistemes constituits per tasques crítiques i no crítiques que són planificades de la millor manera utilitzant planificació basada en prioritats dinàmiques. A més a més, s'ha demostrat que la planificació dinàmica permet una major utilització dels recursos del sistema.

En els darrers anys, alguns autors han publicat diferents esquemes per integrar noves polítiques de planificació a un sistema operatiu. Alguns d'ells proposen que els nous serveis de planificació s'implementen a nivell d'usuari, procurant d'esta manera que l'estructura interna del sistema operatiu no haja de ser modificada i oferint l'oportunitat d'implementar i provar molts dels resultats generats pel treball d'investigació en l'àrea de planificació de sistemes de temps-real.

Entre els treballs relacionats publicats fins hui, destaca el \textit{Model per a la Planificació Definida per l'usuari}, propost per Mario Aldea i Michael González-Harbour. El model presenta una Interfície per a Programes d'Aplicació (API) que permet crear i utilitzar planificadors a nivell d'usuari de manera compatible amb el model de planificació propost per POSIX.

Aquesta tesi es centra en l'estudi de la definició de planificadors a nivell d'usuari en sistemes de temps-real i, particularment, en l'estudi del model per a la planificació definida per l'usuari compatible amb POSIX, per a identificar els problemes no resolts i proposar algunes extensions al model. A més de proposar una estratègia d'implementació que minimitze la sobrecàrrega, en aquesta tesi es proposa una extensió a la interfície del model que simplifique la implementa-

ció d'aplicacions de temps-real, proporcione portabilitat i servisca com a marc de referència per a la creació i l'ús de planificadors a nivell d'usuari.

Abstract

After more than 25 years of active research, real-time scheduling theory has shown a transition from cyclical executive based infrastructure to a more flexible scheduling models, such as fixed-priority scheduling, dynamic-priority scheduling, soft real-time applications, feedback scheduling, to name a few. Nevertheless, just a few scheduling algorithms are available nowadays to implement real-time applications. For instance, almost every existing real-time operating system provides only POSIX-compliant fixed-priority scheduling , but not every real-time application requirement can be efficiently fulfilled using only fixed-priority scheduling. There are real-time systems constituted of hard and soft real-time tasks that could be better scheduled using dynamic-priority policies. Also, it's been shown that dynamic-priority scheduling allows a better utilization of system resources.

Recently, a few proposals have been published to integrate more scheduling policies into a real-time operating system, offering an option to expand the real-time operating systems scheduling capabilities, and to implement, prototype and test many theoretical real-time scheduling research work. Some of the authors propose that the new scheduling services be implemented as user-level schedulers, avoiding the need of modifying the operating system kernel.

From the related work published to date, the Application-Defined Scheduling model proposed by Mario Aldea and Michael Gonzalez-Harbour deserves special attention. They proposed an Application Program Interface (API) to create and use application-defined schedulers in a way compatible with the scheduling model defined in POSIX.

This thesis is about the study of the definition of user-level schedulers in real-time systems, and particularly discusses the POSIX-compatible Application-Defined Scheduling model, in order to identify problems not solved and to propose some extensions to the model. Furthermore, it proposes an implementation strategy of user-level scheduler to minimize overhead. The proposed model in this thesis simplifies real-time systems implementation, provides portability, and serves as a framework to develop and use user-level schedulers.

Nomenclatura

En el siguiente cuadro se presentan los símbolos y abreviaturas utilizadas en esta tesis, así como su significado.

Término	Significado
τ	Un conjunto de tareas
n	Número de tareas, cardinal de τ
τ_i	Una tarea cualquiera
C_i	Tiempo de ejecución en el peor caso de τ_i
D_i	Plazo relativo de τ_i
d_i	Plazo absoluto de τ_i
P_i	Periodo de τ_i
r_i	Tiempo de liberación de τ_i
ϕ_i	Fase de τ_i
ρ	Un conjunto de recursos
m	Número de recursos, cardinal de ρ
ρ_i	Un recurso cualquiera
H	Hiperperiodo de un conjunto de tareas
$U\tau$	Factor de utilización total del procesador
u_i	Factor de utilización de τ_i
P_S	Periodo del servidor
Q_S	Crédito máximo del servidor
e_S	Crédito del servidor
U_S	Ancho de banda del servidor
π_i	Prioridad de τ_i
$\Pi(\rho_i)$	Techo de prioridad del recurso ρ_i
Π	Techo de prioridad del sistema

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

1.1. Motivación

Los sistemas de tiempo-real son aquellos cuyo correcto funcionamiento depende no sólo de la exactitud de los resultados que genere, sino también del tiempo en el cual tales resultados son obtenidos [84]. Un sistema de tiempo-real es, por lo tanto, evaluado por su capacidad para generar resultados en el tiempo especificado.

Un sistema de tiempo-real está generalmente formado por un *sistema a controlar* y un *sistema controlador*. El sistema a controlar puede consistir en una línea de producción automatizada con el uso robots, por ejemplo, mientras que el sistema controlador consiste de ordenadores y programas que administran y coordinan las actividades de los robots en la línea de producción. El sistema a controlar puede considerarse como el entorno con el cual el sistema interactúa.

Los sistemas de tiempo-real tienen aplicación en muchas áreas tales como control digital, procesamiento de señales, sistemas de telecomunicación y sistemas multimedia. Ejemplos de este tipo de sistemas son las aplicaciones de robótica [96], aviónica [25], sistemas de control de procesos [106], navegación espacial, bases de datos y aplicaciones multimedia, entre otras.

Un sistema de tiempo-real está comúnmente compuesto por *tareas* y cada una de ellas está sujeta a una serie de restricciones temporales. La restricción temporal más importante de las tareas es el *plazo*, que representa el instante de tiempo máximo en el que una tarea debe terminar su ejecución. El valor que la tarea aporta al sistema depende de si esta termina o no dentro del plazo especificado, tomando como referencia su instante de activación. Los plazos se clasifican de la siguiente manera [20]:

- *Plazos estrictos*. Es crucial que una tarea concluya su ejecución antes de su plazo ya que de otra manera los resultados pueden ser catastróficos. El

valor que la tarea aporta al sistema es nulo si no se ejecuta antes de su plazo y máximo si lo hace antes.

- *Plazos no estrictos*. Las consecuencias de que alguna tarea no termine su ejecución antes de su plazo no ponen en riesgo la integridad del sistema. El valor que la tarea aporta al sistema es máximo si se ejecuta antes de su plazo, y se decrementa de manera directamente proporcional al tiempo de terminación posterior al plazo.

- *Plazos firmes*. Si la tarea no termina su ejecución antes del plazo, la integridad del sistema no se compromete, y el valor que la tarea aporta al sistema es nulo en este caso. Son similares a los plazos no estrictos, con la diferencia de que si la tarea no cumple su plazo su aportación al sistema es cero.

De acuerdo a los plazos de sus tareas, los sistemas pueden clasificarse en sistemas de tiempo-real *críticos* y *no críticos*. En el caso de los primeros al menos una de sus tareas tiene plazo estricto. Es común encontrar sistemas de tiempo-real compuestos de una combinación de tareas críticas (plazos estrictos) y tareas no críticas (plazos no estrictos o firmes).

En un sistema de tiempo-real, el *planificador* es el responsable de asignar recursos e intervalos de tiempo a las tareas, de tal manera que sus restricciones temporales se cumplan. Para cada conjunto de tareas el planificador genera un *plan*, que representa la manera en que a las tareas se les asignan los procesadores disponibles en el sistema. La forma en que el planificador lleva a cabo estas asignaciones depende del *algoritmo de planificación* utilizado.

Los algoritmos de planificación pueden ser clasificados en *estáticos* y *dinámicos* [27]. El enfoque estático establece el plan *a priori*, antes de la ejecución del sistema y requiere conocimiento previo de las características de las tareas. Los planificadores estáticos, en la forma de ejecutivos cíclicos, calculan el plan *off-line* que posteriormente es almacenado en memoria, de tal forma que durante la ejecución del sistema se tiene establecido el tiempo de inicio y duración de ejecución de cada tarea. En contraste, los planificadores dinámicos no hacen uso de un plan pre-calculado sino que toman las decisiones de planificación en tiempo de ejecución, tomando en consideración el estado y los atributos temporales de las tareas que están listas para ser ejecutadas, permitiendo de esta forma una mayor flexibilidad que los estáticos. Los planificadores dinámicos más utilizados son los *basados en prioridades*, y las prioridades de las tareas a su vez, pueden ser asignadas de manera *fija (estática*) o *dinámica*. Esta tesis se centrará en el ámbito de los planificadores dinámicos basados en prioridades.

En los sistemas de tiempo-real es importante comprobar si el conjunto de tareas es *viable* o *planificable.* Un sistema de tiempo-real es viable si puede garantizar que las tareas que lo conforman cumplan sus plazos de ejecución de acuerdo

a algún algoritmo de planificación determinado. Para comprobar si una política de planificación puede garantizar la viabilidad del sistema es necesario contar con herramientas formales de análisis. A este tipo de herramientas se les conoce como pruebas o *tests de planificabilidad*. Si el conjunto de tareas de un sistema satisface el test de planificabilidad para una política de planificación determinada, será condición suficiente para que las tareas críticas cumplan sus plazos. Por otra parte, es deseable que la satisfacción del test sea condición necesaria para la planificación de las tareas críticas.

Una buena cantidad de tests de planificabilidad han sido propuestos a la fecha tanto para algoritmos basados en prioridades fijas como dinámicas. Los tests de planificabilidad no sólo han ayudado a comprobar si un conjunto de tareas es planificable utilizando un algoritmo de planificación determinado, sino que también han sido útiles para establecer algunas propiedades interesantes de los propios algoritmos de planificación, entre ellas la utilización del procesador. Esto ha permitido demostrar que los algoritmos basados en prioridades dinámicas consiguen una mayor utilización que los basados en algoritmos estáticos [21]. Sin embargo, la inmensa mayoría de sistemas operativos de tiempo-real ofrecen tan sólo planificación basada en prioridades fijas [92], lo que limita la eficiencia de los sistemas de tiempo-real. Por otra parte, agregar más políticas de planificación a los sistemas operativos de tiempo-real puede resultar complejo ya que debe modificarse el núcleo del sistema operativo, y tomando en consideración la gran cantidad de políticas de planificación existentes es difícil determinar cuáles son las que deben agregarse.

El problema de la integración de nuevas políticas de planificación a un sistema operativo a sido abordado en los últimos años, y diversas propuestas han sido publicadas. Algunas de ellas proponen que las nuevas políticas de planificación sean implementadas a nivel de usuario, sin necesidad de agregarlas en el núcleo del sistema operativo. Este esquema reduce significativamente el tiempo de implementación pero puede introducir una sobrecarga considerable. La definición de planificadores a nivel de usuario debe, por lo tanto cumplir, con las siguientes condiciones:

1. Debe incluir una interfaz completa, que permita la implementación de la mayoría de las políticas de planificación existentes.

2. Debe ser flexible y que permita seleccionar la mejor estrategia de implementación para cada planificador a nivel de usuario.

3. Debe proporcionar portabilidad, de tal manera que los nuevos servicios de planificación puedan ser utilizados de manera consistente, independientemente de la forma en que hayan sido implementados.

4. Debe ser eficiente, minimizando la sobrecarga del sistema introducida al utilizar los planificadores definidos a nivel de usuario.

5. Debe ser preferentemente compatible con el modelo de planificación definido por algún estándar de tiempo-real, para así facilitar la portabilidad y el uso de los servicios de tiempo-real existentes.

De entre las propuestas publicadas a la fecha, merece especial atención el *Modelo para la Planificación Definida por el Usuario compatible con POSIX*[1], propuesto por Mario Aldea y Michael González-Harbour en [4], que presenta una Interfaz para Programas de Aplicación (API, por sus siglas en inglés) que permite crear y utilizar planificadores definidos por el usuario de una manera compatible con POSIX, sin necesidad de modificar la estructura interna del núcleo del sistema operativo. Sin embargo, a pesar de que la interfaz propuesta cumple con muchas de las condiciones anteriormente descritas, puede extenderse para facilitar la implementación de sistemas de tiempo-real.

1.2. Objetivos

Esta tesis se centra en el estudio de la definición de planificadores a nivel de usuario, y en particular en el estudio del modelo para la planificación definida por el usuario, extendiendo su interfaz y proponiendo un marco de referencia para la implementación y uso de bibliotecas de planificadores. En concreto, los objetivos de esta tesis han sido:

1. Estudiar las políticas de planificación más importantes publicadas a la fecha, identificando sus características, de tal manera que puedan ser implementadas a nivel de usuario.

2. Estudiar las propuestas existentes para la definición de planificadores a nivel de usuario.

3. Revisar el modelo para la planificación definida por el usuario compatible con POSIX.

4. Identificar mejoras al modelo para la planificación definida por el usuario compatible con POSIX.

5. Determinar la mejor estrategia de implementación el modelo para la planificación definida por el usuario en un sistema operativo de tiempo-real.

[1] POSIX es una marca registrada por el IEEE

6. Extender la interfaz de la planificación definida por el usuario para facilitar el desarrollo de sistemas de tiempo-real, enmarcada en el uso de una biblioteca de planificadores definidos por el usuario.

7. Integrar a la interfaz propuesta, prestaciones no presentes en el modelo para la planificación definida por el usuario ni en POSIX y que sean consideradas de interés para la implementación de sistemas de tiempo-real.

8. Desarrollar una biblioteca de planificadores definidos por el usuario en base al marco de referencia propuesto.

9. Integrar a la propuesta prestaciones de tolerancia a fallos.

10. Comparar el desempeño de los planificadores definidos por el usuario utilizando la interfaz propuesta, con planificadores incluidos en el núcleo del sistema operativo.

1.3. Contenido

Esta tesis está estructurada de la siguiente manera:

En el capítulo 2 se presentan los conceptos básicos relacionados con los sistemas de tiempo-real, y particularmente la planificación de tareas y el modelo computacional asumido en esta tesis. En ese capítulo se estudian algunas políticas de planificación más importantes propuestas a la fecha, para planificar sistemas de tiempo-real con diferentes características.

En el capítulo 3 se estudia el estándar POSIX de tiempo-real, haciendo especial énfasis en los aspectos del estándar relacionados con la planificación de tareas.

El capítulo 4 se dedica al estudio del estado del arte de la definición de planificadores a nivel de usuario, y se presentan de manera cronológica las propuestas publicadas a la fecha. En el mismo capítulo se estudia con profundidad el modelo para la planificación definida por el usuario propuesto por Aldea y González-Harbour , así como su Interfaz para Programas de Aplicación. Posteriormente se analiza y propone una estrategia de implementación en un sistema operativo de tiempo-real.

En el capítulo 5 se discute la planificación definida por el usuario y se proponen algunas extensiones al modelo. Además, se propone una interfaz para el uso de planificadores definidos por el usuario que permite la construcción de sistemas de tiempo-real de una manera sencilla y consistente.

En el capítulo 6 se presenta un marco de referencia para la construcción de planificadores definidos por el usuario que serán utilizados con la interfaz propuesta.

Se presentará también la manera en que se integran prestaciones de tolerancia a fallos al modelo propuesto, y las guías para que los nuevos planificadores puedan beneficiarse de ellas.

Para ilustrar el modelo propuesto en el capítulo 7 se muestran algunos ejemplos y casos de estudio.

Finalmente, en el capítulo 8 se presentan las conclusiones de la tesis y las líneas de trabajo futuro.

Capítulo 2

Planificación de Sistemas de Tiempo-Real

2.1. Conceptos básicos

Un sistema de tiempo-real, como se ha explicado, se define como un sistema cuyo correcto funcionamiento depende no sólo de la exactitud de los resultados que genere, sino también del tiempo en el cual tales resultados son obtenidos. Muchos de los sistemas de tiempo-real se caracterizan por que las consecuencias de que los resultados no se generen a tiempo pueden ser catastróficas, ocasionando pérdidas económicas e inclusive humanas. Debido a lo anterior, en esta clase de sistemas informáticos, en ocasiones es preferible aceptar respuestas imprecisas o subóptimas dentro de los límites de tiempo esperados, a no recibir respuesta alguna [49].

Un sistema de tiempo-real está comúnmente formado por un sistema físico basado en un conjunto de sensores, procesadores y actuadores que forman lo que se denomina el subsistema a controlar, y por un sistema informático formado por un grupo de programas (software) que se ejecutan en los procesadores y que conforman el sistema controlador. El sistema informático interactúa continuamente con el sistema físico a controlar.

Debido a la naturaleza de los sistemas de tiempo-real resulta poco práctico que una tarea se encargue de controlar todo el sistema, por lo que es común que las funciones del sistema se dividan entre un conjunto de tareas, cada una de ellas con objetivos y restricciones temporales específicas. Las tareas pueden ejecutarse en uno o más de los procesadores presentes en el sistema, pero en el ámbito de la tesis se asume que en el sistema existe un sólo procesador y dos o más tareas ejecutándose concurrentemente.

La planificación de tareas es probablemente el área más intensamente estu-

diada en los sistemas de tiempo-real, ya que la característica más importante de este tipo de sistemas informáticos es la de garantizar que todas sus tareas cumplan con sus restricciones temporales. El problema al que se enfrenta la planificación consiste en que las tareas que se ejecutan concurrentemente compiten entre sí por los recursos del sistema, incluyendo entre ellos al procesador, y que estos deben ser asignados de tal manera que las tareas cumplan sus plazos. A continuación se presentan algunos conceptos básicos de la planificación de tareas y que serán de gran utilidad para posteriormente estudiar el modelo de sistema.

Al instante de tiempo en que una tarea está lista para ser ejecutada se le conoce como *tiempo de liberación*[1]. A partir de ese momento la tarea puede ser planificada y ejecutada siempre y cuando las condiciones necesarias para su ejecución se hayan satisfecho. Cuando una tarea es liberada o activada regularmente y en tasas fijas de tiempo se dice que la tarea es *periódica*. Al intervalo de tiempo fijo entre activaciones sucesivas se le llama *periodo*. El comportamiento periódico de las tareas se ajusta correctamente a la naturaleza de la mayoría de los sistemas digitales de control, que ejecutan sus acciones de manera continua en intervalos de tiempo fijos o periódicos. Un conjunto de tareas es llamado conjunto de *tareas periódicas síncronas* si todas sus primeras instancias tienen el mismo tiempo de liberación. Por otra parte, si el tiempo inicial de liberación no es el mismo para todas las tareas, se dice que son *tareas periódicas asíncronas*. Un sistema de tiempo puede contener un conjunto de tareas *híbrido*, formado por tareas periódicas síncronas y asíncronas.

Si la tarea no es periódica puede hacerse una distinción entre tareas *aperiódicas*[2] y *esporádicas*. Las tareas aperiódicas tienen tiempos de liberación irregulares, desconocidos y no acotados. Las tareas esporádicas también tienen tiempos de liberación irregulares, pero a diferencia de las no periódicas la tasa de sus tiempos de liberación es conocida y acotada. Esta tasa es representada generalmente por un tiempo mínimo entre activaciones sucesivas.

El *plazo* es el instante de tiempo en el cuál la tarea debe terminar su ejecución. Se ha explicado ya que el plazo puede ser *crítico, no crítico* o *firme,* y que la aportación de la tarea al sistema depende del tipo de plazo y del instante en que la tarea termina. Los plazos se expresan comúnmente como *plazos relativos*, que representan el tiempo máximo en que una tarea debe ejecutarse. Sin embargo, en ocasiones se expresan también como *plazos absolutos*, que se obtienen sumando el tiempo de liberación y el plazo relativo.

El *tiempo de respuesta* se define como el tiempo transcurrido entre la liberación y la terminación de una tarea. Si bien es cierto que es suficiente que una tarea se ejecute dentro de su plazo, en ocasiones puede ser de interés el que un sistema

[1] release time
[2] aperiodic

de tiempo minimice los tiempos de respuesta, como se verá más adelante.

Tomando en consideración las restricciones temporales y en particular los plazos de sus tareas, un sistema de tiempo-real puede ser *crítico* o *no crítico*. Un sistema de tiempo-real crítico es aquél en el que al menos una de sus tareas tiene plazos críticos. El requerimiento de que las restricciones temporales críticas (plazos) de un sistema se cumplan debe ser verificado invariablemente introduce muchas restricciones en el diseño e implementación de sistemas de tiempo-real, así como en la arquitectura de hardware y en el sistema operativo utilizado. Se ha comentado ya que además de competir por el procesador las tareas compiten entre sí por otros recursos y esta contención introduce aún más restricciones en los sistemas críticos. Por esta razón, al llevar a cabo el análisis de la planificabilidad de un conjunto de tareas debe definirse primero un modelo de sistema en el que se establezcan las restricciones y suposiciones del conjunto de tareas. En la siguiente sección se estudiarán algunos modelos de sistema y se describirá aquél en el que se basa esta tesis. También se analizarán diversas políticas de planificación, identificando las ventajas y desventajas de cada una de ellas.

2.2. Modelo de Sistema

El modelo general de un sistema de tiempo-real se muestra en la Figura 2.1, en donde se observa que el sistema está conformado por un conjunto de aplicaciones, procesadores y recursos. Entre ellos se encuentran los *algoritmos* de *planificación* y de *acceso a recursos* utilizados por el sistema operativo.

Cada aplicación de tiempo-real está formada por un conjunto de n tareas τ_i:

$$\tau = \{\tau_i, \text{ i} \in [1..\text{n}]\}$$

Los parámetros más comúnmente utilizados [77][28] para caracterizar a las tareas son el tiempo de cómputo, el plazo relativo y el periodo de activación. El *tiempo de cómputo* o *tiempo de ejecución en el peor caso*[3], C, representa el máximo tiempo de cómputo que requerirá la tarea para ejecutarse. Se dice que una tarea τ tiene un plazo D si su ejecución debe completarse a más tardar en un tiempo D. Por otra parte, el periodo P de la tarea representa la cantidad fija de tiempo transcurrida entre dos activaciones sucesivas. Una tarea, por lo tanto, puede estar caracterizada como:

$$\tau_i = \{C_i, D_i, P_i\}$$

Una tarea τ tiene un tiempo de liberación r si su ejecución puede iniciar únicamente en el instante de tiempo $t \geq r$.

Para cada tarea periódica τ_i el tiempo de liberación está dado por:

[3]worst case execution time

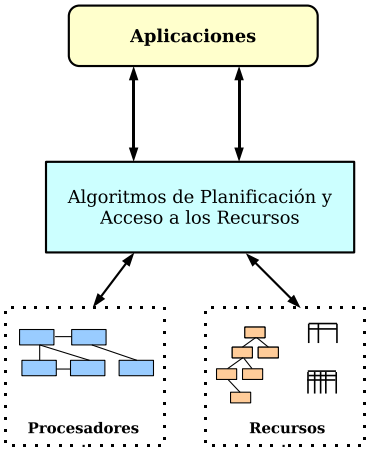

Figura 2.1: Modelo de Sistema de Tiempo-Real

$$r_i = (j - 1)\, P_i$$

en donde *j* representa la $j_{ésima}$ activación de la tarea τ_i.

El tiempo de liberación de la primera instancia de la tarea τ_i es llamado *fase* de τ_i, y se denota como ϕ_i. Si $\phi_i = 0$ significa que la primera instancia de la tarea τ_i es liberada en el instante cero.

El plazo absoluto de la tarea τ_i está dado por:

$$d_i = r_i + D_i$$

El sistema de tiempo-real puede además estar formado por *m* recursos:

$$\rho = \{\rho_i, i \in [1..m]\}$$

Para muchos protocolos de control de acceso a los recursos, la característica de mayor interés de cada recurso es el máximo tiempo de ejecución requerido por las operaciones propias del recurso. Si alguna tarea hace uso de recursos debe considerar en su tiempo *C* el tiempo de ejecución de peor caso de cada uno de los recursos que utilice.

A las tareas les es asignado el procesador y los recursos de acuerdo a las políticas de planificación y acceso a los recursos definidas. Se ha mencionado ya que el módulo encargado de implementar estas políticas se le denomina *planificador*. También se hizo mención a que a la asignación de procesadores y recursos a las tareas se le conoce como *plan*[4]. Los planes generados por el planificador deben ser válidos, lo que significa que satisfacen las siguientes condiciones:

[4]schedule

- A cada procesador se le asigna un máximo de una tarea en cualquier instante.
- Cada tarea es ejecutada en un máximo de un procesador en cualquier instante.
- Ninguna tarea es planificada antes de estar activa.
- Tomando en consideración el algoritmo de planificación utilizado, el tiempo total de tiempo de procesador asignado a cada tarea es igual a su máximo o actual tiempo de cómputo.
- Todas las restricciones de precedencia y de uso de recursos se satisfacen.

Un plan válido es un plan viable si cada tarea cumple con su plazo. Un conjunto de tareas es planificable de acuerdo a algún algoritmo de planificación determinado si este produce siempre un plan viable. Es importante destacar que la planificabilidad es una característica del conjunto de tareas planificadas bajo alguna política y no lo es tanto de la política de planificación en sí. Para evaluar el desempeño de un algoritmo de planificación el criterio más utilizado es su habilidad para encontrar planes viables siempre que tales planes existan. Se define a un algoritmo de planificación como óptimo si es capaz de producir un plan viable siempre que el conjunto de tareas sea planificable. De la misma manera puede decirse que si un algoritmo óptimo es incapaz de generar un plan viable para un conjunto de tareas, dicho conjunto de tareas no puede ser planificado por algún algoritmo.

Existen otros criterios [76] que pueden ser utilizados para evaluar el desempeño de un algoritmo de planificación además del de planificabilidad expuesto anteriormente: la *tardanza*[5], el *retraso*[6], el *tiempo de respuesta*, y las *tasas* de *incumplimiento*[7], de *pérdida*[8] y de *invalidación*[9].

La *tardanza* de una tarea se define como la diferencia entre su tiempo de terminación y su plazo. Su valor será un número negativo si la tarea concluye su ejecución antes de su plazo y positivo si lo hace después. El *retraso* es similar, con la salvedad de que su valor sólo podrá ser un número positivo, lo que significa que es cero si la tarea termina antes o en su plazo y positivo si termina en un tiempo posterior.

El *tiempo de respuesta* de una tarea se mide desde el instante en que se activa hasta que termina su ejecución. En sistemas de tiempo-real compuestos por tareas

[5] tardiness
[6] lateness
[7] miss rate
[8] loss rate
[9] invalid rate

esporádicas puede utilizarse este parámetro para evaluar el desempeño de algún algoritmo de planificación. Así, cuanto menor sea el tiempo de respuesta promedio de las tareas del sistema, mejor será el algoritmo. Si el sistema está compuesto de una combinación de tareas críticas y esporádicas, la política de planificación a utilizar pudiera ser una que garantice el cumplimiento de los plazos estrictos y que minimice los tiempos de respuesta de las tareas esporádicas.

Un sistema de tiempo-real pudiera tolerar que algunas de sus tareas no críticas terminaran su ejecución después de sus plazos, mientras que otro pudiera descartar todas aquellas tareas no críticas que incumplan su plazo. Para este tipo de sistemas las *tasas* de *incumplimiento*, *pérdida* e *invalidación* resultan útiles. La primera proporciona el porcentaje de tareas que concluyen su ejecución después de su plazo, mientras que la *tasa de pérdida* representa el porcentaje de tareas que no se ejecutaron y su plazo se cumplió. La *tasa de invalidación* se calcula sumando las *tasas de incumplimiento* y *pérdida*, y representa el porcentaje de todas aquellas tareas que no produjeron resultados útiles.

Se ha hecho mención a que cada algoritmo de planificación debe disponer de un test de viabilidad para verificar la planificabilidad del conjunto de tareas. Cada test de viabilidad, a su vez, establece las suposiciones y restricciones en base a las cuáles fue formulado. A este conjunto de suposiciones y restricciones se le llama *modelo de sistema*.

El modelo de sistema sobre los que se desarrollaron los primeros tests de viabilidad es conocido como el *modelo de tareas periódicas* [77] y considera las restricciones que se describen a continuación:

- Todas las tareas críticas son periódicas.
- Las tareas pueden ejecutarse en cuanto estén activas.
- Los plazos son iguales a los periodos.
- Las tareas son independientes ya que no comparten recursos, no existen dependencias entre ellas, ni existen restricciones en sus tiempos de liberación o de ejecución.
- No existe sobrecarga en el sistema atribuible a cambios de contexto o a la planificación de las tareas.
- Las tareas pueden ser expulsadas del procesador en cualquier instante.
- Ninguna tarea puede suspenderse voluntariamente.

Es importante mencionar que se asume que los periodos y tiempos de cómputo de las tareas periódicas del sistema son conocidos en todo momento. Tomando como

base este modelo se han desarrollado una gran variedad de algoritmos de planificación que han mostrado tener un buen rendimiento. Por otra parte, no todos los sistemas de tiempo-real cumplen con los requisitos arriba expuestos, pero para poder analizarlos y verificar su viabilidad algunas consideraciones útiles pueden hacerse. Por ejemplo, las tareas esporádicas consideradas como críticas pueden ser transformadas en periódicas usando su mínimo tiempo entre activaciones sucesivas como base. Las restricciones se han podido relajar a medida que la teoría de planificabilidad ha evolucionado, permitiendo así el análisis de sistemas más complejos y la aparición de herramientas formales para su respectivo análisis. Sin embargo el modelo de tareas periódicas representa un buen punto de partida para estudiar con más detalle la planificación de tareas de tiempo-real. En el ámbito de esta tesis se considerará el modelo de tareas periódico como base y en algunos casos las restricciones cambiarán para representar modelos más complejos. Algunos parámetros importantes del modelo se describen a continuación.

Siendo P_i el periodo de la tarea τ_i , se tiene que H representa el mínimo común múltiplo de P_i para $i = 1, 2, ..n$. Al intervalo de tiempo de longitud H se le denomina el *hiperperiodo* del conjunto de tareas. El número máximo de activaciones de tareas en el hiperperiodo está dado por $\sum_{i=1}^{n} \frac{H}{P_i}$. Por ejemplo, la longitud del hiperperiodo del conjunto formado por tres tareas con periodos 3, 5 y 12 es 60, mientras que el número total de activaciones de tareas en el hiperperiodo es 37.

La razón $u_i = \frac{C_i}{P_i}$ es llamada *factor de utilización de la tarea* τ_i, y representa la fracción tiempo de procesador utilizado por la tarea con tiempo de cómputo C_i y periodo P_i . El sumatorio de los factores de utilización de las tareas de un sistema es conocida como el *factor de utilización del procesador,* U_τ , y está dado por:

$$U_\tau = \sum_{i=1}^{n} \frac{C_i}{P_i}$$

Hasta el momento se han tratado los parámetros temporales de las tareas. Sin embargo, existen otro tipo de parámetros que también son de interés para el modelo del sistema y a los que se les conoce como *parámetros funcionales*. Uno de estos parámetros tiene que ver con la interrupción de la ejecución de una tarea. Se dice que una tarea es *expulsable*[10] si su ejecución puede ser interrumpida para permitir la ejecución de otra tarea. En contraste, una tarea es *no expulsable* si debe ejecutarse sin interrupción hasta completarse. En ocasiones, una tarea expulsable puede dejar de tener esa característica en ocasiones excepcionales, como cuando entra en una sección crítica o ejecuta un manejador de interrupciones. Esta pequeña porción de código no puede ser *expulsable* por que de otra manera se pone en riesgo la integridad de los datos compartidos por varias tareas.

[10]preemptable

Antes de que una tarea sea expulsada del procesador, el sistema debe almacenar su estado y cargar en memoria el estado de la nueva tarea. A estas acciones se les denomina *cambio de contexto*[11] y a la cantidad de tiempo requerida para llevarlo a cabo *sobrecarga de cambio de contexto*. El modelo de tareas periódicas supone que la sobrecarga de cambio de contexto es despreciable.

Otro parámetro funcional interesante está relacionado con la importancia que tiene la tarea en el sistema, ya que para el sistema no todas son necesariamente igual de importantes. Para diferenciarlas se utiliza un número entero positivo al que se le llama la *importancia de la tarea*. Entre mayor sea el número, mayor será la importancia. Los términos *prioridad* y *peso* son también utilizados para indicar la importancia de la tarea.

Una vez que se ha descrito el modelo de sistema, sus características y la terminología básica, se hará una revisión de algunas de las políticas de planificación más importantes.

2.3. Algoritmos de Planificación

Los algoritmos de planificación pueden ser clasificados en *estáticos* y *dinámicos* [20]. La planificación estática hace un cálculo previo del plan del sistema, requiere de conocimiento *a-priori* de las características de las tareas y tiene la ventaja de que se ejecuta con poca sobrecarga. La planificación dinámica, por otra parte, determina el plan en tiempo de ejecución permitiendo la implementación de sistemas más flexibles que cuando se utiliza la planificación estática. La sobrecarga atribuible a la planificación dinámica es mayor pero consiguen una mayor utilización del procesador.

En un planificador *estático* todas las decisiones de planificación se llevan a cabo previo a la ejecución del sistema. Una tabla es generada con las decisiones de planificación que serán utilizadas durante la ejecución. Esta tabla contiene el plan estático que indica el instante de tiempo en que cada tarea debe ser ejecutada, de tal manera que el planificador sólo debe seguir las indicaciones de la tabla. Tan sólo es necesario almacenar las actividades del hiperperiodo del conjunto de tareas ya que estas se repetirán durante la ejecución del sistema. Como puede verse esta clase de algoritmos dependen del conocimiento *a priori* de las características y comportamiento de las tareas. La verificación de la planificabilidad utilizando planificación estática debe llevarse a cabo durante la construcción del plan. Por otra parte, este esquema funciona si todas las tareas son periódicas, por lo que no puede utilizarse para sistemas conformados con tareas esporádicas o aperiódicas. El tamaño de la tabla es un inconveniente adicional ya que pudiera ser excesi-

[11] context switch

vamente grande. Además, están la escasa flexibilidad y el que el problema de construir el plan estático sea NP-completo [46], lo que hace que la planificación estática muestre muchas desventajas a pesar de que algunos autores la consideren ventajosa [114].

Un planificador *dinámico* toma las decisiones durante la ejecución del sistema tomando en consideración las características de las tareas y el estado del sistema. Los planificadores dinámicos más utilizados son los *basados en prioridades*, mientras que las prioridades de las tareas a su vez pueden ser asignadas de manera *fija* o *dinámica*. En la planificación dinámica se elige para ejecución a la tarea con función de prioridad mayor de entre las que estén activas. Cuando la prioridad es fija, su valor se mantendrá igual con respecto al resto de tareas durante la vida del sistema. En el caso de las prioridades dinámica, el valor de la función de prioridad podrá variar en relación con el resto de tareas dependiendo de la carga del sistema en un instante determinado.

La planificación dinámica ofrece muchas ventajas sobre la estática ya que es flexible y permite la implementación de sistemas más complejos.

2.3.1. Planificación dinámica con prioridades fijas

En el ámbito de la planificación dinámica con prioridades fijas dos algoritmos tienen particular relevancia: *prioridad a la tarea más frecuente* (*RM)*[12] y *prioridad a la tarea más urgente* (*DM*)[13]. El algoritmo RM asigna a cada tarea una prioridad inversamente proporcional a su periodo y asume que los plazos de las tareas son iguales a sus periodos. Para ejemplificar el funcionamiento del algoritmo RM, en la Figura 2.2 se muestra el cronograma de ejecución de un sistema formado por tres tareas, utilizando el algoritmo de *prioridad a la tarea más frecuente* (*RM)*. Las tareas tienen los siguientes parámetros: T_1 = (3, 1), T_2 = (5, 2) y T_3 = (8, 1), en donde el primer parámetro de las tareas representa su periodo y el segundo su tiempo de cómputo. Se supone que los plazos son iguales a los periodos y que todas las tareas son liberadas en el instante creo. De acuerdo a la política de planificación RM la tarea T_1 es la más prioritaria mientras que T_3 es la menos prioritaria. En el instante 5 es liberada la segunda instancia de T_2 y al no existir alguna otra tarea activa en el sistema inicia su ejecución. Sin embargo, en el instante 6 la tarea T_1 es liberada y al tener una prioridad mayor que T_2 expulsa a esta del procesador y T_1 inicia su ejecución. Una vez que T_1 concluye, T_2 continúa su ejecución en el punto en la que fue suspendida.

En [77] Liu y Layland demostraron que si se analizan las *fases* de las tareas de un conjunto dado, el peor caso ocurre cuando todas tienen fase igual a cero, esto

[12]Rate Monotonic
[13]Deadline Monotonic

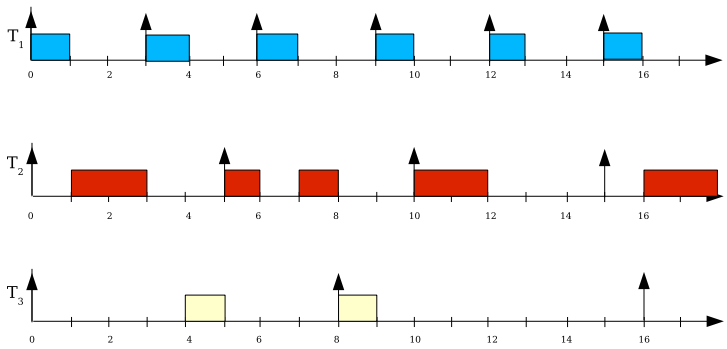

Figura 2.2: Ejemplo de un sistema planificado con el algoritmo RM

es, cuando el conjunto τ_i de tareas periódicas es *síncrono* ($\phi_i = 0 \ \forall 1 \leq i \leq n$). A este instante en el que todas las tareas son liberadas simultáneamente se le denomina el *instante crítico*. Se demostró también que un conjunto de tareas es planificable utilizando el algoritmo de *prioridad a la tarea más frecuente* si todas las primeras instancias de cada tarea cumplen con sus plazos si son liberadas en el instante crítico. El periodo de tiempo transcurrido entre el *instante crítico* y la ejecución de todas las tareas es un *periodo de ocupación*[14], ya que en él el procesador se mantiene ocupado procesando las tareas pendientes. Es importante destacar que un instante crítico pudiera existir no sólo al inicio del sistema.

Tomando como base el concepto de instante crítico, se demostró también el siguiente teorema:

Teorema 2.1

Un conjunto de tareas τ es planificable utilizando el algoritmo de prioridad a la tarea más frecuente si:

$$U_\tau \leq n\,(\,2^{\frac{1}{n}} - 1\,)$$

En el mismo artículo se mostró que la *utilización del sistema en el peor* caso está acotada en un rango que va desde $U_\tau = 0{,}83$ para *n* = 2, hasta $U_\tau = 0{,}693$ en la medida en que $n \rightarrow \infty$. Esto significa que la utilización máxima del procesador bajo el algoritmo de prioridad a la tarea más frecuente es del 83 % (n=2) y en el caso general puede decirse que es del 69 %.

[14]busy period

El test anterior se basa en el número de tareas del sistema. Sin embargo, adicionales tests de planificabilidad han sido propuestos posteriores a este basándose en otros parámetros, como los periodos o la utilización de las tareas. En [65] se mostró que un sistema en el que se generen de manera aleatoria tareas periódicas puede ser planificable si la utilización del procesador es menor o igual a 0,88, pero este hecho no puede ser garantizado para todos los conjuntos de tareas. En el mismo artículo se propuso el siguiente test:

Teorema 2.2

Para un conjunto de tareas τ_i *, ...,*τ_n*,*

La expresión

$W_i(t) = \sum_{j=i}^{i} C_j \left\lceil \frac{t}{P_j} \right\rceil$

proporciona el acumulado de demanda de procesador de las tareas en el periodo [0, t], cuando 0 es el instante crítico, y siendo

$L_i(t) = \frac{W_i(t)}{t}$,
$L_i = min_{(0<t\leq P_i)} L_i(t)$,
$L = max_{(1\leq i\leq n)} L_i$,

se tiene que:

1. τ_i *será planificable, para cualquier valor de fase, utilizando el algoritmo de prioridad a la tarea más frecuente, si y sólo si* $L_i \leq 1$.

2. *El conjunto de tareas* τ *será planificable, para cualquier valor de fase, utilizando el algoritmo de prioridad a la tarea más frecuente, si y sólo si* $L \leq 1$.

Puede observarse que la complejidad del tests anterior es pseudo-nominal. En [63] el test se extiende para el caso en que los plazos de las tareas sean menores o iguales a sus periodos.

En [19] se propone un test llamado *límite hiperbólico*[15] que extiende los anteriores incluyendo recursos compartidos y servidores aperiódicos.

El algoritmo estático de *prioridad a la tarea más urgente* asigna las prioridad de las tareas de acuerdo a sus plazos relativos, de tal forma que entre menor sea el plazo relativo mayor será la prioridad. Puede verse claramente que cuando los plazos relativos de las tareas son proporcionales a sus periodos, los algoritmos

[15]hyperbolic bound

DM y RM son idénticos. Cuando los plazos relativos son arbitrarios, el algoritmo DM muestra un rendimiento superior al RM ya que en ocasiones produce planes viables cuando el RM falla, mientras que el algoritmo RM falla en producir planes viables cuando el DM también falla.

En [68], Leung y Whitehead generalizaron los resultados proporcionados por Liu y Layland para el caso en que los plazos relativos de las tareas fueran menores o iguales a sus periodos y demostraron que el algoritmo DM es óptimo para el esquema de planificación basada en prioridades fijas, aunque no proporcionaron algún test de planificabilidad. Resultados similares fueron obtenidos también por Audsley en [11], quien propuso un test de planificabilidad para el algoritmo DM. Lehoczky, en [63], proporciona también un test de planificabilidad para el algoritmo de prioridad a la tarea más urgente.

2.3.2. Planificación dinámica con prioridades dinámicas

La planificación dinámica basada en prioridades dinámicas tiene dos algoritmos significativos: el de *prioridad a la tarea más urgente* (*EDF*)[16] y el de *prioridad a la tarea con menor holgura* (*LST*)[17]. El algoritmo EDF asigna las prioridades de las tareas de acuerdo a sus plazos absolutos, siendo mayor la prioridad mientras menor sea el plazo absoluto. Por otra parte, el algoritmo LST asigna la prioridad de las tareas de manera inversamente proporcional a la holgura de sus activaciones. En el instante de tiempo *t*, la *holgura* de una tarea se calcula restando del plazo *d* el tiempo de cómputo *x* que le queda por ejecutar (*holgura = d - t - x*).

La Figura 2.3 muestra un ejemplo de tres tareas: T_1 = (3, 1), T_2 = (4,2) y T_3 = (7, 1), planificadas utilizando la política EDF. El conjunto de tareas es síncrono y los plazos son iguales a los periodos.

Al inicio del sistema (instante cero) las tres tareas son liberadas. En ese momento se ejecuta la tarea T_1 debido a que es la de menor plazo y posteriormente, en el instante 1, se ejecuta la tarea T_2. En el instante de tiempo 4, la segunda instancia de la tarea T_2 es liberada. En ese momento en el sistema existen dos tareas activas: T_2 y T_3. De acuerdo al algoritmo EDF, la tarea T_3 es más prioritaria debido a que su plazo es menor que el de T_2 y por lo tanto es ejecutada. Por otra parte, en el instante 6 la tarea T_1 es de nuevo liberada pero no se ejecuta inmediatamente ya que T_2 tiene una prioridad mayor en ese momento. Como puede observarse, y a diferencia de los algoritmos basados en prioridades fijas, en los basados en prioridades dinámicas la prioridad de la tarea puede cambiar durante la ejecución del sistema. En el ejemplo de la Figura 2.3, en el instante cero la tarea T_1 es la más prioritaria pero en el instante 6 ya no lo es. Por otra parte, y para

[16] Earliest Deadline First
[17] Least Slack Time First

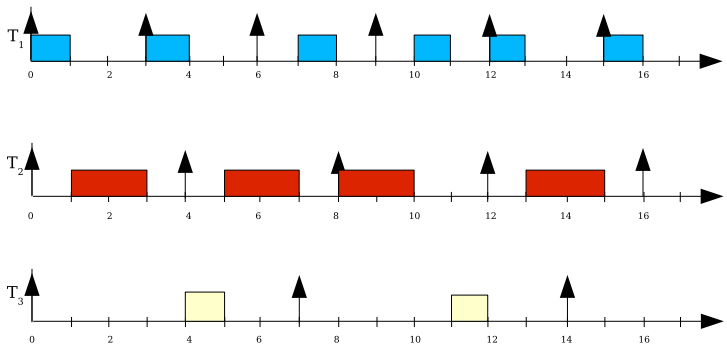

Figura 2.3: Ejemplo de tres tareas planificadas con la política EDF

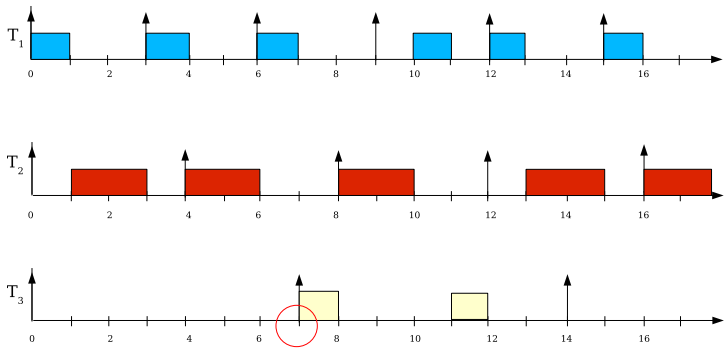

Figura 2.4: Conjunto de tareas de la Figura 2.3 planificadas utilizando el RM

ilustrar de mejor manera la diferencia entre el algoritmo de prioridades dinámicas EDF y el de prioridades fijas RM, en la Figura 2.4 se muestra el cronograma de ejecución del mismo conjunto de tareas utilizado en la Figura 2.3 pero planificado con el algoritmo RM. En el instante de tiempo 4 utilizando el algoritmo EDF la tarea más prioritaria es T_3 pues tiene un plazo menor, mientras que utilizando el algoritmo RM la tarea más prioritaria es T_2 ya que tiene un periodo menor que T_3. Utilizando el RM la tarea T_3 no puede cumplir su plazo al no ejecutarse antes del instante 7. Es interesante observar cómo el mismo conjunto de tareas es planificable utilizando el EDF y no lo es con la política RM.

Los algoritmos EDF y LST son óptimos [37] [81] para planificar tareas *expulsables* en un procesador, lo que significa que siempre que un conjunto de tareas pueda ser planificado de tal manera que cumplan siempre con sus plazos, entonces

los algoritmos EDF y LST generarán planes viables para ese conjunto de tareas. La ventaja que ofrece el EDF consiste en que no requiere conocer los tiempos de ejecución de las tareas, como si lo requiere el LST.

La primera referencia a la característica de *algoritmo óptimo* del EDF para un conjunto de tareas periódicas síncronas fue hecha por Liu y Layland en [77], quienes además demostraron que:

Teorema 2.3

Un conjunto de tareas periódicas síncronas es planificable con el algoritmo EDF si y sólo si:

$$u \leq 1$$

Esta condición es necesaria para todo algoritmo óptimo y es una condición suficiente de planificabilidad bajo el algoritmo EDF. Trabajos posteriores demostraron que el EDF es óptimo para conjuntos de tareas no síncronas [60], o para conjuntos de tareas con tiempos de liberación y plazos arbitrarios, y con tiempos de cómputo arbitrarios y desconocidos por el planificador [37].

Dos conceptos son particularmente importantes al analizar la factibilidad de un conjunto de tareas: la *demanda del procesador*[18] y el *factor de carga*[19] [105]. La demanda del procesador se centra en la cantidad de tiempo de cómputo requerido, con respecto a las restricciones temporales en un intervalo determinado de tiempo, mientras que el factor de utilización es la máxima fracción de tiempo de procesador *posiblemente* demandada por el conjunto de tareas en un intervalo de tiempo. Una gran cantidad de tests de planificabilidad han sido propuestos utilizando los conceptos mencionados, por lo que a continuación se presentas sus definiciones.

Para un conjunto de tareas y un intervalo de tiempo [t_1,t_2), la *demanda del procesador* del conjunto de tareas en el intervalo [t_1,t_2) es:

$$h_{[t_1,t_2)} = \sum_{t_2 \leq r_k, d_k \leq t_2} C_k$$

Para un conjunto de tareas su *factor de carga* en el intervalo [t_1,t_2) es la fracción del intervalo necesario para ejecutar sus tareas. esto es:

$$u_{[t_i,t_2)} = \frac{h_{(t_1,t_2)}}{t_2 - t_1}$$

[18] processor demand
[19] loading factor

Spuri, en [103], demostró el siguiente test de planificabilidad:

Teorema 2.4

Un conjunto de tareas de tiempo-real es planificable utilizando el algoritmo EDF si y sólo si

$$U_\tau \leq 1$$

Si bien es cierto que la condición anterior es necesaria para garantizar la planificabilidad de un conjunto de tareas bajo cualquier algoritmo, la importancia del teorema formulado por Spuri radica en el hecho de haber demostrado ser también una condición suficiente para la planificabilidad de cualquier conjunto de tareas bajo el EDF, a diferencia de lo demostrado por Liu y Layland y que era aplicable sólo para conjuntos de tareas periódicas síncronas., y a la extensión hecha por Coffman en [29] para conjuntos de tareas periódicas no síncronas. Por otra parte, Lung y Merril demostraron en [67] que las extensiones al modelo de tareas periódico, como el considerar conjuntos de tareas no síncronas o plazos menos o iguales a los periodos convierten al test de planificabilidad en NP-completo. Tests de planificabilidad más eficientes han sido propuestos por Barauh *et. al* [17] y Ripoll *et al.* [91].

2.3.3. Servicio de tareas aperiódicas

Hasta el momento se ha considerado principalmente que el sistema consta de un procesador y un conjunto de tareas periódicas. Sin embargo no todos los sistemas de tiempo-real pueden modelarse utilizando tan sólo tareas periódicas. En ocasiones algunas tareas se activan en respuesta a eventos externos o por que ocurre alguna situación anómala, eventos que no ocurren necesariamente en tasas de tiempo periódicas. Para representar de mejor manera a estos sistemas es necesario incluir a las tareas aperiódicas y esporádicas en el modelo de sistema . Se explicó que ambos tipos de tareas tienen tiempos de liberación irregulares y que la diferencia entre ellas consiste en que los tiempos de liberación de las tareas esporádicas están acotados por un mínimo tiempo de liberación entre activaciones sucesivas. Para poder analizar el modelo de sistema con tareas aperiódicas y esporádicas algunas restricciones tienen que hacerse, como por ejemplo al ancho de banda de tiempo de cómputo requerido por la carga aperiódica para que no ponga en riesgo el cumplimiento de los plazos de las tareas periódicas críticas. Una de los primeros en estudiar el tema fue Mok en [80], en donde presentó la noción de *tarea esporádica*.

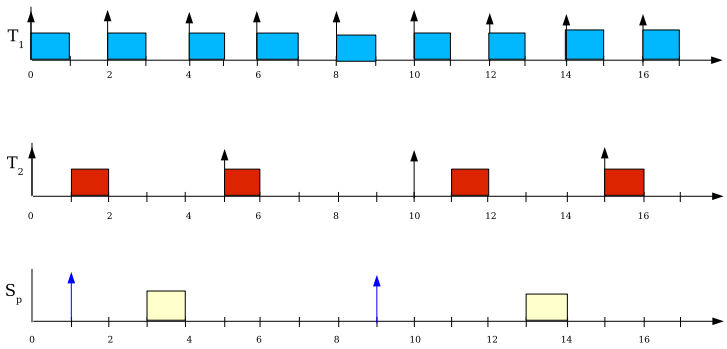

Figura 2.5: Ejemplo del uso de Ejecución en Segundo Plano

Debido a que los tiempos de liberación de las tareas *esporádicas* no son conocidos *a priori*, el análisis de planificabilidad se dificulta a menos que se limite a un particular conjunto de instancias de las tareas. Puede considerarse que el peor escenario ocurre cuando todas las tareas esporádicas son liberadas al mismo tiempo y con su tasa máxima de tiempos de liberación, de tal forma que se comportan como si fuesen un conjunto de tareas periódicas síncronas. En este caso, el conjunto de tareas esporádicas será planificable si y sólo si su equivalente conjunto de tareas periódicas síncronas es factible [105].

2.3.3.1. Ejecución en segundo plano

Para el caso de las tareas *aperiódicas* la gran mayoría de algoritmos de planificación buscan mejorar el desempeño del sistema basándose en tres enfoques diferentes: *ejecución en segundo plano*[20], *consulta*[21] o *basados en interrupciones*[22] [76]. De acuerdo a la *ejecución en segundo plano* las tareas aperiódicas son planificadas y ejecutadas únicamente cuando no hay tareas periódicas críticas por ejecutar. Esta técnica es fácil de implementar y produce siempre planes válidos pero retrasa los tiempos de respuesta de las tareas aperiódicas y del sistema en general. La Figura 2.5 muestra la ejecución de un sistema con dos tareas críticas síncronas planificadas bajo el algoritmo EDF, mientras que las tareas aperiódicas son planificadas con la *ejecución en segundo plano*. Las tareas críticas tienen las siguientes características: T_1 = (2, 1) y T_2 = (4, 1). En el instante 1 se libera una tarea aperiódica con tiempo de cómputo = 1. La tarea aperiódica inicia su

[20] background
[21] polling
[22] interrupt-driven

ejecución en el instante 5, ya que es hasta ese momento cuando no existen tareas críticas por ejecutar. De la misma manera, la segunda tarea aperiódica, liberada en el instante 9, es ejecutada hasta el instante 13. Más adelante se mostrará la ejecución del mismo conjunto de tareas utilizando otra política de planificación de tareas aperiódicas.

Para mejorar estos tiempos de respuesta puede utilizarse el enfoque *basado en interrupciones* y que consiste en interrumpir la ejecución de las tareas periódicas cuando alguna aperiódica es activada. Sin embargo, para evitar que las tareas críticas pierdan sus plazos es necesario cuidar que la ejecución de las tareas aperiódicas no ponga en riesgo a la correcta ejecución de las periódicas. Una manera de hacerlo consiste en ejecutar las tareas aperiódicas usando los tiempos de holgura de las tareas periódicas. A los algoritmos que hacen uso de los tiempos de holgura disponibles en el sistema se les conoce como *extractores de holgura*[23] y han sido propuestos por Chetto *et al.* [28], Lehoczky *et al.* [64] y Ripoll [89], entre otros, como se estudiará a continuación.

2.3.3.2. Algoritmos extractores de holgura

El *Algoritmo de Extracción de Holgura*[24] [64] [86] [87] es considerado óptimo ya que minimiza los tiempos de respuesta de las tareas aperiódicas. Fue propuesto como una mejora a los algoritmos de servidor diferido y servidor esporádico que se estudiarán más adelante. En lugar de crear un servidor periódico para atender a las tareas aperiódicas, en este algoritmo se crea una tarea pasiva llamada extractora de holgura. La tarea, cuando recibe alguna solicitud por parte de una tarea aperiódica, intenta extraer el tiempo de cómputo disponible y no utilizado por las tareas periódicas, sin poner el riesgo el cumplimiento de sus plazos. Para lograr una planificación óptima de las tareas aperiódicas, el algoritmo de extracción de holgura utiliza una tabla, calculada antes de la ejecución del sistema, en la que almacena el tiempo de holgura disponible por cada tarea crítica que se activa. Debido a que las liberaciones de las tareas siguen el mismo patrón en cada hiperperiodo, la tabla de holgura se calcula por cada liberación de las tareas aperiódicas sobre el mínimo común múltiplo de los periodos de las tareas. Esta característica restringe al algoritmo en gran medida ya que la holgura puede ser sólo *robada* o utilizada de tareas que tengan plazos estrictamente periódicos y que no sufran retrasos en sus tiempos de liberación[25]. Otra restricción es el que los tiempos de cómputo deben ser fiables ya que el cálculo de la holgura de cada tarea depende de la exactitud de su tiempo de cómputo.

[23] Slack Stealing
[24] Slack Stealing Algorithm
[25] no release time jitter

Muchas de las limitaciones del algoritmo de extracción de holgura fueron abordadas por Davis *et al.*, quien propuso una versión dinámica del algoritmo de extracción de holgura [32], y que calcula la tabla de holgura en tiempo de ejecución. Esta versión extiende el algoritmo estático para incluir características tales como sincronización, o conjuntos de tareas periódicas con retrasos en sus tiempos de liberación.

Ripoll *et al.* propusieron en [90] el *Algoritmo Extractor de Holgura Exacto EDF* (*EESS*), que está basado en el algoritmo EDF y que es óptimo al obtener los menores tiempos de respuesta para las tareas aperiódicas. El EESS asigna un plazo a cada tarea aperiódica para que sea planificada por el EDF como si fuese una tarea periódica. El plazo asignado a las tareas aperiódicas es el menor posible sin que ponga en riesgo la planificabilidad del sistema. Utiliza una tabla[26], calculada previo a la ejecución del sistema, para determinar los plazos de las tareas aperiódicas. El algoritmo reclama el tiempo de cómputo no utilizado, para poder obtener los menores tiempos de respuesta de la carga aperiódica.

Sin embargo, debido a la complejidad de los algoritmos extractores de holgura, es difícil su implementación.

2.3.3.3. Algoritmos basados en consulta

Los algoritmos basados en *consulta* utilizan un *servidor de consulta*[27] [98] que es una tarea periódica con parámetros $\{p_S, e_S\}$, en donde p_S representa el periodo del servidor y e_S su tiempo de cómputo. El servidor de consulta se ejecuta periódicamente y es planificado de acuerdo al algoritmo utilizado para planificar a las tareas periódicas. Cuando una tarea aperiódica es liberada se coloca en una cola ordenada bajo alguna política específica, como *Primera-en-Entrar-Primera-en-Salir* (*FIFO*)[28]. Cuando el servidor es ejecutado examina la cola de tareas aperiódicas y si está vacía suspende su ejecución inmediatamente. Pero si la cola no está vacía entonces pone en ejecución a la tarea que encabece la cola, que a su vez se ejecutara bajo la política de planificación de las tareas periódicas con periodo igual a p_S. El servidor suspenderá la ejecución de la tareas aperiódicas cuando se hayan ejecutado e_S unidades de tiempo en el periodo o cuando la cola se vacíe, lo que ocurra primero. A los servidores de consulta se les denomina también como *servidores periódicos*, por su naturaleza periódica. Otra forma de nombrarlos es como *servidores aperiódicos* ya que atienden la carga aperiódica de un sistema de tiempo-real.

Los servidores aperiódicos son definidos parcialmente por los parámetros $\{p_S$,

[26] look-up table
[27] Polling Server
[28] First-In-First-Out

$e_S\}$. El parámetro e_S es llamado *crédito de ejecución*[29] o simplemente *crédito*. A la razón $u_s = \frac{e_s}{p_s}$ se le conoce como *tamaño del servidor*. Se dice que el servidor *tiene tareas pendientes*[30] cuando en su cola existe al menos una tarea aperiódica. Por otra parte, el servidor está *inactivo*[31] cuando la cola está vacía. El servidor es *elegible* sólo si tiene tareas pendientes y su crédito es mayor que cero. Cuando el servidor se ejecuta consume su crédito a razón de una unidad de crédito por unidad de tiempo y cuando el crédito es igual a cero se dice que el servidor está *exhausto*. Se han propuesto diferentes tipos de servidores aperiódicos diferenciándose entre ellos en la manera en que el crédito cambia cuando el servidor está inactivo. Si un servidor puede conservar su crédito cuando encuentre vacía la cola de tareas aperiódicas se dice que utiliza un *algoritmo de servicio preservador del ancho de banda*[32]. Este tipo de servidores son definidos por un conjunto de reglas de *consumo*[33] y *reposición*[34]. A continuación se presentarán algunos de los *servidores preservadores del ancho de banda* más importantes.

2.3.3.4. Servidor Diferido

El *servidor diferido* (*DS*)[35] es el más sencillo de los servidores preservadores del ancho de banda. Cuando el servidor diferido se activa y encuentra vacía la cola de tareas aperiódicas, suspende su ejecución y conserva su crédito. Si al activarse existe al menos una tarea aperiódica en la cola estas son ejecutadas con periodo p_S de acuerdo al algoritmo de planificación utilizado por las tareas periódicas. Si no existen más tareas en la cola o el crédito se consume entonces el servidor suspende su ejecución hasta el siguiente periodo, en el que su crédito volverá a ser igual a e_S. No se permite que el servidor acumule su crédito de periodo en periodo, por lo que su máximo valor será siempre e_S.

El servidor diferido puede ser utilizado en conjunción con algoritmos estáticos y dinámicos. Para determinar la planificabilidad de un conjunto de tareas planificadas con el algoritmo de servidor diferido debe identificarse primero algún instante crítico para después llevar a cabo el usual análisis de demanda de tiempo. En el caso de utilizar un algoritmo de planificación estática basada en prioridades (RM) junto con el servidor diferido, Lehoczky *et al.* en [66], Strosnide *et al.* en [109] y Bini *et al.* en [19] han propuesto tests de planificabilidad.

Para el caso de la planificación dinámica, Ghazalie y Baker en [47] propusie-

[29] execution budget
[30] backlogged
[31] idle
[32] banwidth-preserving server algorithm
[33] consumption
[34] replenishment
[35] Deferrable Server

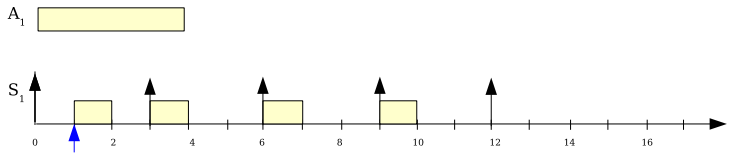

Figura 2.6: Ejemplo de un Servidor Diferido

ron un test de planificabilidad para un conjunto de tareas periódicas utilizando el algoritmo EDF junto con el servidor diferido, al que llamaron *servidor diferido de plazo*[36].

El servidor diferido tiene como mayor limitación el que retrasa por un tiempo superior a un periodo a las tareas de prioridad baja con el mismo periodo y tiempo de ejecución. En la Figura 2.6 se muestra un caso extremo para ejemplificar el funcionamiento del Servidor Diferido. El sistema contiene tan sólo tareas aperiódicas y un Servidor Diferido para atenderlas. El servidor está caracterizado por los parámetros (3, 1), en donde 3 representa su periodo y 1 su crédito. La tarea aperiódica A_1, con tiempo de cómputo = 4 unidades, es liberada en el instante de tiempo 1. Como no existe alguna otra tarea, A_1 y se ejecuta una unidad de tiempo desde el instante 1 hasta el instante de tiempo 2, en el que se consume totalmente el crédito del servidor. La tarea se suspende hasta el siguiente periodo, que inicia en el instante 3, en el que el servidor recarga su crédito al valor máximo y reanuda la ejecución de la tarea aperiódica. La tarea concluye su ejecución en el instante 10 a pesar de no existir alguna otra tarea en el sistema.

2.3.3.5. Servidor Esporádico

El *servidor esporádico* (*SS*)[37] [102] [47] fue propuesto para superar la limitación presente en el servidor diferido. Un conjunto de tareas compuesta por tareas periódicas y aperiódicas puede ser planificable utilizando el servidor esporádico aún cuando el mismo conjunto no es planificable utilizando el servidor diferido. La principal característica del servidor esporádico consiste en que su crédito no es *repuesto* a su máximo valor al inicio de cada periodo sino cuando ha sido consumido, lo que permite mejores tiempos de respuesta de las tareas aperiódicas al reducir la capacidad de procesador que se desperdicia. Los instantes de tiempo en los cuales la reposición del valor del crédito es llevada a cabo dependen de las reglas de reposición utilizadas, ya que existen varias versiones de servidores esporádicos, cada una con sus propias reglas de consumo y reemplazo. La propuesta original del servidor esporádico para planificación basada en prioridades

[36] Deadline Deferrable Server
[37] Sporadic Server

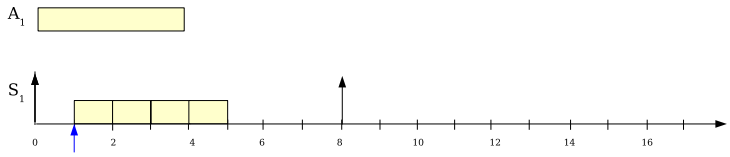

Figura 2.7: Ejemplo del Servidor Esporádico

estáticas hecha por Sprunt *et al.* [102] contenía un defecto, discutido en [76]. Para el caso de la planificación dinámica, las variantes del servidor esporádicas desarrolladas independientemente y conocidas como *servidor esporádico dinámico*[38] [104] y *servidor esporádico de plazo*[39] [47], que se diferencian del servidor esporádico clásico en la manera de asignar la prioridad del servidor y que permite la planificación de tareas aperiódicas en sistemas planificados por el EDF. Otra variante del servidor esporádico, adoptada bajo la política de planificación POSIX_SCHED_SPORADIC, ha sido propuesta por el Comité IPAS (POSIX) [54].

En la Figura 2.7 se muestra la ejecución del mismo ejemplo que en la Figura 2.6. En el instante de tiempo 1 la tarea aperiódica se ejecuta y en el instante 2 consume el crédito del servidor. Sin embargo, se calcula un nuevo periodo y se recarga el crédito inmediatamente, lo que permite que la tarea aperiódica siga ejecutándose al no existir más tareas activas en el sistema. Como puede observarse, las reglas de consumo y reemplazo del Servidor Esporádico permiten un mejor tiempo de respuesta de las tareas aperiódicas.

2.3.3.6. Servidor de Ancho de Banda Total

El *servidor de ancho de banda total* (*TBS*)[40] [104] es uno de los mecanismos más eficientes para la planificación de tareas aperiódicas utilizando el algoritmo EDF. El TBS asigna a cada tarea un plazo de tal manera que la carga total aperiódica nunca exceda el valor especificado por U_s, que representa el factor de utilización del servidor, o su ancho de banda. Cuando la $k_{ésima}$ solicitud de servicio aperiódico se activa en el tiempo $t = r_k$, se le asigna un plazo:

$$d_k \;=\; max\,(r_k,\; d_{k-1}) \;+\; \frac{C_k}{U_s}$$

en donde C_k es igual al tiempo de cómputo de la presente activación de la solicitud de trabajo aperiódico. Por definición, $d_0 = 0$. Una vez que el plazo ha sido asignado, la tarea aperiódica es insertada en la cola de tareas activas para ser planificada por el EDF. El test de planificabilidad se presenta en [104].

[38] Dynamic Sporadic Server
[39] Deadline Sporadic Server
[40] Total Bandwidth Server

2.3.3.7. Servidor de Ancho de Banda Constante

El *servidor de ancho de banda constante* (*CBS*)[41] fue propuesto por Abeni y Butazzo [2] para integrar servicios de reservación de recursos en sistemas planificados por el EDF. Un CBS se caracteriza por un crédito c_S, un plazo d_S y un par ordenado (Q_S, P_S), en donde Q_S representa el crédito máximo y P_S el periodo del servidor. La razón $U_s = \frac{Q_s}{P_s}$ es llamada el *ancho de banda* del servidor. Cuando una tarea aperiódica es liberada y atendida por el CBS, se le asigna un plazo igual al plazo actual del servidor, que se calcula de tal manera que conserve su demanda de acuerdo al ancho de banda reservada. Cuando el crédito es consumido y toma el valor de cero es recargado a su valor máximo Q_S y el plazo del servidor se incrementa en el valor del periodo P_S reduciendo así la interferencia de las tareas aperiódicas con las del resto del sistema. Al incrementar el plazo del servidor, se permite que siga siendo elegible y que la holgura disponible sea aprovechada. El CBS es definido formalmente de la siguiente manera:

1. Un CBS se caracteriza por un crédito c_S y un par ordenado (Q_S, P_S), en donde Q_S representa el crédito máximo y P_S el periodo del servidor. La razón $U_s = \frac{Q_s}{P_s}$ denota el ancho de banda del servidor. En cada instante existe un plazo fijo asociado con el servidor. Al inicio, $d_{s,\,0} = 0$.
2. A cada tarea atendida por el servidor se le asigna un plazo dinámico $d_{i,\,j}$ igual al plazo actual del servidor $d_{s,\,k}$.
3. Siempre que una tarea se ejecute, el crédito c_S se decrementa en la misma magnitud.
4. Cuando $c_S = 0$, el crédito del servidor es recargado en el valor máximo Q_S y un nuevo plazo es generado: $d_{s,\,k+1} = d_{s,\,k} + P_S$.
5. Se dice que un CBS está *activo* en el tiempo *t* si existen tareas pendientes, e *inactivo* si no existen tareas pendientes.
6. Si estando el servidor *activo* una tarea es liberada, se agrega a la cola de tareas pendientes del servidor de acuerdo a alguna política arbitraria no expulsiva, como *primero en llegar primero en salir*.
7. Si estando el servidor *inactivo* una tarea es liberada, y si $c_S \geq (d_{s,\,k} - r_{i,\,j})\,U_S$ se genera un nuevo plazo para el servidor $d_{s,\,k+1} = r_{i,\,j} + P_S$.
8. Cuando una tarea concluye su ejecución y si existe alguna tarea pendiente, se atiende utilizando los valores actuales de crédito y plazo del servidor. Si no existen tareas pendientes el servidor pasa a estado *inactivo*.

[41] Constant Bandwidth Server

9. En cualquier instante, a cada tarea se le asigna el último plazo generado por el servidor.

La Figura 2.8 muestra la ejecución de un sistema formado por dos tareas críticas síncronas: T_1 = (2, 1) y T_2 = (6, 1), con plazos iguales a sus periodos y planificadas con la política de planificación EDF. El sistema está formado además de un servidor de ancho de banda constante S_p = (1, 4), para planificar tareas aperiódicas. En el instante de tiempo cero las dos tareas críticas T_1 y T_2 y el servidor S_p son liberados. De acuerdo a la definición del servidor de ancho de banda constante, el plazo del servidor en el instante cero, $d_{\mathrm{s},\,0}$, es igual a cero, y como no existen tareas aperiódicas por atender su estado es *inactivo*. En el instante 1 se libera una tarea aperiódica con un tiempo de cómputo igual a 1. Debido a que el servidor está inactivo, al aplicarse la regla 7 de la definición del CBS se genera un nuevo plazo, $d_{\mathrm{s},\,1}$. En el instante de tiempo 1 existen dos tareas activas en el sistema: T_1 y la tarea aperiódica, que se planifica también bajo el EDF pero con los parámetros del servidor, a saber, $d_{\mathrm{s},\,1}$ = 5 y c_s = 1. Debido a que el plazo de la tarea aperiódica (5) es menor que el de T_2 (6), en el instante 1 se ejecuta la tarea aperiódica. En el instante 2 se consume por completo el crédito del servidor e inmediatamente se recarga a su valor máximo, y se calcula un nuevo plazo $d_{\mathrm{s},\,2}$ para el servidor de acuerdo a lo establecido en la regla 4 de la definición. Como puede observarse en el ejemplo, la utilización del CBS permite un buen tiempo de respuesta de la carga aperiódica sin poner en riesgo la ejecución de las tareas críticas del sistema.

En un intervalo cualquiera de tiempo *L*, el CBS no demandará nunca un ancho de banda superior a $U_\mathrm{s}L$, independientemente de la solicitudes de trabajo aperiódico existentes. Esta propiedad permite que el CBS sea utilizado como una estrategia de reservación de ancho de banda para asignar una fracción de tiempo de procesador a las tareas aperiódicas cuyo tiempo de cómputo pueda ser fácilmente acotado. De esta manera, las tareas aperiódicas pueden ser planificadas junto con tareas críticas sin afectar su garantía *a priori* de planificabilidad, aún en el caso de que la carga aperiódica sea superior a la esperada.

2.3.3.8. Algoritmos de reclamo[42]

Los servidores TBS y CBS presentan el problema de determinar con exactitud el tamaño óptimo de su ancho de banda. Si es menor que el valor promedio solicitado las tareas aperiódicas tendrán un pobre tiempo de respuesta. Por otra parte, si el valor del ancho de banda es mayor que el necesario, el sistema se ejecutará de manera lenta y desperdiciará recursos. Una forma de resolver el problema de la incorrecta determinación del ancho de banda es proporcionada por las técnicas de

[42] reclaiming algorithms

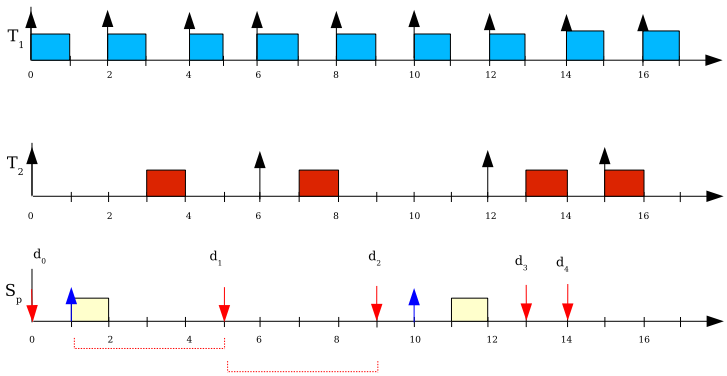

Figura 2.8: Ejemplo de un Servidor de Ancho de Banda Constante

reclamación. En esta sección se estudiarán algunos de los algoritmos de reclamo más importantes propuestos a la fecha.

CASH y GRUB

Una de las formas propuestas para resolver el problema de la incorrecta determinación del ancho de banda es conocida como *algoritmo de participación de capacidad* (*CASH*)[43] [22]. El algoritmo CASH fue definido para trabajar con el CBS y funciona de la manera que a continuación se describe. Cuando una tarea termina su ejecución, si existe alguna capacidad residual, es insertada con su plazo en una cola global de capacidades disponibles, la cola CASH, que está ordenada por plazos. Cuando una tarea es ejecutada, el servidor intenta usar las capacidades residuales con sus plazos si son menores o iguales al asignado por el servidor a la tarea actual. Una vez que la capacidad residual ha sido consumida y si la tarea no ha concluido su ejecución, entonces el servidor comienza a utilizar su propia capacidad. El principal beneficio de esta técnica radica en que reduce el número de cambios de plazos en el CBS, mejorando el tiempo de respuesta de las tareas aperiódicas.

Otro algoritmo similar es el de *reclamación ávida del ancho de banda no usado* (*GRUB*)[44] [72], en el cuál cada tarea aperiódica es ejecutada en un servidor S_{i} distinto, cada uno con participación del procesador U_{i} y periodo P_{i}. El algoritmo GRUB es capaz de emular un procesador virtual de capacidad U_i cuando ejecuta una tarea aperiódica y utiliza la noción de tiempo virtual para forzar el aislamiento entre las diferentes aplicaciones y para reclamar de manera segura las capacidades

[43] CApacity SHaring
[44] Greedy Reclamation of Unused Bandwidth

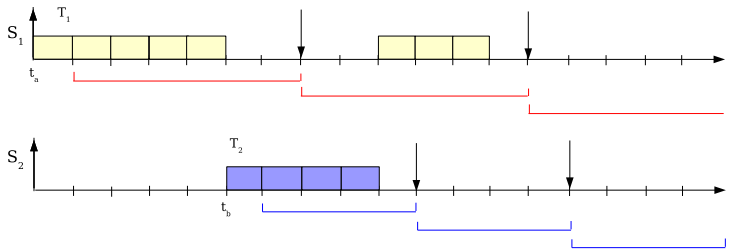

Figura 2.9: Problema del CBS

sin utilizar generadas por las cargas periódicas y aperiódicas. Comparado con el CASH, el algoritmo GRUB tiene una mayor sobrecarga pero un mejor desempeño.

IRIS

Un algoritmo que mejora al algoritmo GRUB con una menor sobrecarga que el CASH es el algoritmo IRIS[45] y fue propuesto por Marzario *et al.* en [78]. Una de las propiedades más interesantes del IRIS es que evita el *envejecimiento del plazo*[46], aprovechando de mejor manera la capacidad disponible con una baja sobrecarga. Para ilustrar este problema, en la Figura 2.9 se muestra la ejecución de dos tareas aperiódicas T_1 y T_2, atendidas por los servidores S_i y S_2 respectivamente. En el instante de tiempo t_a la tarea T_1 es liberada, y debido a que es la única tarea presente en el sistema, inicia su ejecución sin ser interrumpida. El crédito del servidor se consume y recarga constantemente, provocando que el plazo del servidor S_i se incremente en cada recarga, extendiéndose de manera considerable en el tiempo. En el instante t_b se libera la tarea T_2, y debido a que el plazo del servidor S_2 es menor que el de S_i, la tarea T_2 inicia su ejecución. Cuando el crédito de S_2 se consume es inmediatamente recargado y su plazo incrementado. Sin embargo, debido a que el plazo de S_i es mucho mayor por los constantes incrementos sufridos, T_2 continúa su ejecución inmediatamente, sólo hasta que el plazo de S_2 se hace mayor que el de S_i, la tarea T_1 puede ejecutarse. Debido a este problema, los servidores no ejecutan Q_i por cada uno de sus periodos P_i. Usando el algoritmo CSB puede ocurrir que los valores Q_i y P_i con los que realmente es planificado cada servidor sean diferentes a los inicialmente definidos, y que dependan de los parámetros de el resto de servidores presentes en el sistema. El algoritmo IRIS resuelve estos problemas.

El algoritmo IRIS, que se basa en el CBS, puede ser definido de la siguiente manera:

[45] Idle-time Reclaiming Improved Server
[46] deadline aging

1. Cada servidor S_i está asociado a una tarea τ_i y está caracterizado por un par ordenado (Q_i, P_i), en donde Q_i representa el *crédito máximo* y P_i el *plazo relativo*,. La suma de los anchos de banda de los servidores, $\frac{Q_i}{P_i}$, no podrá ser mayor a uno: $\sum_{i=1}^{n} \frac{Q_i}{P_i} \leq 1$.

2. Cada servidor S_i mantiene:

   a) un *crédito actual* q_i, que se decrementa cuando cada tarea *atendida por el servidor* se ejecuta;

   b) un *plazo del servidor* d_i, que es utilizado al insertar al servidor en la cola EDF del planificador del sistema;

   c) un *tiempo de recarga* r_i, que es usado para establecer el tiempo en que el servidor recargará su crédito actual cuando haya sido consumido en su totalidad.

3. Cada servidor puede estar en alguno de los siguientes estados:

   - *Activo*: la tarea atendida por el servidor está lista para ser ejecutada o está en ejecución y el crédito es mayor que 0.
   - *Recarga*: la tarea atendida por el servidor está lista para ser ejecutada pero el servidor está en espera de que se recargue el crédito que se ha consumido en su totalidad.
   - *Inactivo*: no existen tareas por ser atendidas

4. El sistema mantiene:

   - una cola[47] en la que todos los servidores *Activos* están ordenados por plazo.
   - una cola[48] en la que todos los servidores en estado de *Recarga* están ordenados por tiempo de recarga.

5. Cuando el sistema inicia todos los servidores están en estado de *Inactivos*. El crédito actual y el plazo relativo asociado con cada servidor S_i se actualizan aplicando las siguientes reglas:

   - Si una tarea $j_{i,k}$ se libera en el tiempo *t*,
     - si el servidor está *Inactivo*,

[47] ready queue
[48] recharging queue

    - si $t \geq d_i - q_i \frac{P_i}{Q_i}$, entonces $q_i = Q_i$ y $d_i = t + P_i$;
    - si $t \leq d_i$ y $q_i = 0$, el servidor pasa a estado de *Recarga* y es insertado en la *cola de servidores en recarga* con $r_i = d_i$;
    - de otra manera, el servidor pasa a estado *Activo* y es insertado de nuevo en la *cola de servidores activos* del sistema con el mismo crédito y plazo.
  - si el servidor está *Activo* o en *Recarga*, la tarea se almacena en espera de ser atendida posteriormente.
- Cuando una tarea atendida por el servidor se ejecuta por δ, $q_i = q_i - \delta$.
- Si el servidor está *Activo* y y $q_i = 0$, el servidor pasa a el estado de *Recarga* y el tiempo de recarga es $r_i = d_i$.
- Si el servidor está en estado de *Recarga* y $t = r_i$, entonces el servidor pasa a estado *Activo*, $q_i = Q_i$ y $d_i = d_i + P_i$.
- Cuando la tarea concluye su ejecución,
  - si existe alguna otra tarea pendiente, el servidor continúa en estado *Activo*;
  - de otra manera, pasa a estado *Inactivo*.
- Si en el tiempo *t* no existen servidores *Activos* y existe al menos un servidor en estado de *Recarga*,
  - siendo S_i el servidor que encabece la *cola de servidores en recarga* (el que tiene el menor tiempo de recarga), y siendo $\delta = r_i - t$; para cada servidor *i* en la *cola de servidores en recarga*, $r_i = r_i - \delta$;
  - cada servidor S_i en la *cola de servidores en recarga* con $r_i = t$ es eliminado de la *cola de servidores en recarga* e insertado en la cola de servidores en activos; su crédito es recargado al valor máximo $q_i = Q_i$ y su plazo se hace igual a $d_i = t + P_i$.

Las dos principales diferencias entre los algoritmos IRIS y CBS son las siguientes. Por una parte, el algoritmo IRIS establece explícitamente el tiempo de recarga r_i de cada servidor, mientras que el CBS inmediatamente recarga el crédito del servidor cuando se ha consumido. A esta característica se le denomina *técnica de reservación estricta*[49] [83], que consiste en suspender la ejecución de la tarea aperiódica hasta el siguiente tiempo de recarga si el crédito del servidor se ha consumido. Por otra parte, la segunda diferencia consiste en que el algoritmo IRIS contiene una regla para adelantar los tiempos de recarga de todos los servidores

[49] hard reservation technique

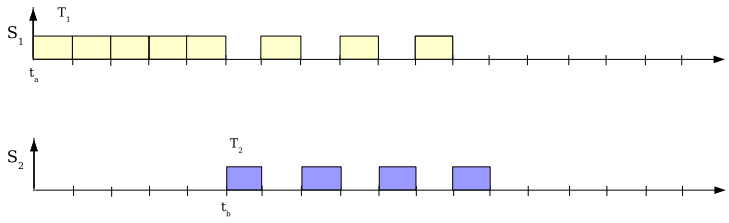

Figura 2.10: Ejemplo del algoritmo IRIS

que se encuentren en estado de *recarga*, cuando ocurre algún instante de tiempo en el que ninguna tarea se ejecuta[50]. Con esta modificación el algoritmo IRIS *reclama* eficientemente el tiempo no usado y lo distribuye entre los servidores que lo requieran. La Figura 2.10 muestra la ejecución del conjunto de tareas de la Figura 2.9, pero planificado utilizando el algoritmo IRIS. Como puede observarse, se obtiene una mejor respuesta de las tareas aperiódicas y se evita el problema de envejecimiento del plazo.

HisReWri

Bernat *et al.* propusieron en [18] un esquema llamado *Reescritura de Historia* o *HisReWri*, para aprovechar el tiempo de cómputo no utilizado, llamado también *tiempo de ganancia*[51] [10], en beneficio de otras tareas del sistema que pudiesen requerirlo. El esquema tiene como objetivo el que todo tiempo de ganancia generado a un nivel de prioridad P_{i} pueda utilizarse en niveles de prioridad cercanos a P_{i} y no en el segundo plano, mejorando así el tiempo de respuesta del sistema. El esquema propuesto se utiliza en sistemas basados en prioridades fijas, y cada tarea es planificada utilizando un *servidor diferido*. El periodo de cada servidor es igual al periodo de la tarea que planifique, mientras que su crédito es igual al *tiempo de ejecución en el peor caso* de la tarea.

HisReWri hace uso del *tiempo disponible para ser compartido*[52] propuesto en [22] pero de manera diferente, ya que el tiempo no utilizado por una tarea τ_i es utilizado en beneficio de otra tarea τ_j , pero sin afectar el crédito de esta última. Además, el algoritmo revisa la historia reciente de ejecución de las tareas, de tal manera que el tiempo disponible pueda ser asignado a tareas que se ejecuten previamente a la tarea donante, y sin que el tiempo de ejecución sea descontado del crédito de la tarea que utilizó el tiempo de holgura. De esta manera se evitan interferencias, ya que el tiempo de holgura se utiliza en beneficio de la tarea que

[50] idle time
[51] gain time
[52] capacity sharing

lo necesite, pero ejecutándose al nivel de prioridad y con el crédito de la tarea donante. Así, la tarea beneficiada podrá disponer de suficiente tiempo de cómputo para ejecutarse.

BACKSLASH

En [69], Lin y Brandt presentan cuatro principios generales para una mejor utilización de la holgura, y con base en ellos definen cuatro algoritmos, siendo el BACKSLASH el que cumple con los cuatro principios descritos, y de acuerdo a los resultados obtenidos y presentados en su artículo, muestra una menor tasa de pérdidas de plazo en las tareas aperiódicas, comparado con la mayoría de los algoritmos estudiados hasta el momento. Puede decirse que el BACKSLASH es una versión mejorada del HiReWri, y que ha sido diseñado para trabajar con sistemas basados en prioridades dinámicas, particularmente con el EDF.

Los principios generales para una mejor utilización de la holgura son los siguientes:

1. Asignar el tiempo de holgura disponible tan pronto como sea posible, con la prioridad de la tarea donante.

2. Asignar la holgura a la tarea que tenga la más alta prioridad (la de menor plazo relativo, para el EDF).

3. Permitir que las tareas puedan utilizar holgura de sus futuras activaciones , pero con la prioridad de la activación actual.

4. Asignar tiempo de holgura de manera retroactiva, a tareas que han usado previamente holgura de sus propias activaciones futuras. Este principio permite que una tarea pueda utilizar tiempo de holgura de otras tareas pero sin que su crédito sea disminuido, ya que se considerará el crédito de la tarea donante. Además, como en el HiReWri, es posible utilizar tiempo de holgura de tareas que aún no inician su ejecución y que han tenido tiempo de holgura en activaciones anteriores, para lo cual es necesario mantener la *historia* de ejecución de las tareas.

En el algoritmo BACKSLAH cada tarea es planificada utilizando un *servidor basado en tasas*[53], que es conceptualmente similar al CBS. Para las tareas críticas, el crédito de su servidor será igual a su *tiempo de ejecución en el peor caso*, mientras que para las tareas no críticas el crédito será la media de sus tiempos de ejecución. Además, de manera similar al CBS, a cada servidor se le asigna un plazo *dinámico*, y las reglas de recarga y consumo implementan los principios arriba descritos.

[53] rate-based server

2.3.4. Recursos Compartidos

En la sección anterior se extendió el modelo de tareas aperiódicas críticas para incluir tareas esporádicas y aperiódicas. En esta sección se extenderá el modelo para considerar la existencia de recursos compartidos.

Además de un procesador y de un conjunto de tareas, un sistema de tiempo-real puede incluir también un conjunto de *m* recursos:

$$\rho = \{\rho_i, \text{ i} \in [1..\text{m}]\}$$

Los recursos del sistema son asignados a las tareas de forma no expulsiva y usados de manera mutuamente exclusiva. Esto significa que cuando un recurso es asignado a alguna tarea ninguna otra podrá utilizar ese recurso hasta que sea liberado. Ejemplos de recursos son los monitores[54], bloqueos de lectura/escritura[55], sockets de conexión, impresoras y servidores remotos. De cada recurso pueden existir ν_i unidades.

Cuando una tarea necesita utilizar algún recurso ejecuta una operación de *bloqueo*[56] para solicitarlo. La tarea continúa su ejecución cuando le es otorgado y una vez que ha concluido las operaciones sobre el recurso, lo libera con una operación de *desbloqueo*[57]. Al segmento que inicia con el bloqueo y termina con el desbloqueo del recurso de le denomina *sección crítica*. Se dice que dos tareas *están en conflicto* o que tienen *conflictos de recursos* cuando ambas requieren el mismo recurso. Dos tareas están *en contención* cuando una de ellas solicita un recurso que ha sido asignado a la otra. El planificador siempre negará un recurso si no existen suficientes unidades disponibles del mismo para satisfacer la solicitud.

Si el planificador niega un recurso entonces la operación de bloqueo falla, lo que provoca que la tareas se *bloquee* y pierda el procesador, y permanecerá en ese estado hasta que el recurso le sea asignado, en cuyo caso la tarea es *desbloqueada*.

Un *protocolo de control de acceso a los recursos*[58] es un conjunto de reglas que determinan cuándo y en que condiciones cada solicitud de recursos es satisfecha, así como la manera en que las tareas que solicitan recursos son planificadas [76]. Los protocolos de acceso a los recursos se hacen necesarios ya que una incorrecta asignación de los mismos puede provocar anomalías en el sistema, provocados principalmente por no acotar o controlar el tiempo de bloqueo durante la exclusión mutua. Dos de los problemas más comunes son los de *inversión de la prioridad*[59], identificado por Cornhill y Sha en [30], y los *interbloqueos*[60].

[54] mutexes
[55] reader/writer locks
[56] lock
[57] unlock
[58] resource access-control protocol
[59] priority inversion
[60] deadlocks

El problema de inversión de la prioridad está relacionado con la naturaleza *expulsiva* de algunas políticas de planificación. En sistemas que utilicen un mecanismo de planificación expulsivo y basado en prioridades, una tarea activa debe poder expulsar a otra e iniciar su ejecución inmediatamente, si tiene una mayor prioridad. Sin embargo, cuando se utilizan recursos compartidos, lo anterior no siempre ocurre ya que una tarea de prioridad baja que se encuentre en una sección crítica no podrá ser expulsada por una tarea de prioridad mayor mientras permanezca en la sección crítica, si la tarea más prioritaria se bloquea al intentar acceder a la misma sección crítica. La tarea más prioritaria se ve obligada a esperar a que la tarea de menor prioridad libere el recurso. El principal problema radica en que el tiempo en que la tarea prioritaria permanezca bloqueada debe de ser considerado en el análisis de planificabilidad, ya que si el tiempo es muy grande o no acotado puede afectar a la planificabilidad del sistema. Para ilustrar el problema de la inversión de la prioridad, en la Figura 2.11 se muestra la ejecución de un sistema formado por tres tareas, T_1, T_2 y T_3, planificadas bajo el algoritmo EDF. Las tareas T_1 y T_3 comparten el mismo recurso, cuyo acceso es controlado por una sección crítica. En el instante 1 la tarea T_3 es liberada y al ser la única tarea activa en el sistema, se ejecuta inmediatamente. En el instante 3, la tarea T_3 entra en la sección crítica. Las tareas T_1 y T_2 son liberadas en el instante 4 y al ser T_1 la tarea de mayor prioridad en ese momento, inicia su ejecución expulsando a T_3 del procesador. Sin embargo, en el instante 6, T_1 intenta acceder al recurso compartido, pero debido a que T_3 está dentro de la sección crítica, T_1 se bloquea y libera el procesador. En ese momento, las tareas activas no bloqueadas son T_2 y T_3, y como T_2 tiene mayor prioridad, se ejecuta. Como puede observarse, la tarea más prioritaria, en este caso T_1, debe esperar a que concluya la ejecución de una tarea de prioridad media T_2. El tiempo durante el cual la tarea más prioritaria permanecerá bloqueada dependerá del número de tareas de prioridad media activas en el sistema en ese momento, lo que provoca que el tiempo de bloqueo no sea controlado. Siguiendo con el ejemplo, la tarea T_2 se ejecuta sin interrupción hasta el instante 8, en el que concluye su ejecución. Es hasta entonces que T_3 puede reanudar la suya, concluyéndola hasta el instante 9, que es cuando libera el recurso. En ese momento, T_1 puede por fin entrar en la sección crítica y ejecutarse, incumpliendo su plazo debido al largo bloqueo provocado por la inversión de prioridad.

A continuación se describirán algunos de los protocolos de acceso a los recursos, y estudiaremos la manera en que evitan o controlan los problemas de inversión de la prioridad y los interbloqueos.

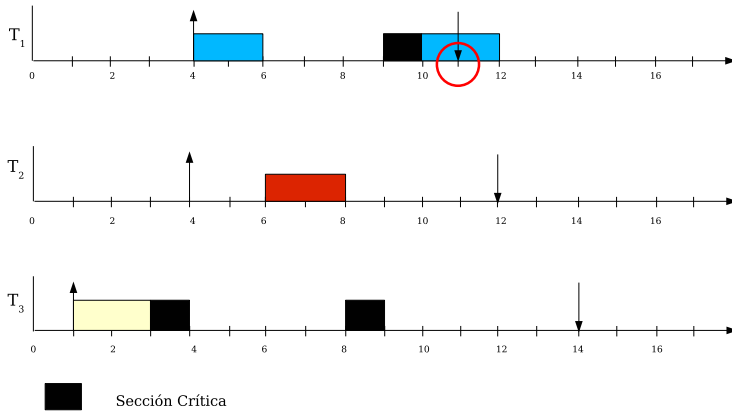

Figura 2.11: Inversión de Prioridad bajo el EDF

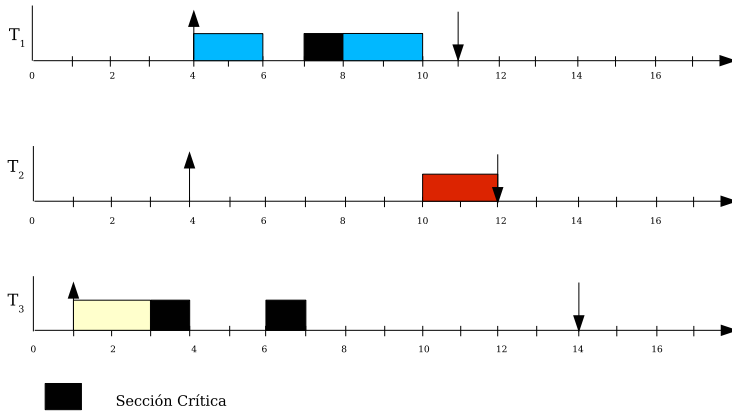

Figura 2.12: Ejemplo de Herencia de Prioridades bajo el EDF

2.3.4.1. Protocolo de Herencia de Prioridades

El *protocolo de herencia de prioridades* (*PIP*)[61] fue propuesto por Sha *et al.* en [100] y es el más sencillo de los protocolos de acceso a los recursos. Funciona con cualquier algoritmo expulsivo basado en prioridades, no requiere conocimiento previo de las requerimientos de recursos de las tareas, no previene los interbloqueos y si no ocurren interbloqueos el PIP garantiza que ninguna tarea es bloqueada de manera indefinida debido a que una incontrolada inversión de prioridad no podrá ocurrir.

Cuando se utiliza este protocolo, se le llama *prioridad asignada* a la prioridad que se le asigna a cada tarea de acuerdo al algoritmo de planificación utilizado (RM, EDF, etc), y *prioridad actual* $\pi_l(t)$ a la prioridad con la cual la tarea se ejecuta. La prioridad actual de una tarea τ_l puede ser cambiada a una prioridad mayor π_h de una tarea τ_h.

El protocolo PIP funciona de la siguiente manera. Todas las tareas periódicas críticas del sistema son planificadas de acuerdo a la política de planificación basada en prioridades seleccionada. Cuando una tarea τ_i es liberada, su prioridad actual $\pi_i(t)$ es igual a su prioridad asignada. Si una tarea solicita algún recurso que se encuentre disponible, le es asignado. Pero si el recurso solicitado ha sido asignado previamente a otra tarea, la que lo solicitó se bloquea. Si la tarea a la que se le asignó el recurso tiene una prioridad actual $\pi_i(t)$ menor a la tarea $\pi_h(t)$ que lo solicitó y está bloqueada, entonces la tarea que tiene el recurso *hereda* la prioridad actual $\pi_h(t)$ de la tarea bloqueada. Esto provoca que la tarea que usa el recurso se ejecute a una prioridad mayor asegurando que la duración de la inversión de prioridad nunca es mayor que la mayor sección crítica de la tarea bloqueada.

Stankovic *et al.* en [105]. propusieron una versión del PIP para utilizarse con prioridades dinámicas, particularmente bajo el EDF, llamada *Herencia de Prioridades Dinámicas* (*DPIP*)[62].

En la Figura 2.12 se muestra la planificación del sistema que se presentó previamente en la Figura 2.11, pero planificada utilizando el algoritmo de herencia de prioridades bajo el EDF. En el instante de tiempo 2, la tarea T_3 entra en la sección crítica, y es expulsada del procesador por la tarea T_1 en el instante 4. En el instante 6, la tarea T_1 intenta entrar en la sección crítica pero no puede hacerlo ya que T_3 aún está en ella, por lo que libera el procesador y se bloquea. Sin embargo, en ese momento T_3 *hereda* la prioridad de T_1 y puede ejecutarse antes que la tarea T_2. Cuando T_3 sale de la sección crítica en el instante 7, *deshereda* la prioridad de T_1. En el instante 7, T_1 puede entrar en la sección crítica y ejecutarse por ser la tarea activa no bloqueada más prioritaria en ese momento. En el instante 10, T_2 inicia su ejecución.

[61] Priority-Inheritance Protocol

[62] Dynamic Priority Inheritance Protocol

Como puede observarse, el PIP no evita que la tarea más prioritaria sea bloqueada por una tarea de prioridad menor, pero si garantiza que la duración del bloqueo no será mayor a una sección crítica.

De las desventajas más importantes del PIP destaca el hecho de que no reduzca los tiempos de bloqueo de las tareas al mínimo posible.

2.3.4.2. Protocolo de Techo de Prioridades

El *protocolo de techo de prioridades* (*PCP*)[63] fue propuesto por Sha *et al.* en [99] [100] como una extensión del PIP para prevenir los interbloqueos y reducir los tiempos en los que las tareas permanecen bloqueadas. El PCP hace dos importantes suposiciones: que las prioridades de las tareas son fijas y que se conocen *a priori* los recursos que utilizarán las tareas. Un nuevo parámetro es definido, el *techo de prioridad* de cada recurso ρ_i, representado por $\Pi(\rho_i)$ y que es igual a la mayor prioridad de las tareas que usan el recurso. En cualquier instante, el *techo de prioridad del sistema* Π es igual al mayor de los techos de prioridad de los recursos que estén en uso en ese instante. Si todos los recursos están disponibles, el techo de prioridad es Ω, una prioridad inexistente inferior a la menor prioridad de todas las tareas.

La prioridad actual de toda tarea τ_i cuando es liberada es igual a su prioridad asignada. Si alguna tarea solicita un recurso que no está disponible, se bloquea. Si el recurso está disponible, se le asigna si la prioridad actual de la tarea es mayor que el techo de prioridad del sistema. Por otra parte, si la prioridad $\pi_i(t)$ no es mayor que el techo del sistema, se le asignará el recurso a la tarea τ_i si y sólo si la tarea τ_i tiene el recurso cuyo techo de prioridad es igual al techo del sistema, por que de otra manera τ_i se bloquea.

El protocolo de *techo de prioridades dinámico* (*DPCP*)[64] fue propuesto por Chen y Lin en [26] para adaptar el PCP cuando se utiliza al EDF para la planificación de tareas periódicas críticas. Por otra parte, Jeffay propuso en [57] el algoritmo de *modificación dinámica del plazo* (*DDM*)[65] como una alternativa más sencilla al DPCP y que además contempla la compartición de recursos con tareas esporádicas.

Los protocolos PIP y PCP se diferencian en que el primero es del tipo ávido[66] mientras que el segundo no lo es. En el PIP todo recurso libre es asignado a la tarea que lo solicite, a diferencia del PCP. Esta diferencia permite que el PCP evite los interbloqueos y que reduzca el tiempo en que una tarea se bloquea, que es cuando mucho, equivalente a la duración de una sección crítica [100].

[63] Priority-Ceiling Protocol
[64] Dynamic Priority Ceiling Protocol
[65] Dynamic Deadline Modification
[66] greedy

Debido a que el protocolo PCP, tal y como fue originalmente definido por Sha, Lehoczky y Rajkumar es complejo de implementar, se ha definido una versión simplificada llamada *emulación del protocolo del techo de prioridades* o *protocolo de la mayor cerradura*[67] [58], adoptad por el Comité IPAS (POSIX) como la política de sincronización POSIX_PRIO_PROTECT [54].

2.3.4.3. Protocolo de Recursos de Pila

El *protocolo de recursos de pila* (*SRP*)[68] fue propuesto por Baker [12] para acotar el fenómeno de inversión de la prioridad tanto en sistemas que utilicen prioridades fijas como dinámicas. Baker detectó que bajo el EDF las tareas con plazos relativos muy grandes pueden retrasarse pero no pueden expulsar a tareas con plazos relativos pequeños. Una consecuencia de esta observación es que una tarea no podrá nunca bloquear a otra con plazo relativo mayor. Por lo tanto, al estudiar los bloqueos bajo el EDF es tan sólo necesario considerar los casos en los que las tareas con plazos grandes bloquean a tareas con plazos relativos cortos. Para utilizar el SRP es necesario definir los *niveles de expulsión*[69] de las tareas de manera que estén relacionados con sus prioridades pero sin ser exactamente iguales. Lo importante es que a mayor prioridad mayor sea el nivel de expulsión. Por ejemplo, para el EDF, una forma de definir los niveles de expulsión sería de manera inversamente proporcional a los plazos relativos de las tareas. Los recursos tienen también su propio nivel de expulsión, que es definido de manera similar a como se definen los techos de prioridad de los recursos en el PCP. Si se utiliza el SRP en conjunción con el EDF, a cada tarea τ_i se le asigna una prioridad dinámica p_i de acuerdo a su plazo absoluto d_i y un nivel de expulsión estático π_i inversamente proporcional a su plazo relativo. A cada recurso se le asigna un techo que es igual al máximo nivel de expulsión de las tareas que utilizan el recurso. Existe también un techo del sistema definido como similar al mayor de los techos de los recursos utilizados en un instante dado. Bajo el SRP, no se permite que una tarea inicie su ejecución hasta que su prioridad sea la mayor de entre las tareas activas y que su nivel de expulsión sea mayor que el techo del sistema. Con esta condición se garantiza que una vez que una tarea ha iniciado su ejecución no se bloqueará hasta terminar de ejecutarse y por tanto sólo podrá ser expulsada por tareas más prioritarias que no comparta recursos con ella. Otras características del SRP son que evita los interbloqueos y que una tarea se bloquea como máximo por un intervalo equivalente a una sección critica. Además, bajo el SRP no es necesaria la utilización de colas de espera ya que una tarea nunca se bloquea durante su ejecución. El SRP permite que las tareas compartan una misma pila de ejecución, lo

[67] highest-locker protocol
[68] Stack Resource Protocol
[69] preemption levels

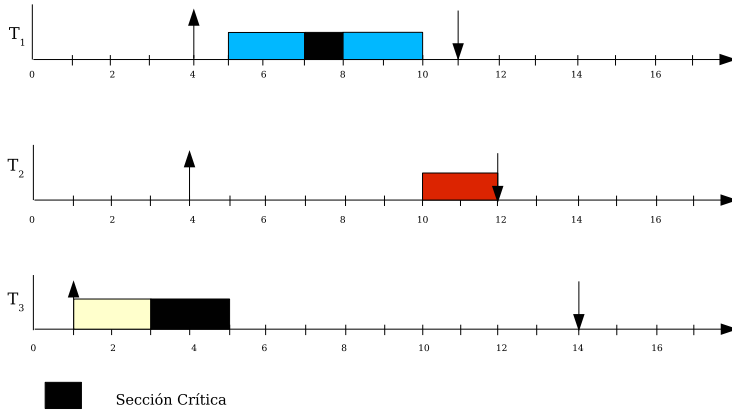

Figura 2.13: Ejemplo del SRP bajo el EDF

que ahorra memoria si existen más tareas que niveles de prioridad relativa.

Siguiendo con el ejemplo utilizado anteriormente, la Figura 2.13 muestra la ejecución del mismo sistema de la Figura 2.12, pero planificado ahora utilizando el SRP. De acuerdo a la definición, el valor del *nivel de expulsión* del recurso compartido por las tareas T_1 y T_3 es igual a $\frac{1}{7}$, que es el valor del inverso del plazo relativo de T_1. En el instante de tiempo 1, en el que la tarea T_3 es liberada, el valor del *techo del sistema* es cero. Debido a que en el instante 1, el valor del *nivel de expulsión* de T_3 es estrictamente mayor que el del *techo del sistema* ($\frac{1}{13} > 0$), T_3 inicia su ejecución inmediatamente provocando que el valor del *techo del sistema* sea igual al *nivel de expulsión* del recurso. En el instante 4 las tareas T_1 y T_2 son liberadas y debido a que ninguna de las dos tiene un valor de *nivel de expulsión* mayor al del *techo del sistema*, la tarea T_3 continúa su ejecución. En el instante 5 la tarea T_3 concluye y el t*echo del sistema* tiene de nuevo el valor de cero. De entre las tareas activas en ese momento, T_1 es la más prioritaria y tiene un valor de *nivel de expulsión* mayor que el del *techo del sistema*, por lo que inicia su ejecución sin ser bloqueada hasta concluir. Como puede observarse, la tarea T_1 no inició su ejecución hasta que el recurso que requería estuviese disponible. Esta característica del SRP evita cambios de contexto innecesarios, lo que hace que el protocolo sea mucho más eficiente y fácil de implementar.

2.3.4.4. Tareas aperiódicas con recursos compartidos

En algunos sistemas se hace necesario que tanto las tareas periódicas como las aperiódicas compartan recursos entre si. En la sección anterior se estudiaron las técnicas más comunes para planificar tareas críticas que utilicen recursos compartidos. Sin embargo, la aplicación de ésas técnicas en sistemas que incluyan tareas

aperiódicas no es trivial. Si una tarea aperiódica consume su crédito de ejecución dentro de una sección crítica no puede ser suspendida por que pondría en riesgo la ejecución de las tareas críticas. En esta sección se estudiarán algunas de las soluciones propuestas.

Uno de los primeros trabajos en abordar el problema fue hecho por Ghazalie y Baker en [47]. En su artículo, se extendieron las definiciones de los servidores *diferido* y *esporádico* para que pudieran ser utilizados con planificadores dinámicos como el EDF, y con recursos compartidos. Para los casos del *servidor diferido*, y su versión dinámica llamada *servidor diferido de plazo*, si se consume el crédito del servidor en una sección crítica, se permite que el servidor incurra en *cómputo en exceso*[70] al continuar su ejecución recargando inmediatamente su crédito lo necesario para salir de la sección crítica. Para garantizar que no se afecte la ejecución de las tareas críticas del sistema al asignar más tiempo de cómputo que el asignado inicialmente para la carga aperiódica, el crédito en exceso será restado del crédito del servidor cuando sea recargado en el siguiente periodo. En los casos del *servidores esporádico* y del *servidor esporádico de plazo* también se utiliza crédito extra en las secciones críticas, de ser necesario. El crédito en exceso utilizado en un periodo debe será descontado en futuras recargas del crédito del servidor.

Kuo y Li, en [59], propusieron una extensión del modelo de *sistema abierto* presentado por Deng y Liu en [35] para planificar recursos compartidos. Un sistema abierto es aquél en el que no es posible conocer *a priori* las características del sistema ni de sus tareas. En su artículo, Kuo y Liu proponen un mecanismo global para la sincronización de recursos en un sistema abierto, basado en el uso de los protocolos de sincronización *PCP* y *SRP*, y del *servidor esporádico*.

Lipari y Buttazzo en [73] extendieron el algoritmo de *servidor de ancho de banda total* (*TBS*), al que integran el protocolo *SRP* para abordar el problema de la sincronización en presencia de tareas aperiódicas. En su propuesta, los *niveles de expulsión* de las tareas aperiódicas se asignan de acuerdo a:

$$\pi_k \ = \ \frac{U_s}{C_k}$$

La propuesta de Lipari y Butazzo considera además el reclamo de tiempo de procesador no utilizado cuando las tareas aperiódicas terminen antes.

Otra propuesta para servidores aperiódicos fue hecha por Caccamo y Sha en [23], en la que proponen una extensión al algoritmo CBS de tal manera que se conserven las principales propiedades del protocolo SRP y se permita así la compartición de recursos entre tareas aperiódicas. A esta nueva versión del CBS le llamaron *CBS-R*. En el CBS-R, antes de que una tarea entre en una sección crítica, debe verificarse que el servidor tenga crédito suficiente para ejecutar ξ_i, que es la duración de la mayor sección crítica de la tarea τ_i. Si $c_s \ < \ \xi_i$ (tal que $\xi_i \ < Q_s$),

[70] overrun

se recarga el crédito del servidor $c_s = c_s + Q_s$ y se genera un nuevo plazo. Otro cambio con respecto a la definición original del CBS esta relacionado con la manera de definir el plazo y el valor del crédito del servidor. En el CBS-R, cuando una nueva tarea se activa y el servidor está inactivo, si $c_s \geq (d_s - r_i)U_s$ el servidor genera un nuevo plazo $d_s = r_i + T_s$ y el crédito recargado al máximo valor Q_s. De otra manera, el servidor genera un nuevo plazo $d_s = max(r_i + T_s,\ d_s)$ y el crédito será $c_s = c_s + (d_s^{new} - d_s^{old})U_s$. Como puede observarse, para implementar el CBS-R es necesario conocer la duración de la mayor sección crítica de cada tarea.

Por otra parte, en [34], de Niz *et al.* proponen el uso de un *núcleo de recursos*[71] para abordar el problema de la sincronización en *sistemas basados en reserva*[72]. Su propuesta hace una extensión de PCP al introducir el concepto de *herencia de reserva* bajo planificación estática. Una idea similar, pero para planificación dinámica, fue propuesta por Lamastra *et al.* y se estudia a continuación.

Lamastra *et al.* en [61] y [74] proponen el algoritmo de *herencia de ancho de banda* (*BWI*)[73] para compartir recursos en un sistema abierto. Una de las principales motivaciones de esta propuesta es el de permitir la planificación de tareas sin necesidad de conocer previamente los tiempos de ejecución de sus secciones críticas, que si que necesitan conocerse al utilizar algunos de los algoritmos previamente estudiados. El BWI se propone como una extensión natural del PIP. Básicamente el BWI modifica el funcionamiento del CBS al agregarle algunas reglas. Si una tarea aperiódica τ_i se bloquea al intentar utilizar un recurso *R* que ha sido asignado a otra tarea τ_j, entonces τ_j se agrega a la lista del servidor S_i. Si se diera el caso en que τ_j esta bloqueada en espera de un recurso *R'* entonces se sigue la cadena de todas las tareas hasta que se encuentre una que no este bloqueada. De esta manera, cada servidor tendrá más de una tarea por atender pero sólo una de ellas no estará bloqueada. Debido a que el PIP no evita los interbloqueos, deben evitarse utilizando alguna técnica adicional, como imponiendo un orden sobre la asignación de recursos. Cuando una tarea τ_j libera un recurso *R*, una de las tareas bloqueadas en *R* deja de estarlo, por ejemplo, τ_i. El servidor S_i deberá en ese momento eliminar a τ_j de su lista. De la misma manera, todos los servidores que agregaron a τ_j a sus listas deberán eliminarla y agregar a τ_i. Así como una tarea hereda una prioridad en el protocolo PIP, usando el protocolo BWI una tarea aperiódica τ_j "hereda" el servidor S_i de la tarea más prioritaria τ_i bloqueada en el recurso. Por otra parte, si una tarea aperiódica consume su crédito en una sección crítica, se recarga el crédito a su máximo valor y se genera un nuevo plazo. Como τ_j heredó al servidor S_i, el crédito consumido ha sido el de ese servidor, por lo que

[71] resource kernel
[72] reservation-based systems
[73] BandWidth Inheritance

después de consumir su crédito dentro de la sección crítica, τ_j puede ejecutarse ahora en el servidor S_j y consumir su crédito. En todo caso, la tarea se ejecutará en el servidor con mayor prioridad (menor plazo) hasta salir de la sección crítica. Debido a que la tarea se ejecuta bajo la política EDF, dependerá del plazo absoluto del servidor usado, en relación con el plazo del resto de tareas, el que suspenda su ejecución o la continúe con los nuevos valores de crédito y plazo.

En [95], Santos *et. al* detectaron que una de las limitaciones del protocolo BWI consiste en que el crédito del servidor bloqueado puede ser consumido por una tarea de menor prioridad sin que le sea retribuido, lo que afecta a la ejecución de las tareas planificadas por él. Para solventar esta limitación, propusieron el *Algoritmo de Limpieza de Fondos*[74], en el que cada servidor lleva control del crédito de otros servidores utilizados por sus tareas, para retribuirlo posteriormente en caso de ser necesario. Además, en ciertos instantes de tiempo llamados *singulares*, que coinciden con el final de cada *periodo de ocupación*[75] (*busy period*), los servidores *olvidan* sus débitos y el sistema se reinicia al establecerse nuevos valores de crédito y plazo para cada servidor.

Baruah, en [16], presentó una propuesta para utilizar políticas de planificación tanto *expulsivas* como *no expulsivas* para sistemas de tiempo-real basados en el uso de un procesador. La propuesta se basa en el hecho de que algunos estudios demuestran que la mayoría de las secciones críticas en los sistemas de tiempo-real son relativamente cortas [85], por lo que si son ejecutadas de manera no expulsiva, mientras si se permita la expulsión fuera de las secciones críticas, se obtendrían los beneficios de una mayor factibilidad y más sencillo acceso a recursos. La propuesta supone el uso del EDF, limitando su capacidad de expulsión en las secciones críticas. El modelo de planificación considera el cálculo de los segmentos de código más grandes, llamados *chunks*, que pueden ser planificados de manera no expulsiva. Una vez conocidos estos parámetros pueden ser utilizados para simplificar la planificación y reducir la sobrecarga, por una parte debido a que algunos recursos compartidos serán requeridos por las tareas por una cantidad de tiempo no mayor a sus parámetros no expulsivos, lo que permite un simplificado acceso a los recursos al no permitir la expulsión en ellos. Por otra parte, aún en ausencia de recursos compartidos críticos, la planificación en tiempo de ejecución se simplifica al conocer que una tarea que obtiene el acceso a un recurso se ejecutará por una cierta cantidad de tiempo previo a ser quizás expulsada por otra tarea.

Como se puede deducir del estudio de la planificación de tareas de sistemas de tiempo-real, se han propuesto una buena cantidad de políticas de planificación para planificar sistemas con diversas características. Cada política presenta ventajas y desventajas y el desarrollador de sistemas de tiempo-real dispone de alternati-

[74] Clearing Fund Algorithm

[75] intervalo en el que los servidores están ocupados sin interrupción

vas para elegir la mejor política de acuerdo a las características del sistema. Sin embargo, los estándares existentes, así como la mayoría de sistemas operativos de tiempo-real desarrollados a la fecha, incluyen un número muy limitado de las técnicas y algoritmos presentados aquí. Uno de los estándares más importantes y con mayor influencia, relacionados con tiempo-real, es el estándar POSIX. En el siguiente capítulo se estudiará brevemente el estándar POSIX y en particular sus extensiones de tiempo-real, con la finalidad de identificar cuáles de las técnicas de planificación hasta el momento estudiadas han sido incorporadas al estándar.

Capítulo 3

Estándares POSIX de Tiempo-Real

POSIX, la *Interfaz de Sistemas Operativos Portátiles*[1] es un conjunto evolutivo de estándares que tienen como principal objetivo el de soportar la portabilidad de las aplicaciones al nivel de código fuente. POSIX define una interfaz de sistema operativo basado en el sistema operativo *UNIX*[2]. Es importante destacar que POSIX no es una especificación sobre portabilidad a nivel de código binario sino que define una serie de *Interfaces para Programas de Aplicación* (*API's*) en las que se especifica lo que los usuarios recibirán del sistema operativo.

A pesar de que originalmente se utilizaba para hacer referencia al estándar *IEEE 1003.1-1988*, el término POSIX refiere más correctamente a una familia de estándares relacionados: el estándar IEEE 1003.*n* y las partes del estándar *ISO/IEC 9945*, en donde *n* es un número que indica una parte específica del estándar. El término "POSIX" surgió para diferenciar al estándar 1003.1 del resto de la familia de estándares POSIX.

POSIX a ido evolucionando desde que fue propuesto y aprobado por primera vez y muchos grupos pequeños se han formado para tratar áreas específicas del trabajo de estandarización, como lo han sido la base de funcionalidad UNIX (1003.1), el conjunto de comandos (1003.2 o POSIX2) y las extensiones de tiempo-real (1003.4 o POSIX4), entre otros. Con el paso de los años se han hecho adiciones y enmiendas al estándar en subsecuentes revisiones. Por ejemplo, el conjunto de comandos (POSIX.2) o las extensiones de tiempo-real (POSIX.4) han sido incorporadas el estándar 1003.1 o POSIX, y por lo tanto no existen como tales. En esta tesis se utilizarán indistintamente los términos POSIX, POSIX.1 e IEEE 1003.1 para referir al mismo conjunto de estándares.

[1] Portable Operating System Interfaces

[2] UNIX es una marca registrada por *The Open Group*

3.1. Breve historia de POSIX

Los trabajos de POSIX fueron iniciados por el *grupo de usuarios UNIX*, conocidos también como */usr/group*, en un intento por integrar las interfaces de los sistemas *AT&T System V* y *Berkeley CSGR*. Este esfuerzo fue conocido como *The 1984 /usr/group Standard* [1]. Después de la publicación del documento en el que presentaban sus resultados, el grupo decidió buscar reconocimiento del IEEE para el estándar. En 1984 el grupo dio por terminadas sus actividades y muchos de sus miembros se involucraron en los trabajos del Comité POSIX del IEEE de tal manera que lo hecho anteriormente pudiera servir como la base de un estándar internacional oficial. En Agosto de 1988 se aprobó dicho estándar como el *IEEE Standard 1003.1-1988, a Portable Operating System Interface for Computer Environments* [50]. Ediciones posteriores de POSIX.1 fueron publicadas en 1990, 1996 y 2001.

La edición de 1990 fue una revisión de la publicada en 1988 y significó un esfuerzo importante debido a que esta edición se convirtió en el estándar base estable sobre el cuál futuras enmiendas fueron llevadas a cabo. Al mismo tiempo, la edición de 1990 fue también aprobada como el estándar internacional *ISO/IEC 9945-1:1990*.

La edición de 1996 mantuvo intacto el texto básico y agregó enmiendas publicadas como los estándares *IEEE Std. 1003.1b-1993, IEEE Std. 1003.1C-1995* e *IEEE Std. 1003.1I-1995*. Algunos de estos estándares están relacionados con sistemas de tiempo-real. La edición de 1996 fue también aprobada como un estándar internacional, el *ISO/IEC 9945-1:1996*.

En 1999 se llevó a cabo la primera gran revisión al estándar base después de 10 años de existencia y se acordó que los trabajos fueran coordinados por el *Austin Group*, formado tanto por miembros del *IEEE Computers Society's Portable Standars Committee* (*PASC*), como de *The Open Group* y del *ISO/IEC Joint Techical Committee*. Los resultados fueron publicados en la edición de 2001 del estándar IEEE 1003.1. Se decidió también publicar en documentos independientes los resultados de otros proyectos que en ese momento estaban desarrollándose.

El grupo *PASC* fue formado en 1985 con el nombre de *Comité Técnico en Sistemas Operativos*[3], bajo el auspicio del *IEEE/SC*[4], y en 1992 fue designado con su nombre actual. *The Open Group* es un consorcio neutral mercadológica y tecnológicamente[5], que tiene como objetivo permitir el acceso integral a la información, dentro de las empresas y entre empresas, con base en estándares abiertos e interoperabilidad global.

Actualmente, el trabajo relacionado con los estándares POSIX se lleva a cabo

[3]Technical Committee on Operating Systems

[4]IEEE Standards Association Standards Board and IEEE Computer Society

[5]vendor-neutral and technology-neutral

con la participación de dos grupos: el *Grupo de Trabajo*[6] y el *Grupo de Votación*[7]. El primero de estos grupos se reúne periódicamente y propone estándares, que son enviados al grupo de votación. El grupo de votación participa en la revisión formal del estándar propuesto. Un estándar no es enviado para su aceptación al *Comité de Estándares de la Asociación de Estándares del IEEE*[8] hasta que haya cumplido los requerimientos de aceptación del grupo de votación. Actualmente, el grupo PASC continúa con el desarrollo de la familia de estándares POSIX, promoviendo la formación de grupos de trabajo y de votación, apoyando y coordinado sus actividades, de acuerdo a las políticas y procedimientos del IEEE/SC.

La edición más reciente del estándar se publicó el 30 de Abril de 2004 y actualiza la edición de 2001 al incluir los *Technical Corrigendum 1* (*TC*1) y *Technical Corrigendum 2* (*TC2*). Este estándar ha sido publicado conjuntamente por el IEEE y The Open Group. Al igual que las ediciones precedentes, la del 2004 es también un estándar internacional. La publicación ISO/IEC se hizo el 18 de Agosto de 2003 bajo la denominación de *ISO/IEC 9945:2003*. Este estándar define una interfaz de ambiente y sistema operativo, incluyendo un intérprete de comandos y un conjunto de programas de utilidades comunes para soportar la portabilidad de las aplicaciones a nivel de código fuente. Está formado por cuatro componentes o volúmenes:

1. El volumen de *Definiciones Básicas*[9] que incluye términos generales, conceptos e interfaces comunes a todos los volúmenes del estándar, incluyendo convenciones de utilidades y definiciones de encabezados de lenguaje C [51].

2. En el volumen de *Interfaces de Sistema*[10] se incluyen definiciones para las funciones y sub rutinas, servicios de sistema específicos para el lenguaje de programación C, aspectos de lenguaje incluyendo portabilidad, manejo y recuperación de errores [54].

3. Definiciones para la interfaz estándar a nivel de código fuente así como de servicios de interpretación de comandos, se incluyen en el volumen de *Intérprete de Comandos y Utilidades*[11] [53].

4. El volumen *Informativo*[12] contiene información histórica relacionada con

[6] Working Group
[7] Balloting Group
[8] IEEE Standards Association - Standards Board
[9] Base Definitions
[10] System Interfaces
[11] Shell and Utilities
[12] Rationale (Informative)

Estándar	Nombre
IEEE Std 1003.1b-1993	Extensiones de Tiempo-Real
IEEE Std 1003.1c-1995	Hilos (Threads)
IEEE Std 1003.1d-1999	Extensiones de Tiempo-Real Adicionales
IEEE Std 1003.1j-2000	Extensiones de Tiempo-Real Avanzada
IEEE Std 1003.1q-2000	Tracing

Cuadro 3.1: Estándares POSIX de Tiempo-Real

el contenido del estándar y la razón por la cual algunas características fue incluidas o excluidas del mismo [52].

3.2. POSIX y Tiempo-Real

Debido a que las aplicaciones de tiempo-real también requieren de garantías relacionadas con su portabilidad, el *PASC Real-Time System Services Working Group* (*SSWG-RT*) ha desarrollado una serie de estándares que enmiendan el estándar IEEE Std. 1003.1-1999 y el estándar de perfiles IEEE Std. 1003.13-1998. Uno de estos estándares, llamado inicialmente IEEE Std. 1003.1b-1993 o Extensiones de Tiempo-Real, define un conjunto de extensiones de tiempo-real al estándar POSIX.1, como por ejemplo planificación basada en prioridades, memoria compartida, señales de tiempo-real y temporizadores. Cabe recordar que todo estos estándares y enmiendas han sido integrados a POSIX en la edición de 2004 y por lo tanto no existen más con sus nombres anteriores. Por ejemplo, el estándar IEEE Std. 1003.1b ha dejado de existir con ese nombre al ser incorporado a POSIX bajo el nombre de Extensiones de Tiempo-Real.

El objetivo del SSWG-RT es el de “desarrollar estándares que signifiquen cambios o adiciones mínimas a los estándares POSIX para soportar la portabilidad de aplicaciones con requisitos de tiempo-real” [6]. Los estándares relacionados con tiempo-real y sistemas empotrados se listan en el Tala 3.1, en donde se muestra el nombre del estándar original, aunque como se ha dicho, han sido incorporados bajo el nombre de POSIX. De estos estándares, las Extensiones para Tiempo-Real y el soporte de múltiples hilos en un proceso, son los estándares más comúnmente implementados.

La definición de POSIX de tiempo-real para sistemas operativos es “la habilidad del Sistema Operativo de proporcionar el nivel de servicio necesario para acotar los tiempos de respuesta” [51]. Por medio de las Extensiones de Tiempo-Real es posible agregar a POSIX los servicios necesarios para lograr un comportamiento temporal predecible, especialmente en áreas tales como la planificación

de tareas y procesos, señales de tiempo-real, administración de memoria virtual, relojes y temporizadores, etc. Además, las Extensiones de Tiempo-Real facilitan la programación concurrente para la sincronización de procesos, memoria compartida, comunicaciones síncronas y asíncronas, y colas de mensajes.

Algunos de los servicios de las extensiones de tiempo-real de POSIX son:

- *Planificación Basada en Prioridades Fijas*: POSIX no especifica política de planificación alguna ni concepto de prioridad. Sin embargo, las Extensiones de Tiempo-Real de POSIX definen políticas de planificación expulsivas basadas en prioridades fijas con un mínimo de 32 niveles de prioridad. Si dos procesos se activan y tienen diferentes valores de prioridad, se elijará para ejecución al proceso con mayor prioridad. Se definen cuatro políticas de planificación: *SCHED_FIFO*, *SCHED_RR*, *SCHED_SPORADIC* y *SCHED_OTHER*.
  SCHED_FIFO es una estrategia basada en la política de "primero-en-entrar-primero-en-salir" (FIFO). La política FIFO se aplica a procesos del mismo nivel de prioridad. Cuando un proceso se activa es colocado al final de la lista de procesos listos (del mismo nivel de prioridad). El proceso que encabece la lista será ejecutado hasta que se complete o se bloquee, en cuyo caso será eliminado de la lista. A esta técnica de planificación se le conoce también como "ejecución hasta terminar"[13] o "ejecución hasta bloquearse"[14]. SCHED_RR es similar a SCHED_FIFO, con la diferencia de que en un mismo nivel de prioridad una estrategia de paso de testigo[15] es utilizada. SCHED_OTHER se definió para que un implementador pueda utilizar alguna política de planificación propia[16] mientras que SCHED_SPORADIC se describe a continuación.

- *Política de Planificación de Servidor Esporádico*: El servidor esporádico (SS) es utilizado para planificar tareas aperiódicas en sistemas de tiempo-real [102] [47]. Utilizando el servidor esporádico, una cierta capacidad acotada de capacidad de cómputo es reservada para procesar los eventos aperiódicos al más alto nivel de prioridad. Una vez que esta capacidad de ejecución ha sido consumida, las tareas aperiódicas que no hayan concluido se ejecutan en segundo plano, al menor nivel de prioridad.

- *Señales de Tiempo-Real*: Las señales son un mecanismo básico de comunicación asíncrona entre procesos. Las señales de tiempo-real mejoran el estándar POSIX al agregar más números de señales para uso por parte de

[13] run-to-completion
[14] run-to-block
[15] round robin
[16] implementation-defined

las aplicaciones. Las señales de tiempo-real son entregadas en orden, de tal manera que puede utilizarse el número de señal como su prioridad. Además, las señales contienen un campo extra de datos que puede ser utilizado por parte de las aplicaciones para identificar la fuente de la señal. Otra de las mejoras en las señales es que se colocan en una *cola de señales pendientes* hasta que son entregadas, evitando así su pérdida.

- *Relojes y Temporizadores*[17]: Los relojes de tiempo-real deben proporcionar una resolución del orden de los nanosegundos. Los temporizadores se benefician de esta alta resolución al poderse programar eventos más precisos. Cuando un temporizador expira se genera una señal al proceso que creó el temporizador.

- *Colas de Mensajes*[18]: Las Extensiones de Tiempo-Real proporcionan servicios de paso de mensajes para la comunicación entre procesos. Esto permite que los mensajes sean colocados en una cola y que se pueda asociar un número de prioridad a cada uno de ellos. El envío y recepción de mensajes puede hacerse de manera asíncrona, y ya sea con bloqueo o sin este.

- *Memoria Compartida*: La memoria compartida permite que se compartan porciones de memoria física con una baja sobrecarga. Debido a que los procesos tiene espacios de direcciones independientes, con los servicios de memoria compartida dos o más procesos pueden *proyectar*[19] el mismo objeto de memoria. También los ficheros pueden ser proyectados en espacios de direcciones de uno o más procesos.

- *Semáforos:* Si se utiliza memoria compartida o ficheros proyectados[20] se hace necesario un mecanismo de sincronización. Las Extensiones de Tiempo-Real definen funciones para la sincronización basados en el uso de *semáforos con contadores*[21]. Los semáforos son un mecanismo de sincronización común que permite el acceso a recursos compartidos de una manera mutuamente exclusiva.

- *Entradas y Salidas Asíncronas*[22]: Los mecanismos de *entradas y salidas* (*E/S*) asíncronas ofrecen la habilidad de solapar el procesamiento de la aplicación y las operaciones de E/S iniciadas por ella. Este servicio permite que

[17] Clocks and Timers
[18] Message Queues
[19] map
[20] mapped files
[21] counting semaphores
[22] Asynchronous I/O

las aplicaciones interactúen con el subsistema de entradas y salidas de forma asíncrona. Si un proceso inicia una operación de E/S puede continuar con su ejecución de manera paralela a la operación de E/S. Una vez que la operación de E/S ha finalizado se envía una señal a la aplicación para notificárselo.

- *Bloqueo de Memoria*[23]: Cuando una aplicación necesita más memoria que la existente, el sistema operativo mueve partes de memoria de algunas aplicaciones a medios de almacenamiento secundario, como el disco duro. La información almacenada en disco vuelve a la memoria del sistema cuando es requerida de nuevo. Las técnicas que utilizan este tipo de mecanismos son la *paginación*[24] y el *intercambio*[25]. Sin embargo, cuando se utilizan estos mecanismos, la administración de la memoria introduce incertidumbre en los tiempos necesarios para acceder a la memoria, ya que los accesos al medio de almacenamiento secundario pueden no estar acotados. Para solucionar este problema, POSIX ha definido el *bloqueo de memoria*, que permite establecer que aplicaciones o partes de las aplicaciones no podrán ser movidas de la memoria principal. Las técnicas de bloqueo de memoria eliminan esta impredecibilidad inherente al uso de la paginación y el intercambio, y proporcionan una tiempo de respuesta acotado al llevar a cabo operaciones en memoria. El bloqueo de memoria se define con respecto al espacio de direcciones de un proceso. Idealmente, la cota o límite superior será la misma que el tiempo requerido por el procesador para acceder a la memoria principal, incluyendo cualquier cálculo de dirección o *fallo de cache*[26]. Al utilizar el bloqueo de memoria se consigue determinismo en los tiempos de respuesta.

3.3. Estándar de Perfiles de Entornos POSIX.13

Intentar incluir todas las características definidas por POSIX en un sistema operativo pudiera no ser siempre lo más apropiado. Existen sistemas en los que basta un subconjunto de las interfaces propuestas por POSIX, como en los sistemas empotrados de tiempo-real que tienen limitaciones de espacio y recursos. El estándar de perfiles de entorno de POSIX o POSIX.13[27] [55] se definió para atender este problema, proporcionando un los adecuados subconjuntos con ca-

[23]Memory Locking
[24]paging
[25]swapping
[26]cache miss
[27]POSIX.13 Profile Standard

Perfil	Varios Procesos	Hilos	Sistema de Ficheros
PSE51	NO	SI	NO
PSE52	NO	SI	SI
PSE53	SI	SI	NO
PSE54	SI	SI	SI

Cuadro 3.2: Perfiles de Entornos de Tiempo-Real

racterísticas del estándar base que son necesarias para cada entorno particular de aplicación. El estándar define cuatro perfiles de entorno de aplicación:

- *PSE51* o *Perfil de Sistema Mínimo de Tiempo-Real*[28]: Desarrollado para sistemas empotrados pequeños sin *Unidad de Administración de Memoria* (*MMU*)[29], sin discos o sistema de ficheros, y sin terminal. Tan sólo se permite un proceso con múltiples hilos en ejecución.

- *PSE52* o *Controlador de Tiempo-Rea*l[30]: Orientado para un controlador de propósito específico, sin MMU pero con un disco conteniendo un sistema de ficheros simplificado.

- *PSE53* o *Sistema Dedicado de Tiempo-Real*[31]: Corresponde con sistemas empotrados de gran tamaño, sin disco pero con MMU. Las aplicaciones pueden beneficiarse de los mecanismos de protección de memoria y comunicación de red en este perfil.

- *PSE54* o S*istema de Tiempo-Real de Propósito Múltiple*[32]: Este perfil esta orientado a grandes sistemas de tiempo-real con todas los servicios incluidos, entre ellos con un ambiente de desarrollo, comunicaciones de red, sistema de ficheros en disco, interfaz gráfica de usuario, entre otros.

El Cuadro 3.2 resume las características de los perfiles de entornos de tiempo-real.

Como se ha estudiado, POSIX ha definido un conjunto de estándares para garantizar la portabilidad de las aplicaciones de tiempo-real. Sin embargo, en el ámbito de la planificación de tareas, POSIX define tan sólo políticas basadas en prioridades fijas. En el siguiente capítulo se estudiarán algunas propuestas para

[28]Minimal Real-Time System Profile
[29]Memory Management Unit
[30]Real-Time Controller
[31]Dedicated Real-Time System
[32]Multi-Purpose Real-Time System

añadir flexibilidad a un sistema operativo en la planificación de tareas, y se estudiará con detalle los modelos propuestos para definir políticas de a nivel de usuario.

Capítulo 4

Planificación Definida por el Usuario

En el capítulo 2 se presentó el modelo de sistema a considerar en el ámbito de esta tesis. Se estudió básicamente el modelo de tareas periódicas con un sólo procesador y con la inclusión de tareas esporádicas y recursos compartidos. Por otra parte, es importante destacar que no es el único modelo de sistema existente. Sha *et al.* presentan en [97] una perspectiva histórica de la planificación de tareas de tiempo-real desde la aparición del artículo de Liu y Layland [77] hasta la fecha, así como otros modelos de sistema y una discusión de los retos a los que se enfrenta la planificación de tiempo-real.

Se han revisado ya diversas políticas para planificar sistemas con características específicas y se ha comprobado que la teoría de planificación ha madurado lo suficiente desde la publicación del fundamental artículo de Liu y Layland. Sin embargo, la inmensa mayoría de sistemas operativos de tiempo-real ofrece casi exclusivamente planificación basada en prioridades fijas [92], a pesar de que se ha demostrado que la planificación basada en prioridades dinámicas permite un mejor uso de los recursos disponibles [21] y que no todas las aplicaciones de tiempo-real pueden ser planificadas satisfactoriamente utilizando exclusivamente planificación basada en prioridades estáticas. Además, en el capítulo 3 se estudió el estándar POSIX de tiempo-real y pudo comprobarse que define solamente políticas de planificación basadas en prioridades estáticas.

Para agregar más políticas de planificación a un SOTR[1] varias alternativas pueden seguirse. Una de ellas consiste en modificar el sistema operativo para agregarle nuevas políticas de planificación. La gran desventaja de esta alternativa radica por una parte en que se hace necesario modificar la estructura interna del sistema operativo, una tarea que no es sencilla pues hay que programar el núcleo del sistema operativo, invertir mucho tiempo e introducir la probabilidad de errores en el código del SOTR. Por otra parte, es tan grande la cantidad de políticas de plani-

[1] Sistema Operativo de tiempo-real

ficación disponibles que decidir cuáles de ellas incluir en un SOTR es difícil. Sin embargo, a pesar de las dificultades ya explicadas, algunas adiciones *a la medida* de políticas de planificación se han hecho utilizando SOTR existentes [13]. Estas implementaciones se han hecho principalmente por motivaciones académicas.

Otra alternativa, estudiada recientemente, consiste en la implementación de planificadores sin la necesidad de integrarlos al núcleo del SOTR. En algunos casos, las nuevas políticas de planificación se implementan como módulos definidos a nivel del usuario, que interactúan con el SOTR para planificar las tareas del sistema de acuerdo a alguna política específica. Si los planificadores definidos a nivel de usuario cumplen con las condiciones definidas en el capítulo 1, se dispondrá de un modelo adecuado para agregar nuevos servicios de planificación a un SOTR.

En este capítulo se estudiarán los modelos existentes para la definición de planificadores a nivel de usuario. De los modelos estudiados, se dará especial atención a la *Planificación Definida por el Usuario compatible con POSIX*, ya que a juicio del autor es el modelo más completo de los propuestos a la fecha. Su estudio permitirá identificar los aspectos en los que su interfaz puede ser mejorada y extendida, así como determinar la mejor estrategia para su implementación.

4.1. Planificación jerárquica de dos niveles

En [35], Deng y Liu propusieron un esquema jerárquico de dos niveles para planificar en un sistema abierto, multitareas y uniprocesador, aplicaciones de tiempo-real junto con aplicaciones que no son de tiempo-real. La definición de sistema abierto fue presentada en la página 43. El modelo propuesto por Deng y Liu permite que las aplicaciones sean planificadas de acuerdo a diferentes algoritmos de planificación. La Figura 4.1 muestra la arquitectura de la *planificación jerárquica de dos niveles*. Las aplicaciones que no son de tiempo-real son ejecutadas por un *servidor de ancho de banda* total (estudiado en la sección 2.3.4.4). Las aplicaciones de tiempo-real, dependiendo de sus características, pueden ser ejecutadas por un *servidor de ancho de banda total* o por un *servidor de utilización constante* [36]. Estos dos servidores son esencialmente iguales, con la diferencia de que el primero es de tipo *expulsivo*.

En el modelo, cada servidor mantiene una cola de tareas activas. Además, cada servidor tiene asociado un planificador *definido por el usuario*, llamado *planificador del servidor*. En la Figura 4.1 puede observarse que el servidor S_1 tiene asociada el algoritmo de planificación RM, junto con el protocolo de herencia de prioridades (PIP). Las políticas de planificación que se agregan al sistema, junto con sus respectivos servidores asociados, forman el primero de los niveles jerárquicos de la arquitectura. El segundo nivel está representado por el planificador del sistema operativo, que mantiene a los servidores del sistema, recarga el crédito

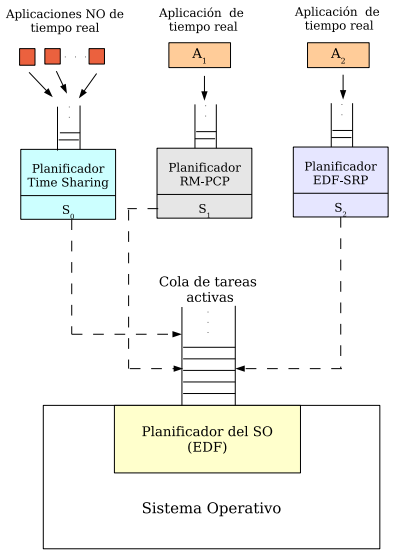

Figura 4.1: Modelo de la Planificación Jerárquica de Dos Niveles

de cada servidor y establece sus plazos de acuerdo a las características de las aplicaciones que ejecuten. Una característica interesante del planificador del sistema operativo es que planifica a los servidores activos utilizando el algoritmo EDF. Un servidor está activo para ser ejecutado si su crédito es diferente de cero y su cola de tareas activas no está vacía. Cuando el planificador del sistema operativo selecciona un servidor para ejecución, quien realmente se ejecuta es la tarea que encabece su lista de tareas activas. Por otra parte, en el modelo propuesto, el control de acceso a los recursos entre las aplicaciones se hace por medio del *protocolo no expulsivo de secciones críticas* (*NPS*[2]) [81].

Como se observa, en la *planificación jerárquica de dos niveles* cada política de planificación es controlada por un servidor. La razón radica en que el tiempo total de cómputo del procesador es dividido entre las diferentes políticas existentes, que está representado en cada servidor por el valor de su crédito. De esta manera, el análisis de planificabilidad se simplifica ya que basta con considerar el máximo tiempo de cómputo asignado a cada planificador al hacer el análisis. Si el conjunto de tareas, planificado bajo alguna política de planificación determinada, no utiliza más tiempo de cómputo que el asignado, será planificable y no interferirá con la planificación de las tareas del sistema que utilicen una política diferente. La idea de dividir el tiempo de cómputo total de un sistema, en *porciones* que serán asignadas a las aplicaciones, se basa principalmente en el algoritmo

[2]Nonpreemptable Critical Sections

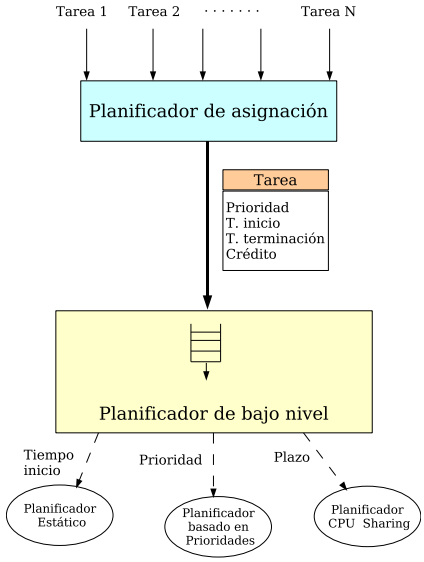

Figura 4.2: Modelo de planificación en Red-Linux

de *Compartición General del Procesador* (*GPS*[3]) [56], en el que las aplicaciones de un sistema se les asigna una cantidad máxima de recurso (procesador principalmente) que podrán utilizar. A este paradigma de planificación se le conoce como *asignación proporcional de recursos*[4], y ha sido la base para el desarrollo de políticas de planificación tales como el *servidor de ancho de banda constante*, entre otros.

4.2. Marco de referencia general para la planificación

Wang y Ling, en [112], propusieron un *marco de referencia general de planificación*, diseñado para integrar diversos paradigmas de planificación a un sistema operativo. En su propuesta, Wang y Ling establecen cuatro atributos de planificación con los cuales pueden implementarse, ya sea directamente, o proyectando sobre ellos algún otro atributo, cualquier algoritmo de planificación existente. Los atributos son: *prioridad, tiempo de inicio, tiempo de terminación* y *crédito*. Además, proponen la utilización de un planificador con dos componentes o niveles, de manera similar al modelo de Deng y Liu presentado anteriormente. La Figura 4.2 muestra la estructura del planificador propuesto. El primer componente, llamado

[3]General Processor Sharing
[4]proportional share resource allocation

planificador de asignación[5], determina el valor correcto de los cuatro atributos asociados con cada tarea, de acuerdo a la política de planificación utilizada. Por ejemplo, si se implementa la política de planificación RM, los parámetros que recibe el *planificador de asignación* son el plazo y tiempo de cómputo de cada tarea, y con ellos puede determinar los valores de *tiempo de inicio*, *tiempo de terminación* y *prioridad*. Además, tomando en consideración a la política de planificación implementada, el *planificador de asignación* debe determinar la *prioridad efectiva* de cada tarea. Si la política implementada es EDF, la *prioridad efectiva* de la tarea se determina utilizando el atributo *tiempo de terminación*, de tal manera que la tarea con mayor *prioridad efectiva* será la tarea con menor *tiempo de terminación* absoluto.

El segundo componente de la arquitectura se denomina *planificador de bajo nivel*[6], que examina los atributos de planificación de cada tarea, selecciona de la cola de tareas activas a la de mayor *prioridad efectiva*, y la ejecuta, tal y como lo hace un planificador basado en prioridades. Es además responsable de planificar todas las tareas de tiempo-real que han sido registradas en el *planificador de asignación*, ya que las tareas que no son de tiempo-real son planificadas por el sistema operativo nativo, que puede ser Linux.

En el *marco de referencia general de planificación*, el *planificador de asignación* tiene la función de traducir cualquier política de planificación a algunos de los cuatro atributos, y de determinar la función de evaluación para obtener la *prioridad efectiva* de cada tarea, tomando en consideración los atributos y la política utilizada. Y una vez que se ha determinado la prioridad efectiva de cada tarea activa, se coloca en la cola de tareas activas del *planificador de bajo nivel* para que sean ejecutadas en orden de prioridad. Esto significa que las políticas de planificación *definidas por el usuario* se implementan en el *planificador de asignación*, mientras que el *planificador de bajo nivel* implementa el *mecanismo* para ejecutarlas en el procesador.

La propuesta ha sido implementada en RED_Linux [112], un sistema operativo de tiempo-real que utiliza el mecanismo de emulación de interrupciones de RTLinux (que se explicará más adelante). En la implementación hecha en RED_Linux, el *planificador de asignación* es un proceso que se ejecuta en el espacio del usuario, mientras que el *planificador de bajo nivel* es un módulo del núcleo.

La propuesta de Wang y Ling, a diferencia de la de Deng y Liu, no ha sido diseñada para soportar aplicaciones complejas que utilicen diferentes políticas de planificación simultáneamente. En cambio, está más orientada a proporcionar un marco de referencia para la planificación que sea eficiente y reconfigurable, para

[5] allocator
[6] dispatcher

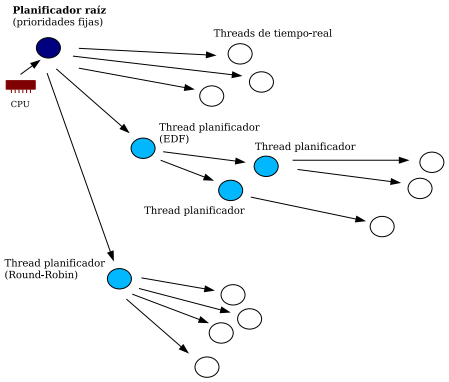

Figura 4.3: Modelo de Herencia de CPU

ser utilizado en sistemas empotrados.

4.3. Planificación de herencia de CPU

En [42] Ford y Susarla presentaron la *Planificación de Herencia de CPU*[7], un marco de referencia de planificación basado en la noción de herencia de prioridad, en el que threads arbitrarios actúan como planificadores de otros threads. En el modelo, los threads llamados *planificadores* donan temporalmente tiempo de cómputo (tiempo de CPU) a los threads que sean planificados por él. Los threads que han recibido el tiempo de cómputo pueden a su vez donarlo a otros threads, formando una estructura jerárquica de planificadores. Cuando el thread que ha recibido el tiempo de cómputo no lo necesita más, por que por ejemplo se ha bloqueado, informa a su correspondiente *thread planificador*, de tal manera que el tiempo de cómputo pueda ser asignado a otro thread. El mecanismo básico de planificación del modelo, llamado *despachador*[8], no necesita utilizar las nociones de prioridad, utilización del CPU, relojes o temporizadores, ya que estas funciones se implementan en los *threads planificadores* si llegan a ser necesarias. Bajo el modelo, cada thread es un procesador virtual cuyo propósito es el de ejecutar instrucciones. Un *thread planificador* es un thread que pasa la mayor parte del tiempo donando tiempo de cómputo a otros threads, llamados *threads clientes*. Los *threads clientes* heredan parte de los recursos de cómputo de su *thread planificador*, y tratan a ésa porción como su *procesador virtual*. Los únicos threads

[7] CPU Inheritance Protocolo
[8] dispatcher

del sistema que tienen asignado tiempo-real de cómputo son los *threads planificadores supervisores*[9]. Los otros threads sólo pueden ejecutarse si se les ha donado tiempo de cómputo. Existe un *thread planificador supervisor* por cada procesador del sistema.

A pesar de que todas las decisiones de planificación de alto nivel son tomadas por los *threads planificadores*, es necesario un mecanismo de bajo nivel para implementar las primitivas de administración de threads. A ese mecanismo de bajo nivel se le denomina el *despachador*. Su función consiste en implementar los mecanismos para bloquear y desbloquear threads, y para la donación de tiempo de cómputo entre threads, sin llevar a cabo decisiones de planificación. El *despachador* no es un thread, sino que es el único componente que debe estar implementado en el núcleo del sistema operativo.

Cuando un thread se bloquea al intentar entrar en una sección crítica, dona voluntariamente su tiempo de cómputo al thread que está en la sección crítica, en un mecanismo similar a la herencia de prioridades, sólo que en este caso lo que se hereda es el tiempo de cómputo.

En el modelo de la *Planificación de Herencia de CPU*, se ha definido que debe existir comunicación entre el *despachador* y los *threads planificadores*, aunque no se define la manera en que dicha comunicación debe llevarse a cabo. Por ejemplo, si un *thread cliente* se activa, el *despachador* informa al correspondiente *thread planificador* que uno de sus thread que requiere tiempo de cómputo. Si algún otro thread estaba en ejecución en ese momento, se expulsa del procesador y se ejecuta el *thread planificador*, que decidirá si el thread expulsado continúa con su ejecución, o si algún otro thread se ejecutará.

Si más de una política de planificación es utilizada al mismo tiempo, el modelo sugiere que todas ellas sean implementadas en un mismo *thread planificador*, ya que se trata de evitar introducir la noción de prioridad en el *despachador*. Sin embargo, para implementar diversas políticas usando más de un thread, dos alternativas pueden usarse. Una de ellas consiste en definir un *thread planificador primario*. Los *threads planificadores* toman las decisiones de planificación de acuerdo al algoritmo utilizado, y posteriormente donan su tiempo de CPU al *thread planificador primario*. El *thread planificador primario* insertará a cada *thread cliente* en la cola de threads activos del *despachador*, de acuerdo a algún criterio de prioridad global predefinido.

La otra alternativa consiste en implementar un *despachador* basado en prioridades, para asignar a cada *thread planificador* un nivel de prioridad y determinar así cómo ordenar a los *threads clientes* dentro de la cola de threads activos del *despachador*.

El modelo ha sido extendido para trabajar con más de un procesador.

[9] root scheduler threads

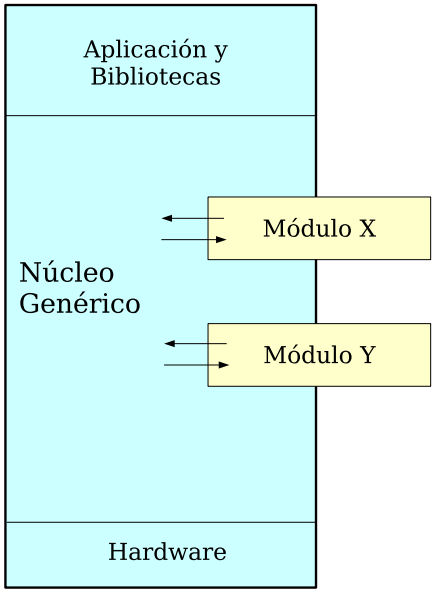

Figura 4.4: Arquitectura S.Ha.R.K.

4.4. Núcleo de tiempo-real crítico y no crítico

Gai *et al.*, en [44], propusieron una arquitectura dinámica reconfigurable de núcleo, llamada *núcleo de tiempo-real crítico y no crítico* (*S.Ha.R.K.*[10]), diseñada para una sencilla implementación, integración y evaluación de diferentes algoritmos de planificación. La principal motivación en su desarrollo es la de proporcionar una plataforma para crear de manera rápida y sencilla prototipos de algoritmos de planificación. La arquitectura de núcleo propuesta ha sido diseñada con el objetivo de mantener independencia entre los mecanismos del núcleo del sistema operativo y las políticas de planificación y de acceso a recursos, así como independencia entre las aplicaciones y los algoritmos de planificación.

La Figura 4.4 muestra la arquitectura del modelo propuesto. Para conseguir independencia entre las aplicaciones y los algoritmos de planificación, así como estos últimos y el núcleo, la arquitectura *S.Ha.R.K.* se basa en el uso de un *Núcleo Genérico*, que no implementa alguna política de planificación, pero si delega las decisiones de planificación a entidades externas llamadas *módulos de planificación*. De manera similar, el control de acceso a recursos se hace por medio de los *módulos de recursos*. Esto significa que las políticas de planificación que se agreguen al sistema deben estar implementadas en los *módulos*.

El *núcleo genérico* proporciona a los módulos los mecanismos necesarios para

[10]Soft and Hard Real-time Kernel

llevar a cabo la planificación y administración de recursos, permitiendo que exista una abstracción entre el sistema y los algoritmos que sean implementados. El núcleo proporciona un conjunto de primitivas que pueden utilizar los módulos al implementar alguna política de planificación.

Cada *módulo* esta formado por un conjunto de datos y funciones con los cuales se implementa la política de planificación. El modelo propone que exista también independencia entre *módulos*, de tal manera que los algoritmos para planificar tareas críticas se implementen en un *módulo*, los de sincronización en otro, e inclusive los algoritmos de servicio de tareas aperiódicas en un *módulo* independiente.

Para que pueda existir independencia entre las aplicaciones y los algoritmos de planificación, la arquitectura *S.Ha.R.K.* introduce el concepto de *modelo*. Cada tarea del sistema solicita ser planificada con algún nivel de *calidad de servicio* (*QoS*[11]), que está especificado por un *modelo*. Cada *modelo* es una entidad que permite separar los parámetros de planificación de los parámetros de *QoS* requeridos por el núcleo. Los *modelos* son descripciones de los requerimientos de planificación expresadas por las tareas. Existen dos tipos de modelos: *modelos de tareas* y *modelos de recursos*. El primero de ellos expresa los requerimientos de *QoS* de las tareas, que son especificados con un conjunto de parámetros. Por otra parte, el *modelo de recursos* se utiliza para definir parámetros *QoS* relacionados con un conjunto de recursos que son utilizados por una tarea. Por ejemplo, un *modelo de recursos* puede especificar que un conjunto de recursos determinado será utilizado por un thread por medio del *Protocolo de Herencia de Prioridades*.

Tanto tareas como modelos están compuestos por un conjunto de parámetros. En el caso de las primeras, deben especificarse dos parámetros obligatorios en toda tarea, llamado *criticalidad de la tarea*[12] y el *identificador del modelo*. Cada *modelo de tarea*, por otra parte, está compuesto por un identificador, un conjunto de parámetros obligatorios, y una secuencia de bits que sólo el módulo específico puede interpretar. El modelo de recursos sólo contiene un parámetro obligatorio, que es su *identificador*.

La Figura 4.5 ilustra el proceso de creación de tareas. Cuando una nueva tarea se crea, la aplicación solicita al núcleo su creación, y le envía el correspondiente *modelo* especificando los requerimientos de *QoS*. Un componente del núcleo, llamado *proyector de modelos*[13], pasa el *modelo* a algún *módulo* siguiendo una política interna. El *módulo* revisa si puede proporcionar la *QoS* solicitada, ya que si no puede hacerlo entonces el *proyector de modelos* selecciona otro *módulo* y repite el proceso. Cuando algún *módulo* acepta la tarea, convierte los parámetros *QoS* del *modelo* en parámetros de planificación. Esta conversión es llevada a cabo

[11] Quality of Service
[12] task criticality
[13] model mapper

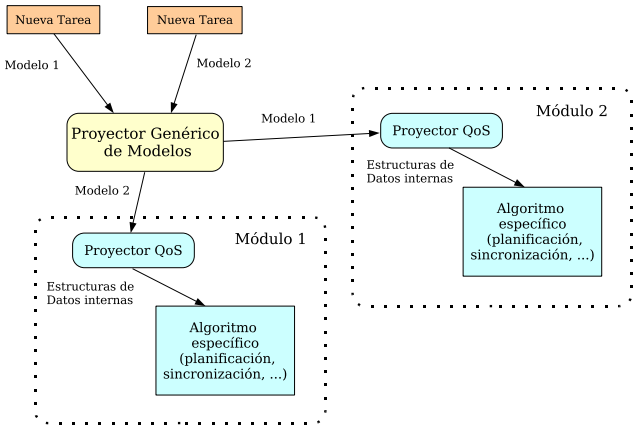

Figura 4.5: Interacción entre proyectores de Modelo y QoS

por un componente del *módulo* llamado *proyector de QoS*[14]. En términos generales, un *módulo* puede tan sólo administrar un subconjunto de *modelos*, que no está limitado por el núcleo.

Una vez que la tarea ha sido creada y asignada a algún *módulo*, cuando se requiere tomar una decisión de planificación, el núcleo pregunta al *módulo* correspondiente para que sea él quien tome la decisión de planificación, de acuerdo a la información contenida en los *modelos*. Por otra parte, cada *módulo* tiene asignada una prioridad fija, que es utilizada por el núcleo al momento de atender o planificar a los *módulos*. El núcleo selecciona para ejecución, a la tarea que encabece la cola de tareas listas de cada *módulo*, de acuerdo a la prioridad de cada *módulo*.

La arquitectura *S.Ha.R.K.* define una interfaz que proporcionan los *módulos*, y que está compuesta por un conjunto de funciones, que son agrupadas en tres clases: *llamadas de nivel*[15], *llamadas de tareas*[16] y *llamadas huéspedes*[17].

El sistema propuesto, además, es compatible con el estándar POSIX 1003.13 PSE52, que se estudió en la sección 3.3.

4.5. Vassal

Vassal es un sistema diseñado para agregar diferentes políticas de planificación a un sistema operativo. El modelo propone que las nuevas políticas se definan como módulos externos al núcleo, que puedan ser cargados y descargados

[14] QoS mapper
[15] level calls
[16] task calls
[17] guest calls

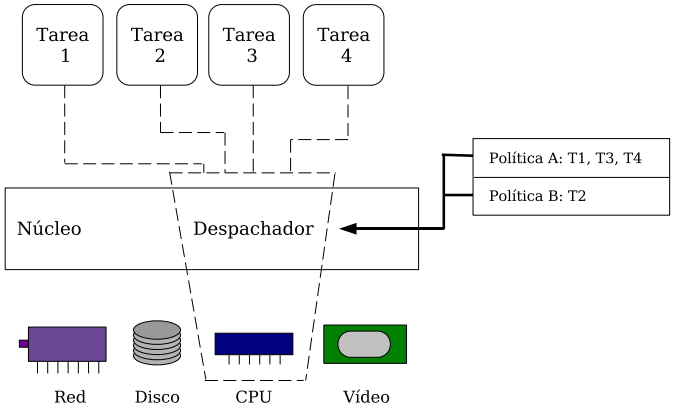

Figura 4.6: Arquitectura del Planificador Cargado

de manera dinámica. El modelo fue propuesto por Candea y Jones [24], y ha sido implementado en Windows NT, en donde cada nuevo algoritmo de planificación se agrega al sistema como un módulo del núcleo. La Figura 4.6 muestra la arquitectura del modelo propuesto, y un ejemplo del sistema formado por cuatro tareas y dos políticas de planificación, integradas como módulos. A pesar de que el modelo ha sido definido para implementarse en cualquier sistema operativo que permita cargar y descargar dinámicamente módulos del núcleo, su arquitectura están íntimamente relacionada con la arquitectura de Windows NT. En este sistema operativo, el núcleo contiene un despachador y no un planificador. La diferencia entre estos dos mecanismos consiste en que el primero asigna el procesador a las tareas, sin implementar alguna política de planificación, mientras que el segundo determina la siguiente tarea a ejecutar siguiendo alguna política establecida. Cuando es necesario determinar alguna decisión de planificación, el despachador invoca al módulo correspondiente para que sea él quien tome la decisión, para que posteriormente el despachador la lleve a cabo. Como puede verse, la arquitectura es similar a la de S.Ha.R.K, estudiada anteriormente. Sin embargo, algunas diferencias interesantes existen en Vassal, ya que incluye una interfaz más completa para comunicar a los elementos de un sistema. Cuando un módulo se carga al sistema, es necesario registrarlo en el núcleo. La función que permite esta operación es:

```
RegisterScheduler(scheduler,
                  decisión_maker,
                  message_dispatcher)
```

en donde el parámetro `decision_maker` contiene la dirección de memoria de la función que será invocada cuando sea necesario tomar una decisión de plani-

ficación relacionada con *scheduler*. El parámetro `message_dispatcher`, por otra parte, contiene la dirección de la función que debe ser ejecutada para manejar los mensajes que los threads envíen a su planificador.

Cuando una aplicación necesita comunicarse con su planificador, debe utilizar la siguiente función:

```
MessageToScheduler(scheduler, buffer, buflen)
```

en donde el parámetro `buffer` contiene el mensaje enviado a *scheduler*.

En ocasiones, un planificador necesita programar eventos futuros de planificación. Para hacerlo, la función que a continuación se presenta debe ser usada:

```
SetSchdulerEvent(scheduler,
                 performance_counter_reading)
```

el parámetro *performance_counter_reading* representa el instante de tiempo en el que debe invocar a *scheduler*.

Windows NT utiliza una máquina virtual, llamada *Capa de Abstracción de Hardware* (*HAL*[18]) [94], que a su vez utiliza un temporizador con un parámetro llamado *performance counter*, que es un valor de 64 bits que monotónicamente se incrementa, y cuyo contenido puede ser conocido usando la función *QueryPerformanceCounter()*. Como puede observarse, la arquitectura Vassal, así como su interfaz, está totalmente orientada a la arquitectura de Windows NT, lo que limita su portabilidad.

4.6. Planificadores Cargados Jerárquicos

En [88], Regehr y Stankovic propusieron un modelo para integrar diferentes algoritmos de planificación de tareas no críticas a un sistema operativo de propósito general, como Linux o Windows 2000. Los planificadores se agregan al sistema operativo como *módulos cargados*[19]. Al modelo le llamaron *Planificadores Cargados Jerárquicos* (*HSL*[20]), y está basado en la extensión de *activaciones del planificador*[21] [9]. Las *activaciones del planificador* es un mecanismo por medio del cual una acción de planificación es ejecutada por un planificador implementado en un thread y no por el núcleo. El núcleo tan sólo lleva un control de los espacios de direcciones de los threads planificadores, e informa de tales direcciones a los threads planificados cuando se requiere ejecutar algún evento de planificación.

[18] Hardware Abstraction Layer)
[19] loadable modules
[20] Hierarchical Loadable Schedulers
[21] scheduler activations

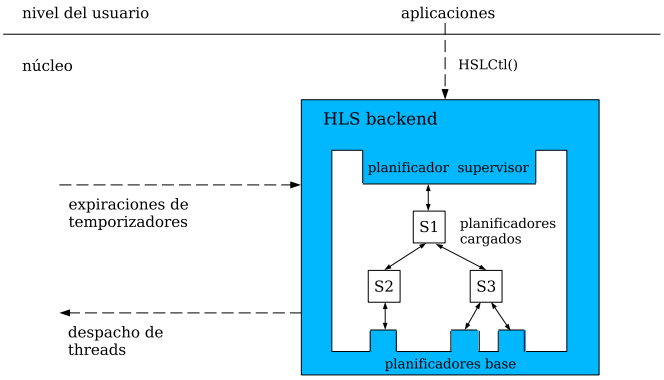

Figura 4.7: Estructura del modelo HSL

Este mecanismo ha sido propuesto para agregar flexibilidad en planificación en entornos multiprocesadores.

El modelo HSL ha sido posteriormente extendido por Abeni y Regehr en [3] para integrar cualquier política de planificación, y no tan sólo para planificar tareas no críticas.

HLS ha sido definido como un esquema jerárquico de planificación en el que existen relaciones padre/hijo entre planificadores. Cada planificador sólo puede comunicarse con su *planificador padre*, con sus *planificadores hijos* y con la *infraestructura HSL*. Las interacciones entre estos elementos se hace por medio de una interfaz bien definida, llamada el *API HSL*. Un *planificador hijo* puede registrarse con un *planificador padre*, solicitar que le asigne un procesador, liberar un procesador que esté utilizando, y enviar mensajes a su *planificador padre*, por ejemplo, para solicitar un cambio de su valor de prioridad o plazo. Por otra parte, un *planificador padre* puede otorgar un procesador o revocar el uso del mismo, a alguno de sus *planificadores hijos*. Las interacciones entre la *infraestructura HSL* y los planificadores ocurren cuando un planificador inicia su ejecución o cuando un temporizador, solicitado por un planificador, expira. Además, tal y como se especifica en el mecanismo de *activaciones del planificador*, cada planificador conoce en todo momento el número de procesadores que tiene asignados. Los planificadores sólo pueden interactuar utilizando el API HSL.

Para proporcionar una interfaz uniforme, cada procesador del sistema es presentado como un *planificador supervisor HSL*, que tiene algunas propiedades especiales: cada *planificador supervisor* tiene asignado tan sólo un planificador hijo y nunca le revoca el procesador. De manera similar, cada thread del sistema se representa como un planificador base, que explícitamente otorga el procesador cada vez que el thread se bloquea, y lo solicita en cada ocasión que deja de estar

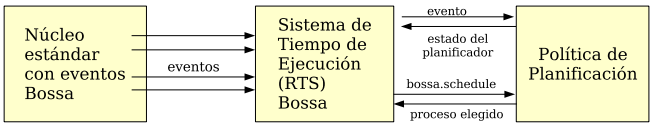

Figura 4.8: Arquitectura de Bossa

bloqueado. Los planificadores *supervisor* y *base* forman un interfaz entre el *HSL backend*[22] y los planificadores que se cargan en el sistema. Obviamente, esta interfaz es el API HSL, que es implementado por los planificadores *supervisor* y *base*. La estructura HSL se muestra en la Figura 4.7.

El modelo HSL ha sido implementado en Windows 2000 [88] y en Linux [3]. Además, se ha construido un simulador para crear y probar rápidamente nuevas políticas de planificación [3]. Sin embargo, los nuevos servicios de planificación no garantizan el cumplimiento de las restricciones temporales de las tareas del sistema, ya que el sistema operativo anfitrión (Linux o Windows) no es modificado para tal fin. Los autores, por otra parte, justifican este aspecto manifestando que "el problema de si los mecanismos del SO son capaces de soportar aplicaciones de tiempo-real debe ser considerado de manera separada del problema de la implementación de nuevos algoritmos de planificación, ya que existen soluciones al primer problema". Las soluciones a las que se hace mención han sido propuestas por Goel *et. al* en [48].

4.7. Bossa

Bossa ha sido definido por sus creadores como un lenguaje para el desarrollo de algoritmos de planificación [15] y como un marco de referencia para la implementación de planificadores [62] [82]. Bossa define un *Lenguaje de Dominio Específico* (*DSL*[23]), que incluye abstracciones de alto nivel para que el desarrollador de planificadores no tenga que preocuparse por los detalles de programación de bajo nivel. Bossa incluye también un compilador y verificador que optimiza el código desarrollado, y que verifica la consistencia interna de la política y su correcto funcionamiento, con respecto al comportamiento del núcleo. También define un marco de referencia que administra la interacción entre un planificador compilado con Bossa y el núcleo destino, a través de un intermediario llamado *Sistema de Tiempo de Ejecución* (*RTS*[24]). Todas estas características de Bossa tra-

[22] mecanismo básico del HSL
[23] Domain-Specific Language
[24] Run-Time System

Evento	Generado por
processBlock.*	llamadas de E/S, manejadores de dispositivos
processUnlock.*	manejadores de dispositivos, servicios de tiempo
processYield	primitiva *sched_yield()*
clockTick	manejador de interrupciones de reloj
processNew	*fork(), clone(), exec()*
processEnd	*exit(), kill()*
Schedule	Sistema de Tiempo de Ejecución (RTS) de Bossa

Cuadro 4.1: Eventos Bossa

bajan en conjunto para permitir que un desarrollador pueda implementar e integrar un planificador sin necesidad de tener conocimientos profundos de programación del núcleo de un sistema operativo.

La Figura 4.8 muestra la arquitectura de Bossa. El núcleo del sistema operativo debe ser modificado para que cada *punto de planificación*[25] sea reemplazado por una notificación de un *evento Bossa*. Cada punto de planificación se implementa como una colección de manejadores de eventos que son escritos en lenguaje Bossa y traducidos a un fichero en lenguaje C por un compilador dedicado. Al reemplazar las acciones de planificación de núcleo por notificaciones de eventos, el RTS debe procesar cada notificación de evento Bossa, quien a su vez invoca al manejador apropiado definido por la política de planificación. Los eventos Bossa se organizan de manera jerárquica, de acuerdo a la función y la fuente del evento. Por otra parte, el manejador invocado devuelve el estado del planificador, indicando si existe algún proceso en ejecución, o si existen procesos listos disponibles. Esta información es utilizada en la siguiente notificación del evento `bossa.schedule`. El RTS solamente invoca al *manejador bossa.schedule* de la política si el estado actual del planificador indica que no existe algún proceso en ejecución y algún proceso listo está disponible. De otra manera, si existe algún proceso en ejecución, el RTS permite que continúe ejecutándose. Si no existen procesos en ejecución o listos, el RTS ejecuta el *bucle inactivo del núcleo*[26].

El RTS es fijo y proporcionado por el marco de referencia. La política de planificación es proporcionada por el desarrollador e implementada utilizando el DSL Bossa (lenguaje Bossa). Tanto el RTS como la política de planificación compilada se integran en el núcleo.

Al construir una política de planificación utilizando el DSL Bossa, deben definirse:

[25] scheduling point
[26] kernel idle loop

- una colección de estructuras relacionadas con la planificación, que serán utilizadas por la política
- un conjunto de manejadores de eventos
- un conjunto de funciones de interfaz, que permiten la interacción entre los usuarios y el planificador

Además, para permitir la construcción de una planificación jerárquica, Bossa hace una distinción entre *Planificadores de Procesos* y *Planificadores Virtuales*. Los primeros, como su nombre lo indica, planifican a procesos, mientras que los segundos planifican a *Planificadores de Procesos*. El DSL Bossa incluye funciones para definir ambos tipos de planificadores.

Cada política de planificación debe definir los *atributos* de cada proceso, entre los que pueden considerarse el periodo, el plazo, etc. También, debe definir el *criterio de ordenación* de los procesos, que será utilizado al momento de seleccionar el siguiente proceso por ejecutar. Por ejemplo, al implementar un planificador EDF, el *criterio de ordenación* será el menor plazo.

El comportamiento del planificador será determinado por los eventos que suscriba. Por cada evento debe definirse el manejador que especificará las acciones que serán ejecutadas cuando el evento ocurra. El Cuadro 4.1 muestra la lista de eventos disponibles en Bossa, y las correspondientes entidades que los generan. Se ha reportado que Bossa ha sido implementado en Linux y RTLinux.

4.8. Otras propuestas

Hasta el momento se han estudiado algunas de las propuestas para agregar políticas de planificación a un sistema operativo publicadas a la fecha. La mayoría de los trabajos presentados han sido publicados antes que la planificación definida por el usuario de Aldea y González-Harbour, aunque también se han presentado trabajos posteriores, como la extensión del HSL para tareas críticas y Bossa. Sin embargo, no son las únicas propuestas publicadas. Por ejemplo, Wolfe propuso en [113] un *servicio de planificación de tiempo-real para CORBA*, llamado *RapidSched*, que permite integrar diferentes políticas de planificación por medio de diferentes implementaciones del *servicio de planificación*. Otra propuesta es presentada en [70] y [71], en donde se implementa un mecanismo que permite la definición de diferentes algoritmos de planificación en un segundo nivel jerárquico de planificación. Sin embargo, estas propuestas quedan fuera del ámbito de la tesis, ya que se aplican a modelos de sistema diferentes al utilizado aquí.

Analizando las propuestas estudiadas es posible encontrar algunas similitudes entre ellas. En particular, algunas propuestas coinciden al integrar los algoritmos

de planificación como módulos *independientes* del núcleo, y al definir alguna *interfaz* para la interacción entre los componentes del sistema, a saber: núcleo, políticas de planificación y aplicaciones. Sin embargo, la manera de implementar cada nueva política de planificación es diversa, pues algunos autores proponen que se integren como threads, mientras que otros proponen que se implementen como módulos cargados del núcleo. También, hemos podido comprobar una gran diversidad en cuanto a las funciones que forman la interfaz de cada propuesta. Además, existen discrepancias en cuanto a la estructura del núcleo del sistema operativo, pudiendo ser un despachador o algún planificador con un algoritmo específico.

El *modelo para la planificación definida por el usuario*, como se comprobará a partir de la siguiente sección, presenta muchas ventajas en comparación con las propuestas estudiadas. Por una parte, facilita la portabilidad de las aplicaciones al ser definido de manera compatible con el modelo de planificación de POSIX. Su implementación es directa en núcleos POSIX. Por otra parte, propone una interfaz mucho más extensa y poderosa. Si bien es cierto que el modelo puede mejorarse, como se estudiará más adelante, representa una excelente alternativa para agregar nuevos servicios de planificación a un sistema operativo. A continuación se estudiará con detalle el modelo propuesto, su evolución, su implementación en RTLinux, así como las extensiones propuestas.

4.9. Planificación Definida por el Usuario Compatible con POSIX

La implementación de sistemas de tiempo-real utilizando algún estándar y en particular el estándar POSIX tiene muchísimas ventajas, ya que se dan garantías de portabilidad y está presente en muchos sistemas operativos de tiempo-real. Sin embargo, para el caso de la planificación de tareas nos encontramos con que POSIX ofrece tan sólo planificación basada en prioridades estáticas. El modelo para la planificación definida por el usuario permite agregar diferentes políticas de planificación a un SOTR y ha sido propuesto ante el PASC SSWG-RT para su consideración como futuro estándar POSIX [5]. También se ha propuesto para que sea integrado en el API de ADA [8] y ha sido implementado en RTLinux [111] [38] y MaRTE OS [4]. Antes de presentar el modelo, se estudiará el funcionamiento del planificador de un sistema operativo.

4.9.1. El planificador de un sistema operativo

Un sistema operativo puede ser definido como un sistema informático que administra los recursos de un ordenador, y que además actúa como interfaz entre

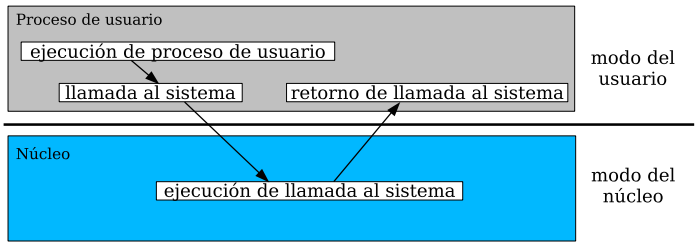

Figura 4.9: Transición de *modo del usuario* a *modo del núcleo*

los usuarios y los recursos del ordenador [101]. Los recursos que un sistema operativo administra pueden ser: los dispositivos de Entrada/Salida, los ficheros, la memoria, procesadores, procesos y tareas, etc. La gran mayoría de los sistemas operativos modernos utilizan un mecanismo de operación *basado en interrupciones*[27]. Si no existen procesos en ejecución, o dispositivos de Entrada/Salida por atender, ni usuarios a quienes responder, entonces el sistema operativo se mantiene en un estado de *inactividad*, a la espera de algún evento. Generalmente los eventos ocurren como *interrupciones*, ya sea por la generación de una señal o excepción. Una *excepción* es una interrupción generada por software y puede ser causada por un error, cómo por ejemplo al intentar efectuar una división por cero. También puede generarse una excepción cuando un programa de un usuario hace una solicitud de un servicio específico al sistema operativo. Estas solicitudes se hacen a través de un mecanismo llamado *llamada al sistema*[28]. En el sistema operativo existen rutinas específicas para atender cada tipo de interrupción y que son ejecutadas por el núcleo ya que requieren del uso de instrucciones especiales llamadas *instrucciones privilegiadas*. Debido a que las instrucciones privilegiadas tienen acceso al hardware del sistema, o a áreas restringidas de memoria, si no son ejecutadas correctamente pueden afectar el funcionamiento de otros procesos diferentes al que hizo la llamada al sistema.

Para garantizar un correcto funcionamiento del sistema operativo, existen dos modos de operación, dependiendo de si se ejecuta código del usuario o código del núcleo. Cuando se ejecuta alguna aplicación de usuario, el sistema está en *modo del usuario*. Por otra parte, cuando se ejecuta código del núcleo del sistema operativo, se está en *modo del núcleo*, también llamado *modo privilegiado*. Si una aplicación del usuario requiere la ejecución de código que es parte del núcleo, hace su solicitud a través de una *llamada al sistema*, como se ilustra en la Figura 4.9. Una vez que el núcleo ejecuta la función solicitada, retorna al sistema a *modo del usuario*, para que la aplicación continúe con su ejecución. De esta forma, las

[27] interrupt driven
[28] system call

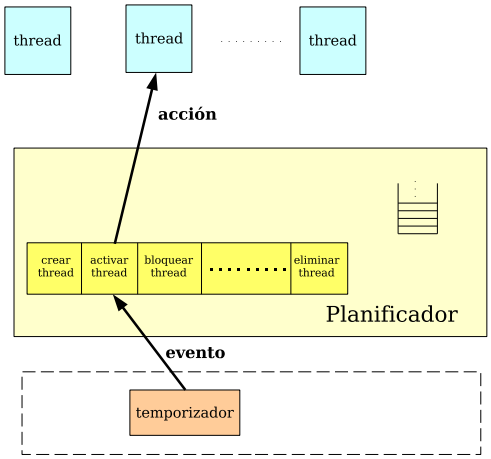

Figura 4.10: Abstracción de un planificador

aplicaciones de los usuarios no ejecutan directamente instrucciones privilegiadas y su ejecución es controlada por el sistema operativo, de tal manera que no afecte el funcionamiento del sistema, o que lo afecte lo menos posible.

El planificador, como se ha estudiado ya, es un módulo del sistema operativo que determina la asignación de recursos entre los procesos o tareas activas en un sistema, y que controla el acceso a los recursos compartidos entre las tareas, como se muestra en la Figura 2.1 en la página 10. En un sistema operativo de tiempo-real, los algoritmos implementados en el planificador deben garantizar el cumplimiento de las restricciones temporales de las tareas. Por ser parte del núcleo, su operación está también *basada en interrupciones.*

La Figura 4.10 muestra una abstracción de lo que es un planificador. Durante el tiempo de ejecución de un sistema, formado por un conjunto de tareas y recursos, se llevan a cabo solicitudes o interrupciones al planificador cuando se generan *eventos* de planificación. Los eventos pueden ser generados por la expiración de un temporizador, por ejemplo, armado por el mismo planificador, o el evento puede ser generado por algún elemento externo, como un thread o proceso, a través de una *llamada al sistema*. La expiración del temporizador puede generar una señal que signifique que una tarea debe ser activada. En tal caso, se informa al planificador para que lleve a cabo la *acción* correspondiente. Internamente, el planificador contiene un conjunto de sub módulos para atender cada evento específico. En el caso de la tarea que se activa, el planificador pudiera agregar a la tarea recién activada a la lista de tareas activas, que quizás esté ordenada por prioridad, para posteriormente seleccionar a la tarea más prioritaria y asignarle el procesador. Puede decirse que un planificador lleva a cabo *acciones* en respuesta a *eventos* de planificación.

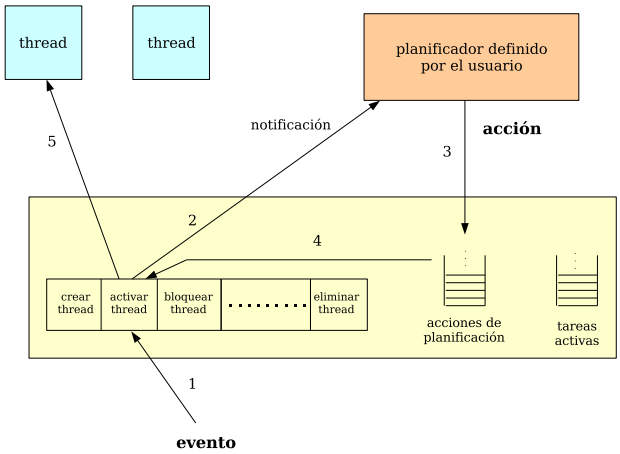

Figura 4.11: Eventos y acciones en la planificación definida por el usuario

4.9.2. Propuesta inicial

Tomando en consideración la abstracción de un planificador, el modelo de la planificación definida por el usuario propone que el sistema operativo mantenga internamente el planificador POSIX basado en prioridades fijas, y que se modifique para que sea capaz de identificar cuando un evento de planificación está dirigido a él, o está dirigido a un planificador definido por el usuario. Si el evento está dirigido a un planificador definido por el usuario, se le notifica para que lleve a cabo las acciones necesarias, y para que determine las acciones que deberá ejecutar el planificador del sistema operativo en caso de ser necesario. En la Figura 4.11 se muestra un ejemplo de un evento solicitando la activación de un thread planificado por un planificador definido por el usuario. El planificador del sistema operativo recibe el evento y detecta que debe notificar al correspondiente planificador para que decida el conjunto de acciones que deben ejecutarse. El planificador puede insertar el conjunto de acciones en una cola, para que posteriormente el planificador del sistema operativo las ejecute.

La propuesta inicial del modelo de la planificación definida por el usuario especifica que los algoritmos de planificación sean implementados en los threads, con la finalidad de proteger al sistema de un fallo causado por un planificador que contenga errores. En otras palabras, los planificadores son implementados como threads que se ejecutan en modo del usuario y se comunican con el núcleo por medio de un conjunto de llamadas al sistema proporcionadas por el modelo. El núcleo, a su vez, notifica al thread planificador cuando un evento de planificación se lleva a cabo. El núcleo notifica al planificador cuando algunos de los siguientes eventos de planificación ocurren:

Código del evento	Descripción	Inf. Adicional
POSIX_APPSCHED_NEW	Un nuevo thread ha solicitado ser asignado a un planificador	No
POSIX_APPSCHED_TERMINATE	Un thread ha terminado	No
POSIX_APPSCHED_READY	Un thread se ha activado	No
POSIX_APPSCHED_BLOCK	un thread se ha bloqueado	No
POSIX_APPSCHED_YIELD	Un thread ha cedido la CPU	No
POSIX_APPSCHED_SIGNAL	Una señal -ha sido aceptada por el thread planificador	Información de la señal
POSIX_APPSCHED_CHANGE_SCHED_PARAM	Un thread ha cambiado sus parámetros de planificación	No
POSIX_APPSCHED_EXPLICIT_CALL	Un thread ha invocado explícitamente a su planificador	Mensaje
POSIX_APPSCHED_TIMEOUT	Un temporizador ha expirado	No
POSIX_APPSCHED_PRIORITY_INHERIT	Un thread ha heredado un valor de prioridad debido al uso de un monitor	Prioridad heredada
POSIX_APPSCHED_PRIORITY_UNHERIT	Un thread ha desheredado un valor de prioridad	Prioridad heredada
POSIX_APPSCHED_INIT_MUTEX	Un thread ha solicitado la inicialización de un monitor planificado por el usuario	Puntero al monitor
POSIX_APPSCHED_DESTROY_MUTEX	Un thread ha solicitado la destrucción de un monitor	Puntero al monitor
POSIX_APPSCHED_LOCK_MUTEX	Un thread ha invocado una operación *lock* a un monitor	Puntero al monitor
POSIX_APPSCHED_TRY_LOCK_MUTEX	Un thread ha invocado una operación *try lock* a un monitor	Puntero al monitor
POSIX_APPSCHED_UNLOCK_MUTEX	Un thread ha invocado una operación *unlock* a un monitor	Puntero al monitor
POSIX_APPSCHED_BLOCK_AT_MUTEX	Un thread se ha bloqueado en espera de un monitor	Puntero al monitor
POSIX_APPSCHED_CHANGE_MUTEX_SCHED_PARAM	Un thread ha cambiado los parámetros de planificación de un monitor	Puntero al monitor

Cuadro 4.2: Eventos de la planificación definida por el usuario

- cuando un thread solicita ser *asignado a un planificador*
- cuando un thread se *bloquea* o *activa*
- cuando un thread invoca a la operación *yield*
- cuando un thread *invoca explícitamente a su planificador*
- cuando un thread *hereda* o *deshereda* un valor de *prioridad*, debido a la utilización de un *monitor*[29]
- cuando un thread efectúa una *operación en un monitor* planificado por un planificador definido por el usuario

El planificador debe ser capaz de detectar si alguna de las situaciones arriba descritas son experimentadas por alguno de los threads del sistema, y generar el evento correspondiente al planificador definido por el usuario que planifique al thread afectado.

Como es probable que el thread planificador no pueda procesar los eventos inmediatamente, estos deben ser almacenados en una cola ordenada por una política *Primero en Entrar Primero en Salir* (*FIFO*), hasta que puedan ser procesados por el correspondiente planificador. Los eventos almacenados deben contener al menos la siguiente información:

- el código del evento
- el identificador del thread que causó el evento
- información adicional relacionada con el evento, por ejemplo:
  - prioridad heredada o desheredada
  - información relacionada con una señal
  - el identificador de un monitor
  - información específica del planificador

El Cuadro 4.2 muestra la lista completa de eventos que pueden ser notificados a un planificador definido por el usuario, así como su descripción e información adicional que debe enviarse con el evento.

Cuando el planificador procesa los eventos almacenados en la cola, puede ejecutar una o varias acciones de planificación, que deben ser almacenadas en un

[29] mutex

objeto de *tipo opaco*[30] para su posterior ejecución, ya que al igual que los eventos no existen garantías de que las acciones se ejecuten inmediatamente. Las acciones que puede ejecutar un planificador definido por el usuario son:

- aceptar o rechazar a un thread que ha solicitado ser asignado a un planificador
- activar o suspender un thread
- aceptar o rechazar la inicialización de un monitor planificado por el usuario
- conceder el bloqueo de un monitor planificado por el usuario

El modelo incluye un conjunto de funciones que son utilizadas para indicar las acciones de planificación que deben ser ejecutadas. Las acciones se especifican, conservando su orden, en un objeto de *tipo opaco* que en el modelo se define como tipo *posix_appsched_actions_t* (definido en <*sched.h*>), que es incluido como parámetro en todas las funciones que a continuación se describen:

```
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_addtimedactivation(
      posix_appsched_actions_t *sched_actions,
      pthread_t thread);

int posix_appsched_actions_addsuspend(
      posix_appsched_actions_t *sched_actions,
      pthread_t thread);
```

[30] su implementación no es visible para el usuario

```
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 continuación se explica cada una de estas funciones:

- *posix_appsched_actions_init*: el objeto especificado por s*ched_actions* se inicializa, de tal manera que no contenga acciones por ejecutar. Después de su inicialización, el número de acciones que contendrá el objeto será cero.
- *posix_appsched_actions_destroy*: el objeto especificado por *sched_actions* se destruye, de tal manera que quede sin inicializar. La única manera de que dicho objeto pueda ser utilizado es inicializándolo.
- *posix_appsched_actions_addaccept*: agrega la acción de aceptación de thread al objeto especificado por *sched_actions*, que servirá para notificar que *thread* ha sido aceptado por su thread planificador.
- *posix_appsched_actions_addreject*: agrega la acción de rechazo de thread al objeto especificado por *sched_actions*, que servirá para notificar que *thread* ha sido rechazado por su thread planificador.
- *posix_appsched_actions_addtimedactivation*: agrega la acción de activación de thread al objeto especificado por *sched_actions*, que servirá para notificar que *thread* ha sido activado para ejecución por su thread planificador.
- *posix_appsched_actions_addsuspend*: agrega la acción de suspensión de thread al objeto especificado por *sched_actions*, que servirá para notificar que la ejecución de *thread* ha sido suspendida por su thread planificador.
- *posix_appsched_actions_addacceptmutex*: agrega la acción de aceptación de monitor al objeto especificado por *sched_actions*, que servirá para notificar que el monitor identificado por *mutex* ha sido aceptado su thread planificador.

- *posix_appsched_actions_addrejectmutex*: agrega la acción de rechazo de monitor al objeto especificado por *sched_actions*, que servirá para notificar que el monitor identificado por *mutex* ha sido rechazado su thread planificador.

- *posix_appsched_actions_addlocktmutex*: agrega la acción de bloqueo de monitor al objeto especificado por *sched_actions*, que servirá para notificar que el bloqueo en el monitor identificado por *mutex* ha sido otorgado a *thread* por su thread planificador.

El conjunto de acciones es determinado por el thread planificador y reportado al sistema por medio de la función *posix_appsched_execute_actions()*. Esta función es la más importante en la interfaz propuesta. Después de que el planificador ha sido notificado de algún evento de planificación, determina el conjunto de acciones que deben ejecutarse y las almacena en orden en el objeto *sched_actions*. Sin embargo, las acciones no son ejecutadas sino hasta cuando se invoca a la función *posix_appsched_execute_actions()*. Con ella, el planificador puede ejecutar la lista de acciones para suspenderse posteriormente en espera de que el siguiente evento de planificación sea reportado al sistema. El prototipo de la función es el siguiente, en la que el parámetro *sched_actions*, como se ha mencionado, contiene la lista de acciones por ejecutar:

```
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);
```

Una vez que la lista de acciones se ejecuta, el planificador se suspende hasta que sea activado nuevamente por la ocurrencia de eventos de planificación. Pero también es posible programar la siguiente activación del planificador, aunque no existan eventos que lo activen. Una manera de hacerlo es programando un *temporizador*[31] al siguiente instante deseado de activación del planificador. En este caso, el parámetro *timeout* contendrá el tiempo en el que el temporizador expirará y activará al planificador. Además, el evento generado deberá contener el código POSIX_APPSCHED_TIMEOUT como parte del parámetro *event*. Otra manera de hacerlo consiste en programar un *temporizador POSIX*[32] que al expirar genere una señal, dirigida al planificador, indicando su activación. En este caso, la

[31] timeout
[32] POSIX timer

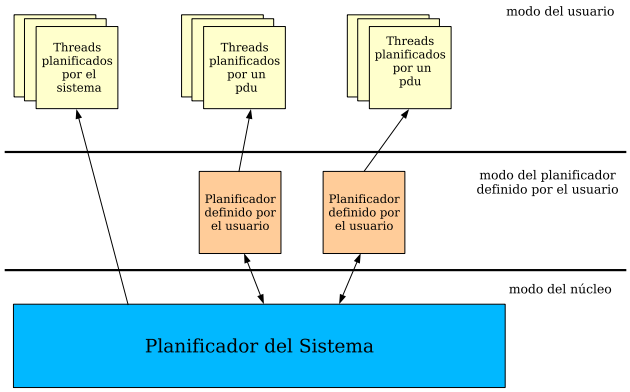

Figura 4.12: Modelo para la Planificación Definida por el Usuario

señal debe pertenecer a *set* y el evento generado deberá contener el código POSIX_APPSCHED_SIGNAL como parte del parámetro *event*.

En el parámetro *current_time*, la función retornará el valor del tiempo de acuerdo al reloj especificado en el atributo *clockid* del planificador, como se verá más adelante, al estudiar la implementación del planificador definido por el usuario.

Hasta el momento se ha dado una descripción general del modelo para la planificación definida por el usuario. Una descripción más detallada está representada por la Figura 4.12. La estructura de planificación es del orden jerárquica, manteniendo al *planificador basado en prioridades fijas de POSIX* en el nivel más básico y colocando sobre él a los planificadores definidos por el usuario. Se mencionó que el modelo propone que los planificadores se definan como threads que se ejecuten en modo del usuario. Sin embargo, el modelo no es restrictivo y permite la ejecución de las políticas de planificación en el modo del núcleo. Por esta razón, ya que el modo de ejecución de los planificadores puede ser de usuario, de núcleo, o una combinación de ambos, al modo de ejecución de los algoritmos de planificación se le denomina en el modelo como *modo del planificador definido por el usuario*.

De acuerdo a lo mostrado en la Figura 4.12, puede observarse que el modelo hace una clasificación de los threads de la siguiente manera:

- *Threads planificados por el sistema*[33]: Son aquellos threads planificados directamente por el *planificador basado en prioridades fijas* del sistema operativo. Son llamados también threads *regulares*. Un planificador definido por el usuario puede ser a su vez un thread planificado por el sistema.

[33] system-scheduled threads

- *Threads planificados-por-aplicación*[34]: Son los threads planificados por algún planificador definido por el usuario.
- *Threads planificadores*[35]: Son threads que planifican a un conjunto de threads *planificados-por-aplicación*.

Para indicar que un thread es *planificado-por-aplicación*, un nueva política de planificación ha sido definida y propuesta para ser integrada al estándar POSIX, llamada SCHED_APP. Los planificadores, por otra parte, deben definirse como threads bajo la política SCHD_FIFO. Para especificar cuál es el planificador de un thread *planificado-por-aplicación*, se utilizan las funciones que tienen el siguiente prototipo:

```
int pthread_attr_setappscheduler(
     pthread_attr_t *attr,
     pthread_t scheduler);

int pthread_setappscheduler(
     pthread_t thread,
     pthread_t scheduler);
```

Por otra parte, para especificar si un thread es un thread planificador o es un thread *planificado-por-aplicación*, la función que a continuación se muestra debe ser utilizada:

```
int pthread_attr_setappschedulerstate(
     pthread_attr_t *attr,
     int appschedstate);
```

en donde appschedulerstate puede tener alguno de los siguientes valores:

- *PTHREAD_REGULAR*: indica que es un thread *planificado-por-aplicación*
- *PTHREAD_APPSCHEDULER*: indica que es un *thread planificador*

Si un thread es definido como thread planificador, una vez que ha sido creado pueden establecerse algunas de sus propiedades:

- *temporizador*[36] *soportado*: relativo o absoluto

[34] application-scheduled threads
[35] application-scheduler threads
[36] *timeout*

- *reloj utilizado*: reloj usado para armar temporizadores[37] POSIX
- *máscara de eventos*: usada para determinar que eventos serán filtrados por el sistema y por lo tanto no serán notificados al planificador

Para definir los atributos del planificador, la interfaz proporciona las siguientes funciones:

```
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_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);

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);
```

El primer grupo de funciones arriba descritas permiten establecer y conocer el valor del atributo *clock_id* del planificador. Este tipo de reloj es el utilizado al

[37] timers

determinar el valor de *current_time* y el parámetro *timeout*, que retorna y usa, respectivamente, la función *posix_appsched_execute_actions()*. El valor por defecto de *clock_id* es CLOCK_REALTIME.

El segundo grupo de funciones permiten establecer y conocer el valor del atributo *flags* del thread planificador. Ningún valor es definido por defecto, lo que significa que el parámetro *timeout* representa un valor *relativo* del intervalo tiempo, medido con el reloj especificado en *clock_id*. Por otra parte, si el valor de *flags* es igual a POSIX_APPSCHED_ABSTIMEOUT, significa que el parámetro *timeout* representará el tiempo *absoluto*.

El tercer grupo de funciones permiten agregar, eliminar y conocer los eventos de la *máscara de eventos* del planificador, que serán filtrados y por lo tanto no serán notificados al planificador. El cuarto grupo de funciones permiten establecer y conocer el valor de la máscara actualmente utilizada por el planificador.

Finalmente, el último grupo de funciones permiten establecer y conocer el valor del atributo *reply* del thread planificador. Con este atributo, un planificador puede enviar información a un thread *planificado-por-aplicación* que la haya solicitado a través de la función *posix_appscched_invoke_withdata()*, que se estudiará más adelante, cuando se presente la interfaz para threads *planificados-por-aplicación*.

De la misma manera que los threads, los monitores del sistema pueden ser clasificados en diferentes categorías:

- *Monitores planificados por el sistema*: Son creados utilizando los protocolos de sincronización definidos en POSIX: sin herencia de prioridad[38] , techo de prioridad inmediato[39], o herencia de prioridad básico[40]. Estos monitores podrán ser utilizados para compartir recursos entre los planificadores definidos por el usuario, entre conjuntos de threads *planificados-por-aplicación*, asignados a diferentes planificadores, e incluso entre un thread planificador y los threads que planifique.

- *Monitores planificados por aplicación*: Son monitores creados con el nuevo protocolo PTHREAD_APPSCHED_PROTOCOL, propuesto por el modelo. El comportamiento del protocolo es definido por el planificador, lo que significa que el protocolo de acceso a los recursos es definido por el usuario. El núcleo notificará al planificador correspondiente cuando se haga una solicitud de bloqueo (lock) en el monitor, una operación de liberación del bloqueo (unlock), o cuando algún thread se bloquee en espera de que el monitor sea liberado. Cuando se solicita entrar en una sección crítica, el planificador decidirá si se asigna o no el recurso al thread solicitante.

[38] PTHREAD_PRIO_NONE
[39] PTHREAD_PRIO_PROTECT
[40] PTHREAD_PRIO_INHERIT

El modelo también ha definido una interfaz para los monitores *planificados-por-aplicación*. El conjunto de funciones que la integran permiten definir y conocer al thread que planifica al monitor, y también definir y conocer sus parámetros de planificación definidos por el usuario. Las funciones se muestran a continuación:

```
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);
```

También, existen funciones para que el monitor pueda almacenar y recuperar información específica y que utilice al implementar su política de planificación. Los prototipos de las funciones son los siguientes:

```
int posix_appsched_mutex_setspecific(
       pthread_mutex_t *mutex,
       const void * value);

int posix_appsched_mutex_getspecific(
       pthread_mutex_t *mutex,
       void ** value);
```

el parámetro *value* contiene generalmente un puntero a un área de memoria que contiene la información específica del monitor.

La propuesta inicial del modelo incluye una interfaz extensa para threads planificadores y threads *planificados-por-aplicación*. Con la finalidad de evitar ser repetitivo, el resto de la interfaz será estudiada cuando se presente la nueva versión del modelo. Más adelante se presentará un ejemplo completo de la implementación de un planificador definido por el usuario utilizando la versión original del modelo. Sin embargo, para poder comprender mejor las diferencias esenciales

con el nuevo modelo, en el Ejemplo 1 se presenta del pseudocódigo de un planificador definido por el usuario. Es importante aclarar que no es la única manera de construirlo. En el ejemplo mostrado, el código es ejecutado por el thread planificador. Después de que el planificador ha sido creado y ha ejecutado sus operaciones de inicialización, se suspende a la espera de la llegada de eventos de notificación. Cuando algún evento ocurre, el thread planificador es activado para que determine las acciones que deben ejecutarse. Posteriormente, decide el siguiente thread a ejecutar de entre los threads que planifique, y ejecuta el conjunto de acciones previamente determinadas utilizando la función *posix_appsched_exeute_actions()*, para suspenderse de nuevo a la espera de un nuevo evento.

El modelo para la planificación definida por el usuario propuesto inicialmente ha sido implementada en RTLinux por Joseph Vidal, y los resultados publicados en [111]. Antes de estudiar algunos aspectos relacionados con esta implementación, se presentará una breve descripción de RTLinux.

4.9.2.1. RTLinux

RTLinux es una versión de Linux que proporciona características de *tiempo-real crítico* [115]. Con RTLinux es posible ejecutar tareas especiales de tiempo-real y manejadores de interrupciones en la misma máquina que ejecute Linux. De hecho, RTLinux funciona ejecutando el núcleo de Linux como si fuese una tarea más que se ejecuta en un sistema operativo de tiempo-real. Linux es la *tarea inactiva*[41] de RTLinux ya que se ejecuta cuando no hay tareas de tiempo-real por ejecutar. Linux no puede bloquear las interrupciones ni puede evitar que sea expulsado del procesador. RTLinux fue desarrollado por V. Yodaiken y M. Baravanov [14] y publicado con licencia GPL. En el año 2001 fue patentado bajo la patente *U.S. Patent No. 5,995,745* y su uso regido por licencia *RTLinux Open Patent license* [43]. A partir de esa fecha el desarrollo de la versión libre de RTLinux ha sido impulsado por la comunidad de software libre.

RTLinux es un sistema operativo en el cual un pequeño núcleo de tiempo-real coexiste con el núcleo POSIX de Linux. RTLinux depende de Linux para el proceso de arranque del sistema, de la mayoría de los manejadores de dispositivos, de los servicios de red, de la administración de ficheros, del control de procesos de Linux, y para cargar y descargar módulos del núcleo. Las aplicaciones de tiempo-real consisten en tareas que son cargadas como módulos. El sistema ofrece las prestaciones de un sistema operativo de propósito general, administrado por Linux, con características de tiempo-real, proporcionadas por RTLinux.

Para garantizar el cumplimiento de las restricciones temporales de las tareas críticas de una aplicación, algunas modificaciones se hacen a Linux, ya que Li-

[41] idle task

Ejemplo 1 Pseudocódigo de un planificador definido por el usuario

```
void *planificador_def_por_el_usuario (void *arg)
{
...
operaciones de inicialización;
...
while (1) {

   determinar thread por ejecutar,
   si existen threads activos;

     //  ejecutar acciones de planificación y
     //  suspender al planificador hasta la llegada
     //  de un nuevo evento

   posix_appsched_execute_actions();

     // procesar evento que es
     // notificado el planificador

   switch (sched_event.event_code) {

      case POSIX_APPSCHED_NEW:
         // procesar nuevo thread

     case POSIX_APPSCHED_EXPLICIT_CALL:
         // procesar llamada explícita del thread
         // que significa probablemente que ha
         // concluido su actual activación

      case POSIX_APPSCHED_TERMINATE
         // procesar terminación de thread

      case POSIX_APPSCHED_SIGNAL
         // procesar señal recibida por el planificador
      ....
   }  // fin switch
}  // fin while

} // fin función
```

nux no es un sistema operativo de tiempo-real. Su objetivo consiste en asignar de manera equitativa los recursos del sistema entre las tareas existentes, de tal manera que se proporcione un buen tiempo promedio de respuesta. Sin embargo, ese enfoque no es apropiado en un sistema de tiempo-real, en el que las tareas críticas deben cumplir sus plazos aún en perjuicio de las otras tareas presentes en el sistema.

Uno de los problemas que afectan el cumplimiento de los requerimientos temporales del sistema es el manejo de las interrupciones de Linux. Si cuando se ejecuta una tarea crítica el sistema operativo permite que se ejecute una interrupción, se introduce incertidumbre en el tiempo de respuesta, lo que es intolerable en un sistema operativo de tiempo-real. En RTLinux, este problema se resuelve al introducir una *capa de emulación de software* entre el núcleo de Linux y el controlador de interrupciones de hardware. El enfoque es similar al propuesto previamente por Stodolsky *et. al* en [108]. Todas las interrupciones son inicialmente manejadas por el núcleo de RTLinux, evitando así que interfieran con la ejecución de las tareas críticas y proporcionando certidumbre en los tiempos de respuesta. En RTLinux, las interrupciones pueden estar *habilitadas* o *inhabilitadas*, y en ambos casos la capa de emulación las maneja. Si las interrupciones se han habilitado, se informan a la tarea Linux únicamente cuando no existen tareas de tiempo-real por ejecutar. Por otra parte, si las interrupciones están inhabilitadas, la capa de emulación almacena en una cola cada interrupción que llega al sistema para notificarlas a Linux cuando las interrupciones se habilitan nuevamente. La comunicación entre RTLinux y Linux se lleva a cabo a través de colas (FIFO) y memoria compartida. La capa de emulación de software que administra el manejo de interrupciones se agrega al núcleo de Linux sin necesidad de introducir grandes cambios en él.

A pesar de que inicialmente se diseñó RTLinux para que las tareas de tiempo-real se ejecutaran en su propio espacio de direcciones para proporcionar protección de memoria, actualmente se ejecutan en modo del núcleo por razones de eficiencia. Las aplicaciones de tiempo-real se cargan como módulos del núcleo, como se mencionó previamente.

4.9.2.2. Planificación definida por el usuario en RTLinux.

La elección de RTLinux para implementar en él la planificación definida por el usuario ofrece muchas ventajas. Debido a que tanto Linux como RTLinux se distribuyen con licencia GPL, es posible estudiar y modificar su estructura. Esta ventaja no esta presente en cualquier sistema operativo. Por otra parte, RTLinux es un sistema operativo de tiempo-real, y a diferencia de un sistema operativo de propósito general, los servicios que garantizan el cumplimiento de las restricciones temporales de las tareas críticas ya están implementados. Además, la posibilidad de cargar y descargar dinámicamente módulos del núcleo permite la ejecución de

aplicaciones de tiempo-real sea eficiente.

La implementación del modelo original de la planificación definida por el usuario en RTLinux muestra muy buenos resultados, ya que según se reporta en [111], la sobrecarga introducida al sistema es de 2,32 %.

4.9.3. Nueva propuesta

La propuesta inicial del modelo para la planificación definida por el usuario ha sido diseñada detalladamente, ofrece una interfaz extensa, un conjunto de mecanismos que permiten la implementación de algoritmos de planificación complejos, y permite la portabilidad de las aplicaciones. Con ella, se tiene una excelente alternativa para integrar de nuevas políticas de planificación a un sistema operativo. Sin embargo, la propuesta inicial es susceptible de mejoras.

En [4], Mario Aldea y Michael González-Harbour propusieron un nuevo y más general enfoque del modelo para la planificación definida por el usuario. Los cambios más importantes en el nuevo modelo tienen que ver con la definición del planificador y la manera de implementarlo. Además, se agrega un nuevo atributo a los threads *planificados-por-aplicación*, llamado *urgencia*, para utilizar la cola de tareas activas del planificador del núcleo de manera más eficiente. También, se integra la política *SRP* al modelo para que la sincronización de tareas se haga de manera transparente. A continuación se estudiará el nuevo modelo.

4.9.3.1. El planificador como objeto abstracto

En el modelo propuesto inicialmente [4], cada planificador definido por el usuario se implementa como un thread especial. Sin embargo, con esta estrategia se incurre en múltiples cambios de contexto cada vez que se necesita tomar alguna decisión relacionada con el planificador. Esta restricción no está justificada del todo ya que en una arquitectura particular, como el caso de RTLinux, los threads se ejecutan en el modo del núcleo, por lo que un error en el thread planificador puede afectar a todo el sistema. Si se recuerda, la motivación para definir a los planificadores como threads era la posibilidad de ejecutarlos en modo del usuario y evitar que afectaran la ejecución de otras aplicaciones. En la nueva definición del modelo el planificador es definido como un *objeto abstracto* y no tiene que ser implementado necesariamente como un thread. El planificador es tan sólo definido como un módulo de software y no se impone una estrategia de implementación particular.

Independientemente de como se implemente al planificador definido por el usuario, este contiene un conjunto de operaciones que son invocadas por el sistema operativo siempre que alguno de los threads planificados por él ejecuten o sean afectados por algún evento de planificación. Esto significa que un planificador se

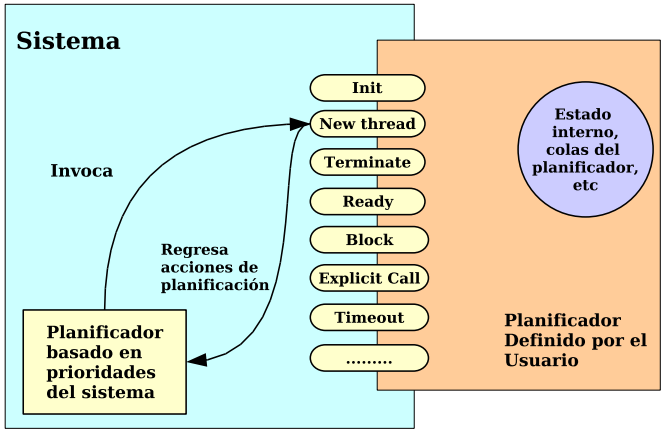

Figura 4.13: Estructura de un Planificador Definido por el Usuario

Operaciones del Planificador
init (...)
new_thread (...)
thread_terminate (...)
thread_ready (...)
thread_block (...)
thread_yield (...)
change_sched_param_thread (...)
explicit_call (...)
explicit_call_with_data (...)
notification_for_thread (...)
timeout (...)
signal (...)
priority_inherit (...)
priority_uninherit (...)
appsched_error (..)

Cuadro 4.3: Operaciones de un planificador definido por el usuario

define como un *objeto abstracto* que está compuesto por un conjunto de *operaciones primitivas*. Las operaciones del planificador son ejecutadas en respuesta a eventos de planificación. Por cada *evento* de planificación de interés para el modelo se ha definido su correspondiente *operación* en el planificador. Cada operación es en realidad una función, creada por el desarrollador del planificador, para implementar la política de planificación. En la Figura 4.13 se muestra la estructura de un *PDU*[42] con su conjunto de operaciones. Estas *operaciones primitivas* o funciones definidas por el implementador se pasan al *PDU*[43] cuando es creado, utilizando una estructura llamada `posix_appsched_scheduler_ops_t` y que contiene los punteros a las operaciones primitivas, definidas en un mecanismo similar al utilizado por UNIX para administrar sus manejadores de dispositivos. El conjunto de operaciones primitivas que forman la estructura se presentan a continuación y también de forma resumida en el Cuadro 4.3:

```
typedef struct {
   void (*init) (void * sched_data, void * arg);

   void (*new_thread) (
            void * sched_data,
            pthread_t thread,
            posix_appsched_actions_t * actions,
            struct timespec *current_time);

   void (*thread_terminate) (
            void * sched_data,
            pthread_t thread,
            posix_appsched_actions_t * actions,
            struct timespec *current_time);

   void (*thread_ready) (
            void * sched_data,
            pthread_t thread,
            posix_appsched_actions_t * actions,
            struct timespec *current_time);

   void (*thread_block) (
            void * sched_data,
            pthread_t thread,
            posix_appsched_actions_t * actions,
```

[42] planificador definido por el usuario
[43] planificador definido por el usuario

```
            struct timespec *current_time);

    void (*thread_yield) (
            void * sched_data,
            pthread_t thread,
            posix_appsched_actions_t * actions,
            struct timespec *current_time);

    void (*change_sched_param_thread) (
            void * sched_data,
            pthread_t thread,
            posix_appsched_actions_t * actions,
            struct timespec *current_time);

    void (*explicit_call) (
            void * sched_data,
            pthread_t thread,
            int user_event_code,
            posix_appsched_actions_t * actions,
            struct timespec *current_time);

    void (*explicit_call_with_data) (
            void * sched_data,
            pthread_t thread,
            const void * msg,
            size_t msg_size,
            void *reply, size_t *reply_size,
            posix_appsched_actions_t * actions,
            struct timespec *current_time);

    void (*notification_for_thread) (
           void * sched_data,
           pthread_t thread,
           posix_appsched_actions_t * actions,
           struct timespec *current_time);

    void (*timeout) (
           void * sched_data,
           posix_appsched_actions_t * actions,
           struct timespec *current_time);
```

```
        void (*signal) (
                void * sched_data,
                siginfo_t siginfo,
                posix_appsched_actions_t * actions,
                struct timespec *current_time);

        void (*priority_inherit) (
                void * sched_data,
                pthread_t thread,
                int sched_priority,
                posix_appsched_actions_t * actions,
                struct timespec *current_time);

        void (*priority_uninherit) (
                void * sched_data,
                pthread_t thread,
                int sched_priority,
                posix_appsched_actions_t * actions,
                struct timespec *current_time);

        void (*appsched_error) (
                void * sched_data,
                pthread_t thread,
                posix_appsched_error_cause_t cause,
                posix_appsched_actions_t * actions);

    } posix_appsched_scheduler_ops_t;
```

Puede observarse que el conjunto de operaciones del planificador es muy similar al conjunto de eventos definidos en el modelo inicial, y que fueron presentados previamente en el Cuadro 4.2 en la página 77. Las operaciones relacionadas con el manejo de monitores no han sido incluidas en las operaciones del planificador. Las razones e implicaciones de esto se discutirán más adelante. Por otra parte, nuevos eventos han sido definidos. Para manejar estos eventos, dos operaciones han sido incorporadas al modelo: *explicit_call_with_data* y *notification_for_thread*. La operación *explicit_call_with_data* es notificada al planificador cuando alguno de sus threads invoca a la función *posix_appsched_invoke_withdata()*, que se estudiará posteriormente. La segunda operación, *notification_for_thread*, se ejecuta cuando una notificación de un evento temporal, asociada con un thread específico, es informada al planificador.

Cuando ocurre algún evento de planificación que afecte a alguno de los threads

planificados-por-aplicación, es reportado a su *PDU* al ejecutar la correspondiente operación primitiva o función. Esto permite que el *PDU* decida cuál es la acción de planificación que debe ejecutarse de entre las que se muestran en el Cuadro 4.4 y de las cuales se muestra a continuación el prototipo :

```
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_addtimedactivation (
          posix_appsched_actions_t *sched_actions,
          pthread_t thread,
          posix_appsched_urgency_t urgency);

int posix_appsched_actions_addtimedactivation (
          posix_appsched_actions_t *sched_actions,
          pthread_t thread,
          posix_appsched_urgency_t urgency,
          const struct timespec *at_time);

int posix_appsched_actions_addsuspend (
          posix_appsched_actions_t *sched_actions,
          pthread_t thread);

int posix_appsched_actions_addtimeout (
          posix_appsched_actions_t *sched_actions,
          const posix_appsched_scheduler_id_t
                              * scheduler_id,
          clockid_t clock_id,  int flags,
          const struct timespec *at_time);

int posix_appsched_actions_addthreadnotification (
          posix_appsched_actions_t *sched_actions,
          pthread_t thread,
          const struct timespec *at_time);
```

Por medio del uso de estas funciones, el *PDU* puede activar o suspender a los threads que planifica, armar temporizadores o notificaciones con temporizadores

asociadas con algún thread, entre otras acciones. El conjunto de acciones del nuevo modelo, comparado con el definido en la propuesta inicial y presentado en la página 79 muestra algunas diferencias. Las acciones relacionadas con el manejo de monitores ya no existen y nuevas acciones han sido definidas. A continuación se explican las funciones relacionadas con las nuevas acciones de planificación:

- *posix_appsched_actions_addtimedtimedactivation*: agrega la acción de activación de thread al objeto especificado por *sched_actions*, que servirá para notificar que *thread* debe ser activado para ejecución por su thread planificador, en el instante de tiempo especificado por *at_time*.

- *posix_appsched_actions_addtimout*: agrega la acción de activación del planificador al objeto especificado por *sched_actions*, que servirá para notificar que el planificador debe ser activado en el instante de tiempo especificado por *at_time*.

- *posix_appsched_actions_addthreadnotification*: agrega la acción de notificación asociada a un thread al objeto especificado por *sched_actions*, que servirá para notificar que al planificador un evento asociado con *thread* en el instante de tiempo especificado por *at_time*.

La integración de estas funciones en la interfaz aporta flexibilidad y mejora el desempeño. Por ejemplo, si se requiere programar notificaciones de eventos asociadas con threads, o si se desea programar futuras activaciones de threads, existen ahora funciones para hacerlo directamente. De otra manera, es posible programar este tipo de acciones utilizando el atributo *timeout* que era uno de los parámetros de la función *posix_appsched_execute_actions()*, pero para hacerlo es necesario que el planificador implemente una cola de eventos temporales, y que establezca el valor de *timeout* al más urgente de ellos. Con las nuevas funciones la implementación se simplifica.

En el modelo previo, la función *posix_appsched_execute_actions()* representaba la función más importante de la interfaz, ya que ejecutaba las acciones de planificación definidas por el planificador, como se muestra en el Ejemplo 1 en la página 88. Sin embargo, en la nueva definición del modelo, esta función ya no existe. Como se ha explicado, los eventos son notificados al planificador siguiendo un mecanismo *basado en interrupciones*. En el modelo previo, la ocurrencia de un evento implicaba la activación del *thread* planificador, para que procesara el evento y ejecutara las acciones correspondientes. Ahora, como el planificador no necesariamente es implementado como un thread, la ocurrencia del evento invoca la ejecución de la correspondiente función primitiva para que en su código se determinen las acciones por ejecutar. La forma en que se ejecuten las acciones de planificación dependerá de la manera en que este sea implementado. Si el planificador

no es un thread, la existencia de la función *posix_appsched_execute_actions()* no se justifica. La restricción que si que debe existir es que el sistema debe garantizar que la invocación de las operaciones primitivas y la ejecución de las acciones de planificación se haga de manera secuencial. Además, debe garantizarse que las operaciones del planificador se ejecuten antes que cualquier thread planificado por él. Esto implica que si la prioridad de algún thread cambia debido a que heredó una prioridad mayor al usar un monitor[44] se hace necesario que las operaciones de planificación hereden ésa prioridad. Si el planificador es un thread, deberá tener siempre una prioridad mayor o igual a los threads que planifique.

El objeto de *tipo opaco* llamado *sched_actions* en las funciones de acciones de planificación mostradas en la página 95 sigue siendo utilizado en el modelo. En él, el planificador almacenará el conjunto de acciones para que se ejecuten cuando sea posible.

4.9.3.2. Noción de urgencia

Los planificadores son responsables del orden en que los threads activos deben ser ejecutados. Debido a que el planificador existente en el núcleo del sistema operativo utiliza generalmente una política FIFO a cada nivel de prioridad fija, la implementación de algún algoritmo de planificación diferente al del núcleo imposibilitaba, el uso de la *cola de tareas activas* interna del SOTR. Para poder utilizar la *cola de tareas activas* del núcleo del sistema operativo se introdujo en el modelo la noción abstracta de *urgencia*, con la cual cualquier parámetro de planificación particular de alguna política pueda ser proyectado[45]. Los threads *planificados-por-aplicación* tienen un nuevo parámetro, llamado urgencia. Con este nuevo parámetro, en un mismo nivel de prioridad los threads *planificados-por-aplicación* no estarán ordenadas bajo la política FIFO sino en función del valor de urgencia, y que puede ser utilizado para expresar la prioridad de la política de planificación definida por el usuario. Por ejemplo, si se implementa un planificador EDF, el valor de la urgencia será mayor en la medida en que los plazos de los threads sean menores. El uso del nuevo parámetro permite en algunos casos evitar que el planificador tenga que utilizar internamente una cola para ordenar a sus threads activos, ya que podrá utilizar la cola del planificador del sistema.

A cada thread se le asignará un valor numérico particular de *urgencia*, que tendrá esta forma:

typedef long long **posix_appsched_urgency_t**

[44] mutex
[45] mapped

Acciones de Planificación
posix_appsched_actions_addaccept (...)
posix_appsched_actions_addreject (...)
posix_appsched_actions_addctivate (...)
posix_appsched_actions_addtimedactivation (...)
posix_appsched_actions_addssuspend (...)
posix_appsched_actions_addtimeout (...)
posix_appsched_actions_addthreadnotification (...)

Cuadro 4.4: Acciones de Planificación

de tal manera que pueda ser directamente proyectado a una variable de tipo *POSIX timepsec* o a cualquier variable que represente el tiempo en un SOTR, ya que en la mayoría de las arquitecturas tendrá un tamaño de 64 bits.

Cuando algún *PDU* necesite conocer el valor del parámetro de urgencia de alguno de sus threads, podrá hacerlo utilizando la función:

```
int posix_appsched_geturgency (pthread_t thread,
            posix_appsched_urgency_t *urgency);
```

Antes de presentar la integración de la política SRP al modelo, se estudiará su nueva interfaz. Algunos cambios se han hecho con respecto a la interfaz original, principalmente en las funciones para crear planificadores. A continuación se presenta detalladamente la interfaz del nuevo modelo.

4.9.3.3. Interfaz para los planificadores definidos por el usuario

Para crear o definir un planificador definido por el usuario debe utilizarse la función *posix_appsched_scheduler_create()*, cuyo prototipo es:

```
int posix_appsched_scheduler_create (
     const posix_appsched_scheduler_ops_t * scheduler_ops,
     size_t scheduler_data_size,
     void * arg, size_t arg_size,
     posix_appsched_scheduler_id_t *scheduler_id);
```

Los parámetros que recibe la función se explican a continuación:

- *scheduler_ops:* Es la estructura que contiene los punteros a las operaciones primitivas, presentada anteriormente. No es necesario definir todas las operaciones pues dependerá de la política de planificación y de su implementador la determinación de cuales deben definirse. Si alguna operación no se define, deberá pasarse un puntero a NULL en el lugar correspondiente.

- *scheduler_data_size*: Se utiliza para que el planificador definido por el usuario solicite al sistema una área de memoria del tamaño indicado por este parámetro. En la página 92 se muestran los prototipos de las funciones que forman la estructura `posix_appsched_scheduler_ops_t`. Puede observarse que el primer parámetro que reciben es `sched_data`, que es un puntero al área de memoria de tamaño `Scheduler_data_size` y que es utilizada por el *PDU* para almacenar su datos y que constituye además una porción de memoria compartida entre las operaciones de planificación.

- *arg*: Al momento de ser creado, un *PDU* puede recibir un conjunto de parámetros, como pueden ser el número de threads por planificar, su quantum, etc. En este parámetro se pasa un puntero al área de memoria de tamaño `arg_size` en el que se encuentra el conjunto de parámetros.

- *sched_id:* Identificador del *PDU*.

En el modelo anterior, el planificador se implementaba en el código de un thread. Antes de crearlo, usando la función *pthread_attr_setappschedulerstate()* se definía que sería un thread planificador, como se mostró en la página 83. Posteriormente, cuando el thread *planificador* se creaba con la función POSIX *pthread_create()*, se incluía entre sus parámetros un puntero a la función que implementaba la política de planificación. En la nueva definición del modelo, con la función para crear planificadores no se especifica algún thread que implemente el algoritmo de planificación, sino tan sólo se definen las operaciones y parámetros del planificador. Ahora, la política se implementa en el conjunto de funciones que forman las operaciones del planificador, y en cada implementación del modelo se determinará el mecanismo para ejecutarlas, que puede ser utilizando un thread especial que se activa en el nivel de prioridad apropiado, como en el modelo previo. Pero no es la única opción para implementarlo, como se discutirá más adelante. Lo importante a destacar aquí es que no se impone una estrategia particular para implementar al planificador.

4.9.3.4. Señales

POSIX [51] define que una señal es un mecanismo por medio del cual un proceso o thread puede ser notificado o afectado por algún evento que ocurre en el sistema. Cuando una señal se genera, puede ejecutarse la acción que haya sido definida para ella, o puede ser entregada a un thread. La acción que se ejecuta cuando una señal se genera es definida utilizando la función *sigaction()*, y puede ser alguna de las siguientes:

- ejecutar la acción por defecto

- ignorar la señal
- ejecutar el manejador de la señal[46], que es una función que se invoca cuando la señal se genera y acepta

Se dice que una señal ha sido *aceptada* cuando se ejecuta su *manejador*, en caso de haber sido definido. Si algún proceso o thread espera por la señal usando la función *sigsuspend()*, la señal será *aceptada* (se ejecutará el manejador de la señal) y el proceso o thread será activado. Por otra parte, algunos procesos o threads pueden esperar la llegada de la señal usando alguna de las funciones *sigwait()*. En este caso, el manejador no se ejecuta y el thread que espera es activado. Se dice que la señal ha sido *entregada* cuando alguno de los procesos o threads la selecciona y retorna de alguna de las funciones *sigwait()*.

En ocasiones, algún proceso o thread puede bloquear, ya sea de manera temporal o permanente, la llegada de algunas señales, y puede permitir la llegada de otras. Para hacerlo, debe definir el conjunto de señales bloqueadas y no bloqueadas manipulando su *máscara de señales* con la función *sigprocmask()*.

Por otra parte, el modelo para la planificación definida por el usuario, especifica que si una señal es enviada al planificador debe ejecutarse el código de la operación *signal()*, cuyo prototipo se presentó en la página 92. Utilizando la función POSIX *sigaction()* es posible establecer que la función manejadora de una señal particular sea la operación *signal()* del planificador definido por el usuario. La señal, como se explicó, es entregada a un thread cuando la función manejadora sea ejecutada, siempre y cuando no esté bloqueada en la máscara de señales del thread. El problema radica en que si el planificador no es un thread, se dificulta la manipulación de su *máscara de señales* ya que la funciones *sigprocmask()* y *pthread_sigmask()* definen la máscaras de un proceso o thread, respectivamente. Para eliminar esta limitación, el modelo incluye una función para especificar el conjunto de señales que espera un planificador. Si una señal perteneciente al *conjunto de señales esperadas* es recibida, se ejecuta la operación. Por defecto, el planificador es creado con un conjunto de señales vacío. Los prototipos de las funciones para definir y conocer este conjunto de señales son:

```
int posix_appschedattr_setwaitsignalset(
        posix_appsched_scheduler_id_t
                          * scheduler_id,
         const sigset_t *set);

int posix_appschedattr_getwaitsignalset(
        posix_appsched_scheduler_id_t
```

[46] signal handler

```
                                * scheduler_id,
        sigset_t *set);
```

4.9.3.5. Interfaz para los threads planificados por aplicación

Como se explicó en la página 83, antes de crear un thread *planificado-por-aplicación* debe especificarse que será planificado con la nueva política llamada *Application-Defined Scheduling Policy* o *SCHED_APP*, utilizando para esto la función POSIX *pthread_attr_setschedpolicy()*. Una vez hecho esto, debe definirse cuál es el *PDU* responsable de planificar al thread, usando la siguiente función:

```
int pthread_attr_setappscheduler(
        pthread_attr_t *attr,
        posix_appsched_scheduler_id_t scheduler);
```

Para conocer el planificador de algún thread, puede utilizarse la función:

```
int pthread_attr_getappscheduler(
        pthread_attr_t *attr,
        posix_appsched_scheduler_id_t *scheduler);
```

Es posible también definir *parámetros de planificación definidos por el usuario*[47] que podrán ser utilizados también por el *PDU*, entre los que pueden encontrarse el plazo relativo, el periodo, etc. Para definir y conocer estos parámetros se utilizan las funciones:

```
int pthread_attr_setappschedparam(
                    pthread_attr_t *attr,
                    const void *param,
                    size_t paramsize);

int pthread_attr_getappschedparam(
                    pthread_attr_t *attr,
                    const void *param,
                    size_t paramsize);
```

Una vez que se han definido los atributos del thread, incluyendo los relacionados con la planificación definida por el usuario, el thread se crea utilizando la función POSIX *pthread_create()*.

La posibilidad de definir *datos específicos de un thread*[48] representa un mecanismo muy útil en la implementación de aplicaciones multi hilos. POSIX ha

[47] application-defined scheduling parameters
[48] thread-specific data

definido un conjunto de funciones para administrar datos específicos asociados a un thread, que son manipulados utilizando un valor llamado *llave*[49]. Con este mecanismo, varios threads utilizan la misma llave para definir y conocer datos diferentes, cada uno de ellos asociados a un thread distinto. Los datos específicos asociados a los threads no son diferentes a lo que sería si los datos se hubiesen definido como variables globales. Sin embargo, la utilización de este mecanismo ofrece muchas ventajas cuando se construye una biblioteca[50] o cuando no se conoce el número de threads que serán creados y activados en una aplicación.

En términos generales, la secuencia de operaciones definidas por POSIX para utilizar los *datos específicos asociados a un thread*, es la siguiente:

1. Crear (inicializar) una llave, de tipo *pthread_key_t*, utilizando la función *pthread_key_create()*.

2. Establecer, cuando sea necesario, el valor específico de cada thread asociado a la llave, usando la función *pthread_setspecific()*.

3. Obtener, cuando sea necesario, el valor específico del thread asociado con la llave, por medio de la función *pthread_setspecific()*.

4. Una vez que se ha concluido con su uso, liberar la llave utilizando la función *pthread_key_delete()*.

Con este mecanismo, cada thread puede manipular únicamente su información relacionada. Sin embargo, en el contexto de la planificación definida por el usuario, puede ser muy útil el que un planificador tenga acceso a los datos específicos del thread, ya que pueden contener información relacionada con su planificación. Por este motivo, la interfaz presenta un par de funciones para establecer y conocer los datos específicos de un thread asociados a una llave. Los prototipos de las funciones son los siguientes:

```
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);
```

[49] key
[50] library

Con ellas, un planificador puede acceder a los datos de *thread* referidos por *value*, y asociados a la llave *key*.

Cuando el thread se ejecuta, en muchas ocasiones es necesario que informe a su planificador que ha concluido su ejecución para la presente activación. Una vez que es notificado, el planificador puede, por ejemplo, suspender al thread, programar su siguiente activación y elegir el próximo thread a ejecutar. Para que un thread informe a su planificador que su activación ha concluido, las siguientes funciones están disponibles:

```
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);
```

Una vez que el thread ejecuta estas funciones, se suspende en espera de que el planificador determine las acciones a ejecutar. Es posible que exista intercambio de información entre planificador y thread cuando se usa este mecanismo. La función *posix_appsched_invoke_withdata()*, además de notificar al planificador, permite el envío de información entre el thread y el planificador. Si el valor de *msg_size* es mayor que cero, la función hará disponible para el planificador una área de memoria cuyo contenido es idéntico al área de memoria referida por *msg* y de tamaño *msg_size*. Cuando el planificador es notificado de este evento y si el valor de *reply* es diferente a NULL, podrá responder al thread copiando en el área de memoria referida por *reply* un mensaje de tamaño *reply_size*.

4.9.3.6. Sincronización

Si la aplicación utiliza recursos compartidos se hace necesaria la presencia de algoritmos que controlen el acceso a esos recursos. POSIX define los protocolos POSIX_PRIO_INHERIT y POSIX_PRIO_PROTECT [51] para sincronización entre threads que comparten recursos. El primero de ellos implementa el *protocolo de herencia de prioridades*, que se estudió en la sección 2.3.4.1 en la página 39. El segundo implementa una versión simplificada del *protocolo de techo de prioridades*, estudiado en la sección 2.3.4.2 en la página 40 llamado *emulación de techo de prioridad*. Ambos protocolos funcionan bien en sistemas basados en prioridades estáticas, como los definidos por POSIX. Sin embargo, no todas las políticas de planificación pueden adecuarse correctamente a ambos protocolos, por lo que el modelo propone la integración del *protocolo de herencia de urgencia*[51] para monitores y threads que utilicen el protocolo POSIX_PRIO_INHERIT,

[51] urgency inheritance protocol

y la integración del protocolo *SRP*, estudiado en la sección 2.3.4.3 en la página 41, para utilizarse con monitores y threads del tipo POSIX_PRIO_PROTECT.

La integración del protocolo *SRP* y del *protocolo de herencia de prioridades* tiene la finalidad de proporcionar mecanismos de sincronización de manera transparente a las aplicaciones que utilicen la planificación definida por el usuario. Como se recordará, en la definición original del modelo existía la posibilidad de implementar algoritmos de control de acceso a los recursos *definidos por el usuario*. El modelo incluía un conjunto de eventos y funciones para administrar las secciones críticas de una aplicación. En el nuevo modelo, todos estos eventos y funciones han sido eliminados y en su lugar se han integrado los dos protocolos mencionados, que serán estudiados en esta sección.

Para el caso del SRP, para poder hacer que su integración sea compatible con las políticas POSIX existentes, se ha definido un nuevo atributo para threads y monitores que utilicen protocolo POSIX_PRIO_PROTECT, del tipo `unsigned short int` y llamado `preemptionlevel`. La prioridad de los threads y los techos de prioridad de los monitores siguen siendo utilizados, pero se da mayor peso a los techos de prioridad de threads y monitores al definir el nuevo atributo `preemptionlevel`, de la siguiente manera:

$$actual \ = \ prio \ x \ 2^n \ + level$$

en donde *n* es el número de bits utilizado para representar el atributo que representa el nivel de expulsión, llamdo `preemptionlevel` (16 bits).

Para asignar y conocer los valores de `preemptionlevel` las siguientes funciones se han propuesto:

```
int pthread_attr_setpreemptionlevel(
                    pthread_attr_t *attr,
                    unsigned short int level);

int pthread_attr_getpreemptionlevel(
                    pthread_attr_t *attr,
                    unsigned short int *level);

int pthread_mutexattr_setpreemptionlevel(
                    pthread_mutexattr_t *attr,
                    unsigned short int level);

int pthread_mutexattr_getpreemptionlevel(
                    pthread_mutexattr_t *attr,
                    unsigned short int *level);
```

Por otra parte, el protocolo de herencia de urgencia integrado al modelo es una extensión del protocolo de herencia de prioridades, en el que se combinan los dos parámetros utilizados para planificar un thread: sus valores de *prioridad fija* y *urgencia*. Con esta extensión, los threads *planificados-por-aplicación* pueden utilizar monitores del protocolo POSIX_PRIO_INHERIT de manera segura y transparente .

Ya se ha explicado el modelo para la planificación definida por el usuario y se ha presentado la totalidad de su interfaz. A continuación se estudiará su implementación en RTLinux.

4.9.3.7. Implementación del nuevo modelo

El nuevo modelo para la planificación definida por el usuario se ha implementado en RTLinux. RTLinux es un sistema operativo de tiempo-real pequeño y eficiente, que utiliza Linux como sistema operativo de propósito general, mientras que los servicios de tiempo-real están presentes en un sólo proceso multi-hilos, como se estudió en la sección 4.9.2.1 en la página 87. Linux puede verse como un Sistema Multi-propósito de Tiempo-Real PSE54 y RTLinux como un Sistema Mínimo de Tiempo-Real PSE51 [55]. La implementación se hizo utilizando el núcleo OCERA GPL [31], basado en el núcleo Linux 2.4.18 y RTLinux versión 3.2pre1.

Para elegir la mejor estrategia de implementación del modelo para la planificación definida por el usuario se evaluaron tres alternativas. La primera de ellas consiste en implementar a los *PDU* como threads y mejorar su desempeño al incluir la noción de urgencia descrita anteriormente. Cuando un evento ocurra, las operaciones del planificador serán ejecutadas por el thread, que deberá ser activado al nivel de prioridad adecuado. El problema con esta opción es que se incurren en muchos e innecesarios cambios de contexto, pero tiene la ventaja de que es sencilla de implementar. Además, en algunas arquitecturas con esta estrategia sería posible ejecutar a los threads planificadores en modo del usuario, lo que evitaría que un fallo en un planificador afecte al resto del sistema.

La segunda alternativa evaluada consiste en ejecutar las operaciones primitivas en el contexto de los threads planificados por una aplicación. Esto implica que cuando algún thread *planificado-por-aplicación* es elegido para ejecución, antes de hacerlo se ejecutan las operaciones primitivas pendientes de su *PDU* además de las relacionadas con el thread actual. Para garantizar que las operaciones se ejecuten de manera serial[52], pueden protegerse con un monitor en una sección crítica. En el caso de que no existan threads activos y haya operaciones pendientes se hace necesaria la existencia de un *thread de servicio* para que las ejecute. El

[52] serialized

planificador implementado con esta estrategia interfiere tan sólo con threads de prioridad menor que la prioridad del thread de servicio, y con los threads planificados. Esta alternativa es también fácil de implementar y reduce los cambios de contextos de la opción anterior.

La tercera opción es la que se eligió y consiste en ejecutar las operaciones primitivas en el contexto del núcleo, tan pronto como sean generadas. Es la opción más eficiente ya que evita cambios de contextos innecesarios, la existencia de algún thread especial y además incurre en muy poca sobrecarga. La estrategia de implementación seleccionada tiene ventajas adicionales. Debido a que las operaciones y acciones de planificación se ejecutan inmediatamente después de que se producen no es necesario mantener una cola de eventos. De igual manera, las acciones de planificación son ejecutadas inmediatamente y no es necesario mantener un objeto para almacenarlas y ejecutarlas posteriormente. Por otra parte, como el *PDU* no es un thread no se requiere proteger la ejecución de las operaciones primitivas con un monitor para garantizar su ejecución serial. La desventaja de esta opción es que un fallo en el planificador afecta la ejecución del sistema, y su implementación es más compleja. Como se ha mencionado, en arquitecturas como la de RTLinux, en la que las tareas de tiempo-real se ejecutan en modo del núcleo, el fallo en un thread afectará al resto de aplicaciones del sistema, por lo que no existe una ventaja adicional al implementar al planificador como un thread en esta clase de arquitecturas. Por otra parte, la eficiencia que se obtiene ejecutando las operaciones del planificador como *ganchos del núcleo*[53] justifica su complejidad de implementación.

Para implementar la planificación definida por el usuario siguiendo la estrategia seleccionada se hicieron algunas modificaciones al núcleo de RTLinux. Ya estaban implementados los servicios de señales y temporizadores de POSIX [110] así como los mecanismos POSIX para medir los tiempos de ejecución[54] [93], pero era necesario que se agregaran las Extensiones de Señales y Temporizadores de Tiempo-Real de POSIX, por lo que se implementaron e integraron al núcleo OCERA [41].

Por otra parte, debido a que en cada punto de planificación[55], cuando ocurre algún evento relacionado con un thread *planificado-por-aplicación* se debe invocar la operación primitiva correspondiente en caso de que haya sido definida para el *PDU*, todas las funciones relacionadas con eventos de planificación fueron modificadas para que invocaran a la correspondiente operación primitiva. Por ejemplo, la función *pthread_wait_np()* ha sido modificada como se muestra en el Ejemplo 2.

[53] kernel hooks
[54] CPU Time (Execution Time)
[55] scheduling point

Ejemplo 2 Función *pthread_wait_np()*

```
int pthread_wait_np(void)
{
  long interrupt_state;
  pthread_t self = pthread_self();

  rtl_no_interrupts(interrupt_state);
   #ifdef CONFIG_OC_APPSCHED
   if (regular_thread(self) &&
      self->appscheduler->appscheduler_ops_t->thread_block) {
      struct timespec appsched_current_time;
      appsched_current_time = timespec_from_ns(gethrtime());
      self->appscheduler->appscheduler_ops_t->thread_block(
            self->appscheduler->sched_data,
            self, NULL, &appsched_current_time);
   }  else
   #endif
   {
      RTL_MARK_SUSPENDED (self);
   }
   __rtl_setup_timeout (self, self->resume_time);
   rtl_schedule();
   pthread_testcancel();
   rtl_restore_interrupts(interrupt_state);
   return 0;
}
```

En RTLinux, la función *rtl_schedule()*[56] implementa la política de planificación basada en prioridades fijas y también ha sido modificada para implementar el modelo de la planificación definida por el usuario. El pseudocódigo de la función *rtl_schedule()* se muestra en el Ejemplo 3. Como puede observarse en el ejemplo, en un mismo nivel de prioridad se selecciona para ejecución a la tarea *planificada-por-aplicación* que tenga el mayor valor de urgencia.

La totalidad del API de la planificación definida por el usuario se implementó en RTLinux, incluyendo la integración del protocolo SRP para sincronización de threads, y los resultados han sido publicados en [38]. El *protocolo de herencia de urgencia* no fue integrado debido a que threads y monitores del protocolo PTHREAD_PRIO_INHERIT no son soportados en RTLinux. Por otra parte, para el caso de sincronización de tareas críticas y no críticas el modelo propuesto presenta algunos problemas, que serán discutidos en la siguiente sección, así como algunas propuestas para mejorarlo. En el capítulo 7 se presentan ejemplos del uso de la planificación definida por el usuario en RTLinux.

En el siguiente capítulo se discutirán las extensiones que se proponen al modelo.

[56] ésta función se encuentra en el fichero rtl_sched.c

Ejemplo 3 Pseudocódigo de la función *rtl_schedule()*

```
rtl_schedule() {
  obtener tiempo actual;
  establecer new_task=0;
  expirar temporizadores de PDU's;

  bucle a través de la lista de tareas {
      expirar todas las notificaciones para tareas
      planificada-por-aplicación;
      expirar todos los temporizadores;

      si la tarea es planificada-por-aplicación y ha expirado,
      su temporizador de activación, se ejecuta la función
      task->appscheduler->thread_ready()(si está definida);

      new_task = tarea más prioritaria con señales pendientes,
      y si es planificada-por-aplicación,al mismo nivel de
      prioridad, new_task es la que tenga mayor valor de urgencia;
   } fin de bucle
   bucle a través de la lista de tareas {
      actualizar temporizadores one-shot, si es
      que alguna tarea expulsará a new_task;
   } fin de bucle
   la nueva tarea seleccionada
   no es la tarea anterior? {
      cambio a new_task;
      la nueva tarea seleccionada
      usa fpu? {
         almacenar registros fpu;
      }
   }
   manejar señales pendientes de las nuevas tareas;
 }  fin de función
```

Capítulo 5

Interfaz para Planificadores Definidos por el Usuario

En el capítulo anterior se estudiaron diferentes modelos para la definición de planificadores a nivel de usuario, y se concluyó que el más completo de ellos es el propuesto por Aldea y González-Harbour. Éste permite elegir la mejor estrategia de implementación para cada particular arquitectura, contiene una extensa interfaz, y es compatible con POSIX, por nombrar algunas de sus ventajas. Sin embargo, a pesar de ser una extraordinaria propuesta, no cumple todas las condiciones que se definieron en esta tesis como las deseables en un modelo para la definición de planificadores a nivel de usuario. Por una parte, su API para el desarrollo de planificadores no permite la implementación de algunas de las políticas de sincronización existentes, por lo que puede extenderse de tal manera que proporcione mayor flexibilidad. Por otra parte, su API es complejo y puede dificultar la portabilidad de las aplicaciones, por lo que nuevas funciones pueden integrarse a la interfaz para resolver estos problemas y facilitar su uso. En este capítulo se presentarán las extensiones al modelo, que integradas a la interfaz existente, permiten contar con un modelo para la definición de planificadores a nivel de usuario que cumpla con los objetivos definidos en esta tesis.

5.1. Sincronización de tareas aperiódicas

El modelo para la planificación definida por el usuario presentado en [7] aborda el problema de la sincronización de tareas aperiódicas, planificadas por algún servidor aperiódico, de la siguiente manera. Cuando una tarea consume su crédito dentro de una sección crítica, no debe suspender su ejecución sino mantenerse en la cola de threads activos del sistema, probablemente con un nuevo valor de urgencia, hasta que concluya con su sección crítica. Se ha comentado que suspender

una tarea aperiódica dentro de una sección crítica puede ocasionar graves consecuencias para el sistema, por lo que la idea de no suspenderla parece adecuado en primera instancia. Sin embargo, aún a pesar de que la tarea aperiódica no se suspenda dentro de la sección crítica, la planificabilidad del sistema puede ponerse en riesgo ya que el servidor utilizará más tiempo de cómputo que el previamente asignado. Este problema ha sido abordado por varios autores y algunas soluciones han sido publicadas. En la sección 2.3.4.4 se estudiaron las extensiones al *servidor aperiódico* propuestas por Ghazalie y Baker en [47], quienes trataron el problema arriba descrito y propusieron una solución. Si una tarea aperiódica consume su crédito dentro de una sección crítica, deber continuar con su ejecución utilizando crédito extra hasta salir de ella. En las siguientes activaciones del servidor, se restará de su crédito el crédito extra consumido previamente, como se explicó en la página 43. De esta manera, la carga aperiódica no utilizará nunca más tiempo de cómputo que el asignado.

Caccamo y Sha también abordaron este problema en [23] y propusieron una nueva versión del protocolo CBS, a la que llamaron CBS-R, y que previene que un thread consuma completamente su crédito en una sección crítica. Este protocolo se estudió en la página 43. En el CBS-R se restringe el tamaño de las secciones críticas a *porciones* o *chunks*, cada una de ellas con un tamaño máximo equivalente al crédito máximo del servidor. Esto significa que el crédito máximo de cada servidor será siempre suficiente para la ejecución de la mayor de las secciones críticas de sus threads. Antes de que un thread ingrese en una sección crítica se lleva a cabo una verificación del crédito disponible, y si no es suficiente para completarla, el crédito se recarga y un nuevo plazo se genera. Con esta técnica se evita que un thread consuma su crédito después de que se la ha asignado un monitor.

Con la interfaz del nuevo modelo para la *PDU* no es posible implementar correctamente las soluciones arriba descritas, ya que no en todos los casos el planificador es notificado cuando alguno de sus thread entra en una sección crítica. Si bien es cierto al *PDU* se le informa cuando un thread hereda alguna prioridad o nivel de expulsión al asignársela un monitor, puede darse el caso de que el *PDU* no sea notificado si una tarea aperiódica entró en una sección crítica (por ejemplo, si se trata de la tarea más prioritaria en usar el recurso), lo que le impediría definir alguna acción cuando la tarea consuma su crédito después de que se la ha asignado algún monitor. El control del consumo del crédito dentro de la sección crítica, o por ejemplo, el protocolo BWI estudiado en la página 44, no pueden ser correctamente implementados con el API actual.

Para solucionar esta limitación, se propone extender el API del modelo, integrando en él nuevas funciones, que son muy similares a las que existían en la primera versión. En la propuesta inicial del modelo para la planificación definida por el usuario presentado en [4], se incluían funciones para informar al *PDU* cuando una tarea entraba o salía de una sección crítica. Sin embargo, en el nuevo

modelo presentado en [7] las funciones desaparecen. Si se requiere la implementación de alguna política de sincronización más allá de las existentes en el modelo se hacen necesarios mecanismos que permitan determinar que acciones llevar a cabo antes de que un monitor sea asignado a alguna tarea. Por esta razón se propone que se incluyan las operaciones primitivas que informen al *PDU* cuando una tarea *planificada-por-aplicación* intente entrar o salga de una sección crítica. De esta manera, si la aplicación puede ser planificada correctamente con los mecanismos de sincronización presentes en el modelo, dichas funciones no serán definidas, pero si el sistema requiere de mecanismos adicionales, podrán ser fácilmente integrados a la política de planificación.

La propuesta modifica la estructura `posix_appsched_scheduler_ops_t` de la página 92, que contará ahora con tres nuevos miembros, como se muestra a continuación:

```
typedef struct {
   void (*init) (
           void * sched_data, void * arg);
   ...
  void (*lock_mutex) (
           void * sched_data,
           pthread_t thread,
           pthread_mutex_t * mutex,
           posix_appsched_actions_t * actions,
           struct timespec *current_time);

   void (*trylock_mutex) (
           void * sched_data,
           pthread_t thread,
           pthread_mutex_t * mutex,
           posix_appsched_actions_t * actions,
           struct timespec *current_time);

   void (*unlock_mutex) (
           void * sched_data,
           pthread_t thread,
           pthread_mutex_t * mutex,
           posix_appsched_actions_t * actions,
           struct timespec *current_time);

  ...
```

```
} posix_appsched_scheduler_ops_t;
```

La operación primitiva *lock_mutex()* será ejecutada cuando una tarea *planificada-por-aplicación* invoque la operación *lock* sobre un monitor, mientras que la operación primitiva *trylock_mutex()* será ejecutada cuando una tarea *planificada-por-aplicación* invoque una operación *try lock* sobre un monitor. La operación *unlock_mutex()* será ejecutada cuando una tarea *planificada-por-aplicación* libere un monitor previamente cerrado por ella. Con ellas, el planificador será notificado cuando algún evento relacionado con la sincronización de tareas se lleve a cabo.

En el código de las operaciones primitivas propuestas, el planificador podrá ejecutar algunas de las acciones proporcionadas por la interfaz del modelo. Debido a que no se han incluido acciones para administrar secciones críticas, se propone también la integración de una nueva función al modelo. Su prototipo se muestra a continuación:

```
int posix_appsched_actions_addlockmutex(
          posix_appsched_actions_t * actions,
          pthread_t thread,
          pthread_mutex_t * mutex);
```

Con esta función, el planificador agrega la acción de asignación de monitor al objeto especificado por *sched_actions*, que servirá para notificar que el monitor especificado por *mutex* ha sido asignado a *thread*. De esta manera, con la extensión propuesta de la interfaz el planificador será notificado cuando algún thread planificado por él intenta entrar y sale de una sección crítica, y podrá determinar cuándo asignar un monitor a un thread, pudiendo con esto implementarse políticas de planificación definidas por el usuario.

Cabe señalar que si estas operaciones relacionadas con los monitores no han sido especificadas en algún *PDU*, el sistema deberá operar de acuerdo a la política de planificación definida para el monitor correspondiente (PCP, PIP o SRP). Por otra parte, no es necesario definir monitores del tipo *monitores-definidos-por-aplicación* o definir atributos adicionales para los monitores, como se hace en [4], ya que como se estudió en la sección 2.3.4.4 los mecanismos propuestos para la compartición de recursos considerando tareas aperiódicas son en gran medida extensiones de las políticas de planificación y sincronización existentes. Las extensiones propuestas aquí han sido publicadas en [39].

5.2. Bibliotecas de planificadores

El modelo de la planificación definida por el usuario ha sido cuidadosa e inteligentemente diseñado, y con la extensión propuesta para abordar el problema de

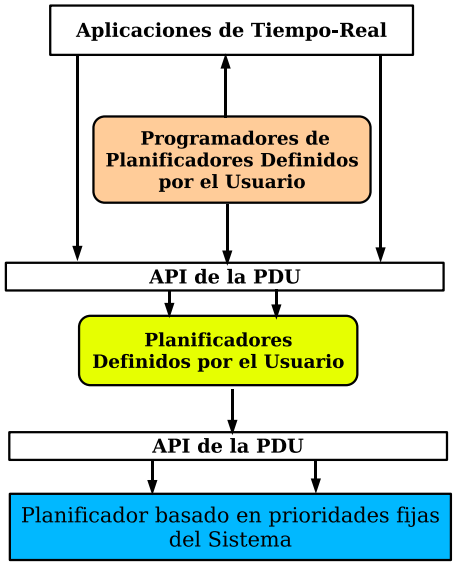

Figura 5.1: API de la planificación definida por el usuario

la sincronización de tareas aperiódicas, se dispone de una alternativa para agregar nuevos servicios de planificación a un SOTR. Sin embargo, no todas las condiciones deseables para la definición de planificadores a nivel de usuario se satisfacen con él, ya que presenta algunas limitaciones. Una de ellas es su complejidad, ya que su API está más orientado a programadores expertos del núcleo de sistemas operativos, por lo que su utilización en el desarrollo de aplicaciones de tiempo-real implica un profundo conocimiento del funcionamiento de cada una de sus llamadas al sistema. Por otra parte, debido a que el modelo no impone una implementación particular y cada planificador puede ser desarrollado de forma diferente, la portabilidad de las aplicaciones se dificulta. Considérese el caso en el que un PDU se implementa como un thread. En el código de la aplicación deberá crearse el thread planificador correspondiente junto con sus parámetros. Sin embargo, la misma aplicación deberá ser cambiada cuando se utilice un PDU que no haya sido implementado como thread.

La portabilidad también es afectada por otros factores. En la medida en que nuevas políticas de planificación se agregan y si no se define una metodología para desarrollar y utilizar los nuevos servicios de planificación, muchos problemas pueden surgir. Para evitar estos problemas y para aprovechar mejor el potencial que ofrece el modelo, se propone una extensión a su interfaz, que además servirá como un marco de referencia para la creación y uso de planificadores desarrollados utilizando el modelo.

El API del modelo de la planificación definida por el usuario actúa como in-

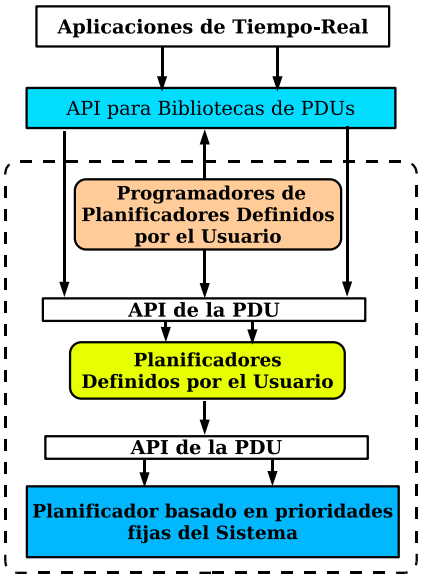

Figura 5.2: API para bibliotecas de planificadores definidos por el usuario

terfaz para programadores del núcleo del sistema operativo, tal y como se muestra en la Figura 5.1. En la Figura 5.2 se muestra la estructura del API propuesto, que actúa como una interfaz entre las aplicaciones de tiempo-real y los servicios de la planificación definida por el usuario. La propuesta presentada en esta tesis no modifica el API existente sino que lo extiende para facilitar su portabilidad y uso, al mismo tiempo que permite agregar nuevos servicios. En lo que resta del capítulo se describe con detalle el API propuesto, que está constituido por dos tipos de funciones. El primer grupo de funciones es de tipo *obligatorio*, ya que deberán estar presentes en toda implementación de la interfaz. Pero debido a que el modelo debe ser flexible para soportar diversas políticas de planificación, existe un segundo grupo de funciones de tipo *opcional*, ya que dependen de las características particulares de los algoritmos de planificación que se implementen. Se estudiará primero la parte obligatoria del API, y posteriormente se presentarán las guías para implementar las funciones opcionales.

5.3. Interfaz para planificadores

Como se explicó, la manera en que cada PDU sea implementado afecta la portabilidad de las aplicaciones. Se hace necesario la definición de un conjunto de funciones que faciliten la portabilidad y que además aislen los aspectos relacionados con la implementación del PDU, permitiendo que las políticas de planifica-

Interfaz para planificadores
appschedlib_attr_init()
appschedlib_attr_destroy()
appschedlib_attr_set_synchstart()
appschedlib_attr_get_synchstart()
appschedlib_attr_set_schedhandlers()
appschedlib_attr_get_schedhandlers()
appschedlib_attr_set_server()
appschedlib_attr_get_server()
appschedlib_init_schedpolicy()
appschedlib_get_schedpolicy()
appschedlib_set_schedprio()
appschedlib_set_synchstart()
appschedlib_get_synchstart()
appschedlib_set_schedhandlers()
appschedlib_get_schedhandlers()
appschedlib_set_server()
appschedlib_get_server()

Cuadro 5.1: Funciones obligatorias para planificadores en la interfaz propuesta

ción que se integren al SOTR sean utilizadas de manera consistente. El conjunto de funciones propuestas y que forman la interfaz obligatoria para planificadores se presenta de forma resumida en el Cuadro 5.1.

5.3.1. Objeto de atributos

Cada PDU que se utilice en una aplicación debe contener al menos dos atributos, a saber: su *política* de planificación y su *prioridad*. El primero de ellos identifica la política de planificación a utilizar: RM, DM, EDF, etc. El segundo, el nivel de prioridad al que operará el PDU. Sin embargo, en algunos casos pudiera ser necesario la definición de atributos adicionales. Con la interfaz propuesta deben definirse todos los atributos del PDU de manera similar al modelo definido por POSIX. Cuando sea necesario definir atributos adicionales a la política y la prioridad del planificador, puede utilizarse un *objeto* de atributos del PDU, del tipo *appschedlib_attr_t*, y que se inicializa utilizando la siguiente función:

```
int appschedlib_attr_init(
          appschedlib_attr_t * attr);
```

en donde *attr* es el objeto que contendrá los atributos del PDU. Esta función inicializará el objeto de atributos con los valores por defecto. En el siguiente capítulo se explicará con más detalle la estructura del objeto y los valores que por defecto se asignan. Por otra parte, la siguiente función debe utilizarse si se desea destruir el objeto de atributos:

```
int appschedlib_attr_destroy(
          appschedlib_attr_t * attr);
```

El uso de esta función destruye el objeto al que hace referencia *attr*, de tal manera que el objeto quede sin inicializar. El objeto al que hace referencia *attr* puede ser inicializado de nuevo utilizando la función *appsched_attr_init()*.

5.3.1.1. Valor de retorno y errores

Los valores que estas funciones pueden retornar son los siguientes:

[0] Si las funciones tienen éxito, retornarán el valor de cero.

La función *appsched_attr_init()* fallará si:

[ENOMEM] No existe suficiente memoria para inicializar el objetos de atributos del planificador.

La función *appsched_attr_destroy()* fallará si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado.

5.3.2. Atributos de la política de planificación

Por cada atributo, la interfaz contiene las funciones correspondiente para definir y conocer su valor, ya sea de manera estática o dinámica. Uno de los atributos que pueden definirse en relación con el PDU es el número de threads que forman parte del conjunto síncrono de tareas del sistema, y que se explica a continuación.

En muchas ocasiones, controlar el estado inicial del sistema puede ser importante. Por ejemplo, pudiera requerirse que todos los threads inicien su ejecución en el mismo instante de tiempo. De hecho, casi todos los tests de planificabilidad suponen que el conjunto de tareas es síncrono. POSIX ha abordado este problema y ha propuesto una solución por medio del uso de *barreras*[1]. Las barreras de POSIX son usadas típicamente en bucles DO/FOR para garantizar que todos los threads

[1] barriers

hayan alcanzado un estado particular en el cómputo paralelo o concurrente, antes de permitirles continuar al siguiente estado [52]. Sin embargo, están definidas en la *especificación Avanzada de Threads de Tiempo-Real*[2], que es opcional y por lo tanto no está presente en todo sistema operativo. Además, algunas de las funciones relacionadas con el uso de barreras deben estar presentes en el código del thread, lo que hace más compleja la implementación de sistemas de tiempo-real. Por otra parte, algunos sistemas operativos han propuesto una solución a este problema y la han incorporado en su API, como por ejemplo RTLinux con las funciones *pthread_make_periodic_np()* y *pthread_wait_np()*, que requieren de la definición del instante de tiempo en el que las tareas iniciarán su ejecución. Sin embargo, la determinación de ese instante de tiempo se hace de manera arbitraria. El API propuesto ofrece una solución diferente a este problema y no requiere del uso de funciones adicionales en el código de los threads ni de establecer un tiempo inicial. La siguiente función:

```
int appschedlib_attr_set_synchstart(
          appsched_attr_t * attr,
          int n);
```

hace que los primeros `n` threads definidos con la política `appscheduler` inicien su ejecución inmediatamente después de que el $n_{\text{ésimo}}$ thread haya alcanzado su *punto de sincronía*. Es importante señalar que para sincronizar a los threads, la función *posix_appsched_invoke_scheduler()* debe ser utilizada. Cada thread del conjunto se bloqueará al usar la función, y continúan su ejecución cuando el último de los `n` threads la haya llamado. La utilización de este mecanismo permite implementar de mejor manera un conjunto de tareas síncrono. Para ilustrarlo, considérese el caso que a continuación se presenta.

Más adelante se discutirá la manera de definir threads periódicos usando la interfaz propuesta. Por lo pronto, puede decirse que el código de un thread periódico esta típicamente compuesto por dos secciones. La primera de ellas consiste en un conjunto de *funciones de inicialización*, que se ejecutan sólo una vez e inmediatamente después de que el thread es creado. Con estas funciones, el thread puede crear e inicializar colas de mensajes, temporizadores, dispositivos, etc. La segunda sección del código del thread está formada por un conjunto de funciones que se ejecutan en cada periodo de activación del thread, a las que llamaremos *funciones periódicas*. Con ellas es con las que se llevan a cabo las acciones del thread. Generalmente, en la primera activación de cada thread se ejecutan tan sólo las funciones de inicialización, y posteriormente el thread se suspende en espera de sus siguientes activaciones, para ejecutar las funciones periódicas. En el Ejemplo 4 se muestra el pseudocódigo de un thread periódico. En el ejemplo, antes

[2] Advanced Real-Time Threads specification

Ejemplo 4 Pseudocódigo de un thread periódico

```
int thread_code(...) {
   ...
   ejecutar funciones de inicialización
   ...
   while (1) {
      posix_appsched_invoke_scheduler();
      ...
      ejecutar funciones periódicas
      ...
   }
```

del bucle infinito implementado con la instrucción *while(1)*, el thread ejecuta sus funciones de inicialización. Al entrar al bucle, el thread se suspende al invocar a la función *posix_appsched_invoke_scheduler()*. Cuando reanuda su ejecución, a partir de ese periodo el thread ejecuta sus funciones periódicas.

La Figura 5.3 muestra un ejemplo en donde un conjunto formado por tres tareas no síncronas: T_1 = (3, 1), T_2 = (5, 1) y T_3 = (8, 1), planificadas con el RM. Puede observarse como las tareas ejecutan las funciones de inicialización al ser creadas, y luego se suspenden en espera de sus siguientes periodos para ejecutar las funciones periódicas. Por otra parte, la Figura 5.4 muestra el mismo conjunto de tareas, que ha sido definido como un conjunto síncrono utilizando la función *appschedlib_attr_set_synchstart()*. Las tareas ejecutan sus funciones de inicialización y se suspenden en el punto en el que invocan a la función *posix_appsched_invoke_scheduler()*. Una vez que la última tarea ha hecho esto, todas las tareas son activadas nuevamente en el punto en el que se suspendieron por primera vez, llamado *instante de sincronía*. que en el modelo propuesto se define utilizando la función *posix_appsched_invoke_scheduler()*, como ya se ha explicado. Como puede observarse, la interfaz propuesta ofrece un mecanismo limpio y eficiente para la definición de conjuntos de tareas síncronos. Para conocer el número de tareas que forman el conjunto síncrono puede utilizarse:

```
int appschedlib_attr_get_synchstart(
          const appsched_attr_t * attr,
          int * n);
```

Además, es posible definir y conocer este valor de forma dinámica, con el uso de las siguientes funciones:

```
int appschedlib_set_synchstart(
```

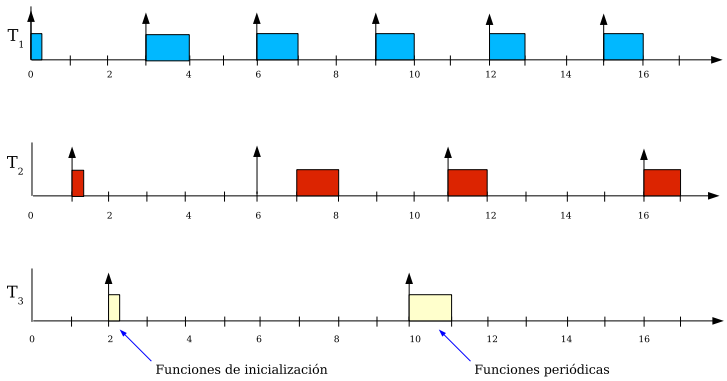

Figura 5.3: Ejemplo de un conjunto de tareas no síncrono

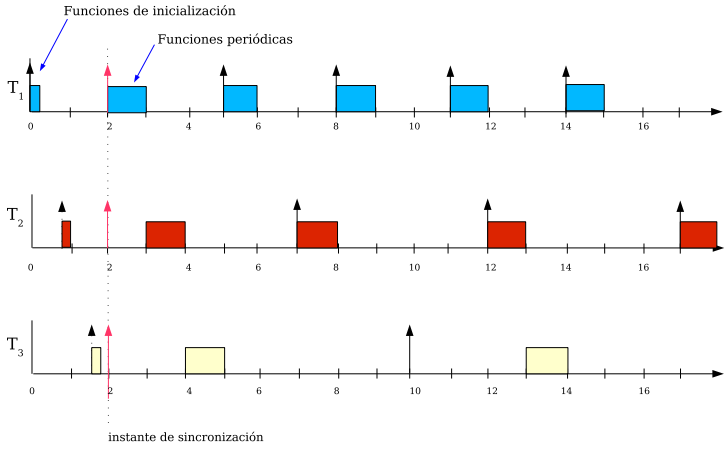

Figura 5.4: Ejemplo de un conjunto de tareas síncrono

```
            appschedlib_sched_t * appscheduler,
            int n);

int appschedlib_get_synchstart(
            const appschedlib_sched_t * appscheduler,
            int * n);
```

en donde *appscheduler* es el identificador de la política de planificación.

Por otra parte, en ocasiones la política de planificación hace uso de uno o más servidores aperiódicos. Por ejemplo, en un sistema formado por tareas periódicas críticas y por tareas aperiódicas, puede utilizarse la política EDF para planificar a las primeras, y la política CBS para las segundas. En este caso, la política EDF + CBS requiere de la definición de uno o más servidores aperiódicos. En la interfaz propuesta, cada servidor puede ser definido, junto con sus parámetros de *periodo* y *crédito*, utilizando la siguiente función:

```
int appschedlib_attr_set_server(
            appsched_attr_t * attr,
            appschedlib_server_t * server,
            struct timespec * period
            struct timespec * budget);
```

Esa función deberá invocarse por cada servidor que sea utilizado en la política de planificación. El parámetro `server` se utiliza como identificador único de cada servidor.

Para conocer los parámetros del servidor identificado por `server` , o para definir y conocer los parámetros del servidor de manera dinámica, las siguientes funciones se han definido:

```
int appschedlib_attr_get_server(
            const appsched_attr_t * attr,
            const appschedlib_server_t * server,
            struct timespec * period
            struct timespec * budget);

int appschedlib_set_server(
            appschedlib_sched_t * appscheduler,
            appschedlib_server_t * server,
            struct timespec * period
            struct timespec * budget);

int appschedlib_get_server(
```

```
const appschedlib_sched_t * appscheduler,
const appschedlib_server_t * server,
struct timespec * period
struct timespec * budget);
```

5.3.2.1. Valor de retorno y errores

Los valores que estas funciones pueden retornar son los siguientes:

[0] Si las funciones tienen éxito, retornarán el valor de cero.

La función *appsched_attr_set_synchstart()* fallará si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado, o el valor especificado por *n* no es válido.

La función *appsched_attr_get_synchstart()* fallará si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado.

La función *appsched_set_synchstart()* fallará si:

[EINVAL] El valor especificado por *n* no es válido.

[ESRCH] El valor especificado por *appscheduler* no hace referencia a un identificador de alguna política de planificación previamente inicializada.

La función *appsched_get_synchstart()* fallará si:

[ESRCH] El valor especificado por *appscheduler* no hace referencia a un identificador de alguna política de planificación previamente inicializada.

La función *appsched_attr_set_server()* fallará si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado, o los valores especificado por *period* o *budget* no son válidos.

La función *appsched_attr_get_server()* fallará si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado.

La función *appsched_set_server()* fallará si:

[EINVAL] El valor especificado por *server* no hace referencia a un objeto identificador de servidor válido, o los valores especificado por *period* o *budget* no son válidos.

[ESRCH] El valor especificado por *appscheduler* no hace referencia a un identificador de alguna política de planificación previamente inicializada.

La función *appschedlib_get_server()* fallará si:

[ESRCH] El valor especificado por *appscheduler* no hace referencia a un identificador de alguna política de planificación previamente inicializada.

5.3.3. Inicio de la política de planificación

Una vez que se han definido los atributos de la política de planificación, se inicia su operación utilizando la función cuyo prototipo se muestra a continuación:

```
int appschedlib_init_schedpolicy(
          appschedlib_sched_t * appscheduler,
          appsched_attr_t * attr,
          appschedlib_policy_t policy,
          int prio);
```

El parámetro `appscheduler` es un identificador de la política de planificación. El parámetro `policy` define la política de planificación a utilizar, y puede tener la siguiente forma:

```
typedef enum {
         APPSCHEDLIB_RR,
         APPSCHEDLIB_RM,
         APPSCHEDLIB_EDF,
         APPSCHEDLIB_RM_DS,
         APPSCHEDLIB_EDF_CBS,
         APPSCHEDLIB_EDF_IRIS,
          ..
          ..
}   appschedlib_policy_t;
```

Para conocer la política de planificación definida para un identificador determinado, la siguiente función es utilizada:

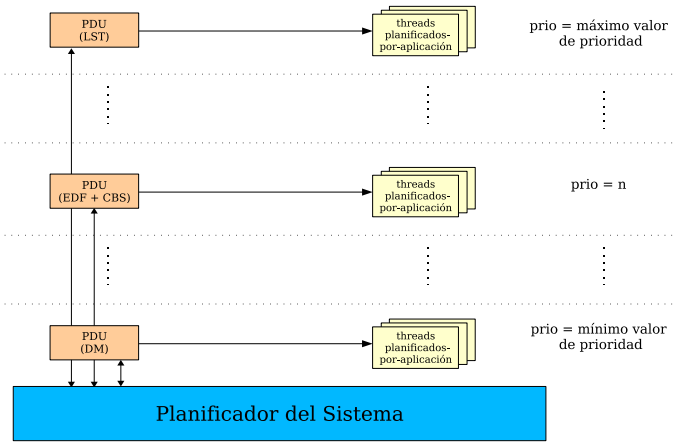

Figura 5.5: PDUs con diferentes niveles de prioridad

```
int appschedlib_get_schedpolicy(
          const appschedlib_sched_t * appscheduler,
          appschedlib_policy_t * policy,
          int * prio);
```

Cada política de planificación debe tener definido un nivel de prioridad, y ese nivel de prioridad será asignado a los threads que planifique. El parámetro `prio` define la prioridad de la política de planificación. El nivel de prioridad de la política de planificación es la prioridad fija de sus thread y es utilizada por el planificador del sistema operativo. Dentro del mismo nivel de prioridad cada thread podrá tener una prioridad diferente, que será asignada de acuerdo al algoritmo específico, representada en el valor de *urgencia* del thread, y utilizada para tomar las decisiones de planificación. Este enfoque no es restrictivo ya que pueden definirse varias políticas de planificación al mismo tiempo, cada una trabajando a su propio nivel de prioridad, como se ilustra en la Figura 5.5.

Una vez iniciada la política de planificación es posible modificar o conocer el valor de algunos de sus atributos, utilizando las funciones *dinámicas* presentadas en esta sección. En el caso particular del nivel de prioridad, puede utilizarse la siguiente función:

```
int appschedlib_set_schedprio(
          appschedlib_sched_t * appscheduler,
          int prio);
```

Con ella, el nivel de prioridad de la política de planificación puede cambiarse.

Para ilustrar el uso de la interfaz para planificadores, en el Ejemplo 5 se muestra el pseudocódigo de la función *main()* de un programa que utiliza la política EDF usando el modelo propuesto. En el ejemplo no se definen atributos

Ejemplo 5 Ejemplo de uso de la interfaz para planificadores

```
. . .

 /* se define identificador de planificador */
static appschedlib_sched_t edf_sched_id;
. . .
int main(void)
{
  /* se define la política de planificación */

   appschedlib_init_schedpolicy(&edf_sched_id,
                  NULL, APPSCHEDLIB_EDF, 10);

  . . .
   return 0;
 }
```

adicionales para la política de planificación. Por esta razón, es sólo suficiente con definir un objeto identificador de la política de planificación, del tipo *appschedlib_sched_t*, y posteriormente iniciar la política usando la función *appschedlib_init_schedpolicy()*, indicando el ella la política a utilizar, así como su nivel de prioridad. En el Capítulo 7 se muestran más ejemplos.

5.3.3.1. Valor de retorno y errores

Los valores que estas funciones pueden retornar son los siguientes:

[0] Si las funciones tienen éxito, retornarán el valor de cero.

La función *appsched_init_schedpolicy()* fallará si:

[EINVAL] El valor especificado por *appscheduler* no hace referencia a un identificador válido, o los valores especificados por *policy* o *prio* no son válidos.

[EAGAIN] El sistema no cuenta con recursos suficientes para crear un nuevo PDU, o el número de planificadores ha alcanzado el límite impuesto por el sistema.

La función *appsched_get_schedpolicy()* fallará si:

[ESRCH] El valor especificado por *appscheduler* no hace referencia a un identificador de alguna política de planificación previamente inicializada.

La función *appsched_set_schedprio()* fallará si:

[ESRCH] El valor especificado por *appscheduler* no hace referencia a un identificador de alguna política de planificación previamente inicializada.

[EINVAL] El valor especificado por *prio* no es válido.

5.4. Interfaz para threads

De acuerdo al enfoque propuesto por POSIX, antes de la creación de un thread pueden definirse algunos de sus atributos, como su prioridad, entre otros. Siguiendo este enfoque, también es posible definir los atributos de los threads *planificados-por-aplicación* antes de crearlos, y que son relevantes para la política de planificación. En esta sección se estudiará la interfaz *obligatoria* para threads, que se muestra de manera condensada en el Cuadro 5.2.

Antes de crear un thread debe definirse la política con la que será planificado. Esto se hace especificando cuál será su planificador, utilizando la siguiente función:

```
int pthread_attr_set_appschedlib_scheduler_np(
          pthread_attr_t * attr,
          appschedlib_sched_t scheduler);
```

Por otra parte, para conocer el planificador al que algún thread fue asignado, se utiliza la función:

```
int pthread_attr_get_appschedlib_scheduler_np(
          const pthread_attr_t * attr,
          appschedlib_sched_t * scheduler);
```

Y para definir y conocer de manera dinámica el planificador del thread, se utilizan:

```
int pthread_set_appschedlib_scheduler_np(
          pthread_t thread,
          appschedlib_sched_t  scheduler);

int pthread_get_appschedlib_scheduler_np(
          const pthread_t  thread,
          appschedlib_sched_t * scheduler);
```

Interfaz para threads
pthread_attr_set_appschedlib_scheduler_np()
pthread_attr_get_appschedlib_scheduler_np()
pthread_attr_set_appschedlib_server_np()
pthread_attr_get_appschedlib_server_np()
pthread_attr_set_appschedlib_period_np()
pthread_attr_get_appschedlib_period_np()
pthread_attr_set_appschedlib_deadline_np()
pthread_attr_get_appschedlib_deadline_np()
pthread_attr_set_appschedlib_maxexectime_np()
pthread_attr_get_appschedlib_maxexectime_np()
pthread_attr_set_appschedlib_schedhandlers_np()
pthread_attr_get_appschedlib_schedhandlers_np()
pthread_set_appschedlib_scheduler_np()
pthread_get_appschedlib_scheduler_np()
pthread_set_appschedlib_server_np()
pthread_get_appschedlib_server_np()
pthread_set_appschedlib_period_np()
pthread_get_appschedlib_period_np()
pthread_set_appschedlib_deadline_np()
pthread_get_appschedlib_deadline_np()
pthread_set_appschedlib_maxexectime_np()
pthread_get_appschedlib_maxexectime_np()
pthread_set_appschedlib_schedhandlers_np()
pthread_get_appschedlib_schedhandlers_np()
pthread_get_appschedlib_exectime_np()

Cuadro 5.2: Funciones obligatorias para threads en la interfaz propuesta

en donde *scheduler* es el identificador del planificador.

Es importante señalar que bajo el modelo propuesto no es necesario definir la prioridad fija del thread, ya que le será asignada la prioridad de la política de planificación que lo planifique.

Para el caso de threads aperiódicos, como se recordará, existen políticas de planificación que utilizan un servidor aperiódico para planificarlos. En el modelo propuesto, si algún thread es planificado por algún servidor, no se necesario especificar su planificador sino que será suficiente con definir el servidor que lo planificará. Para definir y conocer el servidor asociado a un thread se utilizan las siguientes funciones:

```
int pthread_attr_set_appschedlib_server_np(
          pthread_attr_t * attr,
          appschedlib_server_t server);

int pthread_attr_get_appschedlib_server_np(
          const pthread_attr_t * attr,
          appschedlib_server_t * server);

int pthread_set_appschedlib_server_np(
          pthread_t  thread,
          appschedlib_server_t server);

int pthread_get_appschedlib_server_np(
          const pthread_t  thread,
          appschedlib_server_t * server);
```

En el caso de los threads aperiódicos, serán planificados con los parámetros de crédito y periodo del servidor correspondiente.

Por otra parte, la definición del comportamiento periódico de los threads ha sido tratado de manera especial en el modelo propuesto. POSIX proporciona funciones para definir threads periódicos, por medio del uso de temporizadores o de funciones como *clock_nanosleep()*. Sin embargo, al utilizar estos mecanismos POSIX la aplicación debe incluir funciones que garanticen el comportamiento periódico de los threads, como puede ser armar y desarmar temporizadores, o llevar control del tiempo para establecer el siguiente periodo. Para evitar lo anterior y simplificar su implementación, en el API propuesto se ha definido un mecanismo para la definición de threads periódicos, sin la necesidad de incluir en el código del thread funciones adicionales. Para definir un thread periódico tan sólo es necesario establecer el periodo del thread usando la siguiente función antes de crearlo.

```
int pthread_attr_set_appschedlib_period_np(
```

```
            pthread_attr_t * attr,
            struct timespec * period);
```

Al igual que en las funciones presentadas previamente, es posible conocer el valor del periodo de un thread, y de definirlo y conocerlo de manera dinámica, utilizando las siguientes funciones:

```
int pthread_attr_get_appschedlib_period_np(
            const pthread_attr_t * attr,
            struct timespec * period);

int pthread_set_appschedlib_period_np(
            pthread_t thread,
            struct timespec * period);

int pthread_get_appschedlib_period_np(
            const pthread_t thread,
            struct timespec * period);
```

en donde *period* indica el intervalo de tiempo entre activaciones sucesivas del thread.

Para indicar en el código del thread el instante en el que el thread se suspenderá en espera del siguiente periodo, la función *posix_appsched_invoke_scheduler()* debe utilizarse, con la que además invocará explícitamente al planificador indicando que ha concluido su actual activación. Cuando el planificador es invocado de esta manera y si el thread es periódico, suspende al thread que hizo la llamada a la función y lo activa en el siguiente periodo. Para ilustrar lo sencillo que es el mecanismo de implementación de threads periódicos se muestra en el Ejemplo 4 el pseudocódigo de un thread periódico. Es importante destacar que la función *posix_appsched_invoke_scheduler()* es utilizada para invocar al planificador, para definir el instante de sincronización de los threads síncronos, y para definir su naturaleza periódica. De esta manera se consigue que el código del thread sea lo más sencillo posible.

Para ilustrar el uso de la interfaz para threads del modelo propuesto, en el Ejemplo 6 se muestran el pseudocódigo de la función *main()* del ejemplo presentado en la sección anterior, y en la que se define un thread periódico y que será planificado con la política EDF.

Cuando un thread periódico se crea de la manera anteriormente descrita se considera que su plazo es igual a su periodo. Sin embargo, un plazo diferente puede definirse utilizando la función que se muestra a continuación:

```
int pthread_attr_set_appschedlib_deadline_np(
```

Ejemplo 6 Ejemplo de uso de la interfaz para threads

```
. . .
pthread_t thread;

/* se define identificador de planificador */
static appschedlib_sched_t edf_sched_id;
. . .
int main(void)
{
   pthread_attr_t th_attr;
   struct timespec period;

 /* se define la política de planificación */

   appschedlib_init_schedpolicy(&edf_sched_id,
                  NULL, APPSCHEDLIB_EDF, 10);

   pthread_attr_init(&th_attr);

/* se define el periodo del thread */
   period.tv_sec = 0;
   period.tv_nsec = 30*ONEMILI;
   pthread_attr_set_appschedlib_period_np(
                          &th_attr, &period);

 /* se define la política de planificación del thread */
   pthread_attr_set_appschedlib_scheduler_np(
                          &th_attr, edf_sched_id);

 /* se crea el thread */
   pthread_create (&thread, &th_attr, thread_code, NULL);
   . . .
   return 0;
 }
```

```
            pthread_attr_t * attr,
            struct timespec * deadline);
```

Además, las siguientes funciones relacionadas con el plazo del thread se han definido:

```
int pthread_attr_get_appschedlib_deadline_np(
            const pthread_attr_t * attr,
            struct timespec * deadline);

int pthread_set_appschedlib_deadline_np(
            pthread_t  thread,
            struct timespec * deadline);

int pthread_get_appschedlib_deadline_np(
            const pthread_t  thread,
            struct timespec * deadline);
```

en donde *deadline* representa el plazo del thread.

Como se verá en la siguiente sección, es posible llevar control del *tiempo de ejecución* de un thread e invocar alguna función específica si el thread se ejecuta por más tiempo que el esperado. Para definir el *tiempo de cómputo máximo* esperado en un thread, se utiliza la función:

```
int pthread_attr_set_appschedlib_maxexectime_np(
            pthread_attr_t * attr,
            struct timespec * wcet);
```

Las siguientes funciones relacionadas con el tiempo máximo de ejecución de un thread también están disponibles:

```
int pthread_attr_get_appschedlib_maxexectime_np(
            const pthread_attr_t * attr,
            struct timespec * wcet);

int pthread_set_appschedlib_maxexectime_np(
            pthread_t thread,
            struct timespec * wcet);

int pthread_get_appschedlib_maxexectime_np(
            const pthread_t thread,
            struct timespec * wcet);
```

En estas funciones, el parámetro *wcet* especifica el tiempo máximo en el que el thread debe ejecutarse, en el peor caso. Si algún thread consume el tiempo de ejecución especificado en este parámetro y aún no ha concluido con su ejecución, ocurrirá lo siguiente:

1. Se ejecutan el manejador de cómputo en exceso, si es que se ha definido.
2. Se invoca a la función del planificador *thread_block()*, si se ha definido, y si el thread es planificado por algún servidor aperiódico, ya que significa que ha consumido su crédito.
3. Se ignora, si ninguna de las dos condiciones anteriores se cumple.

Se ha comentado que la gran mayoría de los tests de planificabilidad consideran el *tiempo de cómputo* de las tareas, y que algunas políticas de control de acceso a recursos requieren conocer la duración máxima de sus secciones críticas, por lo que se ha incluido en la interfaz una función que permita conocer los tiempos de cómputo de las tareas. Dicha función hace uso del mecanismo CPU-Time de POSIX para conocer con precisión el tiempo-real de CPU utilizado por una tarea. El prototipo de la función es el siguiente:

```
int pthread_get_appschedlib_exectime_np(
          const pthread_t thread,
          struct timespec * exectime);
```

Una vez utilizada la función, en el parámetro *exectime* se tendrá el *tiempo de cómputo* de la última invocación del thread. Esta función permite además conocer fácilmente el cómputo en exceso incurrido por una tarea aperiódica dentro de una sección crítica, para ser descontado en siguientes recargas del crédito, como en los mecanismos de sincronización de tareas aperiódicas propuestos por Ghazalie y Baker estudiados previamente.

Existen otras funciones que forman parte de la interfaz obligatoria para threads y que serán estudiadas en la siguiente sección, ya que están relacionadas con las prestaciones de tolerancia a fallos del modelo propuesto.

5.4.1. Valor de retorno y errores

Los valores que estas funciones pueden retornar son los siguientes:

[0] Si las funciones tienen éxito, retornarán el valor de cero.

La funciones:
pthread_attr_set_appschedlib_scheduler_np(), pthread_attr_get_appschedlib_scheduler_np(), pthread_attr_set_appschedlib_server_np(), pthread_attr_get_appschedlib_server_np(), pthread_attr_set_appschedlib_period_np(), pthread_attr_get_appschedlib_period_np(), pthread_attr_set_appschedlib_deadline_np(), pthread_attr_get_appschedlib_deadline_np(), pthread_attr_set_appschedlib_maxexectime_np(), pthread_attr_set_appschedlib_schedhandlers_ np(), pthread_attr_get_appschedlib_maxexectime_np(), pthread_attr_get_appschedlib_ schedhandlers _np()
fallarán si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado.

La funciones:
pthread_set_appschedlib_scheduler_np(),pthread_get_appschedlib_scheduler_np(),pthread_set_appschedlib_server_np(),pthread_get_appschedlib_server_np(),pthread_set_appschedlib_period_np(), pthread_get_appschedlib_period_np(),pthread_set_appschedlib_deadline_np(),pthread_get_appschedlib_deadline_np(), pthread_set_appschedlib_maxexectime_np(), pthread_set_appschedlib_schedhandlers_np(), pthread_get_appschedlib_maxexectime_np(), pthread_get_appschedlib_sched handlers_np(), pthread_get_appschedlib_exectime_np()
fallarán si:

[ESRCH] El valor especificado por *thread* no hace referencia a un thread existente.

Las funciones *pthread_attr_set_appschedlib_scheduler_np()* y *pthread_set_appschedlib_scheduler_np()* fallarán si:

[EINVAL] El valor especificado por *scheduler* no es válido.

Las funciones *pthread_attr_set_appschedlib_server_np()* y *pthread_set_appschedlib_server_np()* fallarán si:

[EINVAL] El valor especificado por *server* no es válido

Las funciones *pthread_attr_set_appschedlib_period_np()* y *pthread_set_appschedlib_period_np()* fallarán si:

[EINVAL] El valor especificado por *period* no es válido.

Las funciones *pthread_attr_set_appschedlib_deadline_np()* y *pthread_set_appschedlib_deadline_np()* fallarán si:

[EINVAL] El valor especificado por *deadline* no es válido.

Las funciones *pthread_attr_set_appschedlib_maxexectime_np()* y *pthread_set_appschedlib_maxexectime_np()* fallarán si:

[EINVAL] El valor especificado por *wcet* no es válido.

5.5. Interfaz para tolerancia a fallos

Un sistema de tiempo-real puede experimentar fallos a pesar de que se diseñe correctamente. Los fallos pueden ser causados por un incorrecto diseño del hardware, errores en el código de la aplicación, deficiencias en el funcionamiento de los canales de entrada, o cambios en el entorno del sistema, por mencionar algunos. Además, aún cuando no ocurra alguno de estos fallos, el sistema puede incumplir sus restricciones temporales si las tareas se ejecutan por más tiempo del esperado. Hay que recordar que la gran mayoría de los tests de planificabilidad existentes se basan en el conocimiento del *tiempo de cómputo en el peor caso* (*wcet*)[3] de las tareas. A pesar de que el conjunto de tareas satisfaga el test correspondiente, en la ejecución de estos sistemas pueden ocurrir errores debidos principalmente a la dificultad para determinar con exactitud el *tiempo de cómputo* del código. Los fallos en el sistema pueden provocar el *incumplimiento de los plazos*[4] de las tareas, que ocurren cuando las tareas de tiempo-real fallan en terminar su ejecución antes del plazo tolerado [107].

Se explicó en la sección anterior que la interfaz propuesta incluye una función para conocer el *tiempo de cómputo* de las tareas, que es de gran utilidad en la fase de prueba de una aplicación de tiempo-real, ya que con ayuda de esta función de podrán conocer con precisión los *tiempos de cómputo* de las tareas en una arquitectura específica. Sin embargo, aún así el sistema pudiera presentar fallos, como se ha explicado.

Dos de los problemas más importantes que un sistema erróneo puede sufrir son los *fallos en plazos*[5] y el *cómputo en exceso*[6]. El primero de ellos puede ocurrir cuando una tarea no se completa antes de su plazo, debido a algún fallo en el sistema o por la interferencia de tareas de mayor prioridad. Por otra parte, una tarea incurre en *cómputo en exceso* cuando se ejecuta por un tiempo mayor que su tiempo de cómputo garantizado (*wcet*) [45]. Muchas técnicas de tolerancia a fallos que han sido desarrolladas se benefician ampliamente de las capacidades de detección de *fallos en plazos* o *cómputo en exceso* [22]. Varias acciones pueden llevarse a cabo cuando alguna tarea falla, por ejemplo: se puede disminuir su prioridad, modificar su plazo, ejecutar una versión subóptima de la tarea, ejecutar acciones de recuperación, etc. Por estas razones, se ha incluido servicios de tolerancia a fallos en el modelo propuesto, y en particular, un conjunto de funciones para detectar *fallos en plazos* y *cómputo en exceso*.

Por otra parte, algunos algoritmos requieren que el sistema sea capaz de detectar fallos en el sistema. Por ejemplo, en el *modelo de planificación de computación*

[3] worst case execution time
[4] missed deadline
[5] deadline misses
[6] overrun

opcional para sistemas tolerantes a fallos[7] propuesto por Mejía-Alvarez *et al*. en [79], las tareas se dividen en dos partes: una llamada obligatoria y otra opcional. Cuando la parte obligatoria de la tarea falla algún plazo, debe ejecutarse de nuevo o ejecutarse un conjunto de operaciones de recuperación. También, en [75], Liu y Lee propusieron un modelo de programación para sistemas empotrados orientado a eventos, que se beneficia de la capacidad del sistema para detectar cuando una tarea falla un plazo o incurre en cómputo en exceso. Como puede verse, si el modelo es capaz de detectar este tipo de problemas, es posible implementar una gran cantidad de técnicas de tolerancia a fallos, lo que hace que el modelo sea aún más flexible.

Es posible hacer uso de mecanismos POSIX para detectar fallos en un sistema. Sin embargo, la implementación de servicios de detección de *fallos en plazos* y c*ómputo en exceso* utilizando funciones POSIX no es sencilla. Las funciones relacionadas con señales, temporizadores y relojes son de gran utilidad pero su uso introduce complejidad en las aplicaciones de tiempo-real y además no están presentes en todos los SOTR. Un ejemplo lo constituyen los mecanismos definidos en POSIX para medir los tiempos de ejecución, que pueden ser utilizados para detectar cuando un thread incurre en *cómputo en exceso* o en el caso de threads periódicos, cuando consumen el crédito de ejecución de su servidor. Pero su uso requiere la creación de temporizadores y de incluir en el código del thread las operaciones necesarias para llevar a cabo el control del tiempo de ejecución, lo que hace compleja la implementación de servicios de tolerancia a fallos. Además, los *relojes y temporizadores CPU-Time de POSIX* están definidos en la *especificación Avanzada de Threads de Tiempo-Real*, que es opcional y por lo tanto no está presente en todo sistema operativo, como ya se ha explicado.

Por otra parte, el modelo de la planificación definida por el usuario incluye un conjunto de funciones que pueden ser utilizadas para implementar mecanismos eficientes para detectar *fallos en plazos* y *cómputo en exceso*, entre las que se incluyen funciones para:

- activar o suspender la ejecución de threads *planificados-por-aplicación*
- informar explícitamente al *PDU* cuando el thread a concluido su ejecución para la presente activación
- armar temporizadores globales (timeouts)
- armar notificaciones específicas por thread

Sin embargo, el uso de estas funciones sigue siendo complejo y requiere de un conocimiento profundo del significado y consecuencias de cada una de ellas. En

[7] scheduling optional computations

el API propuesto, estos inconvenientes se eliminan al integrar en él los servicios de detección de *fallos en plazos* y *cómputo en exceso* de manera directa, y que permiten manejarlos inmediatamente.

Para poder detectar estos errores temporales se pueden definir manejadores a nivel global, y manejadores específicos por thread. Estos manejadores son funciones definidas por el usuario que serán ejecutadas cuando un thread pierda un plazo o cuando incurra en un exceso de cómputo. A continuación se presentan las funciones que permiten definir los manejadores globales o por *PDU*. Estos manejadores serán invocados cuando alguno de los threads planificados por el PDU incurra en alguno de los fallos mencionados.

```
int appschedlib_attr_set_schedhandlers(
    appsched_attr_t * attr,
    void (*overrun_handler) (pthread_t *thread),
    void (*deadline_miss_handler) (pthread_t *thread);

int appschedlib_attr_get_schedhandlers(
    const appsched_attr_t * attr,
    void (*overrun_handler) (pthread_t *thread),
    void (*deadline_miss_handler) (pthread_t *thread);

int appschedlib_set_schedhandlers(
    appschedlib_sched_t * appscheduler,
    void (*overrun_handler) (pthread_t *thread),
    void (*deadline_miss_handler) (pthread_t *thread);

int appschedlib_get_schedhandlers(
    const appschedlib_sched_t * appscheduler,
    void (*overrun_handler) (pthread_t *thread),
    void (*deadline_miss_handler) (pthread_t *thread);
```

Por otra parte, si se requiere la definición de manejadores específicos por thread, que tendrán preferencia sobre los manejadores globales, podrán definirse y conocerse con el uso de:

```
int pthread_attr_set_appschedlib_schedhandlers_np(
    pthread_attr_t * attr,
    void (*overrun_handler) (pthread_t *thread),
    void (*deadline_miss_handler) (pthread_t *thread);

int pthread_attr_get_appschedlib_schedhandlers_np(
```

```
        pthread_attr_t * attr,
        void (*overrun_handler) (pthread_t *thread),
        void (*deadline_miss_handler) (pthread_t *thread);

    int pthread_set_appschedlib_schedhandlers_np(
        pthread_t thread,
        void (*overrun_handler) (pthread_t *thread),
        void (*deadline_miss_handler) (pthread_t *thread);

    int pthread_get_appschedlib_schedhandlers_np(
        pthread_t  thread,
        void (*overrun_handler) (pthread_t *thread),
        void (*deadline_miss_handler) (pthread_t *thread);
```

Las funciones manejadoras, como puede observarse, reciben un único parámetro, que es un puntero al thread que incurrió en el fallo. Esto se debe a que la función manejadora puede ser ejecutada en un contexto diferente al del thread erróneo, y por lo tanto funciones como *pthread_self()* no son de utilidad para detectar a la tarea que contiene el error.

5.5.1. Valor de retorno y errores

Los valores que estas funciones pueden retornar son los siguientes:

[0] Si las funciones tienen éxito, retornarán el valor de cero.

Las funciones *appschedlib_attr_set_schedhandlers(), appschedlib_attr_get_schedhandlers(), pthread_attr_set_appschedlib_schedhandlers_np()* y *pthread_attr_get_appschedlib_schedhandlers_np()* fallarán si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado.

Las funciones *appschedlib_set_schedhandlers()* y *appschedlib_get_schedhandlers()* fallarán si:

[ESRCH] El valor especificado por *appscheduler* no hace referencia a un identificador válido, de alguna política de planificación previamente inicializada.

Las funciones *pthread_set_appschedlib_schedhandlers_np()* y *pthread_get_appschedlib_schedhandlers_np()* fallarán si:

[ESRCH] El valor especificado por *thread* no hace referencia a un thread existente en el momento de invocar la función.

Las funciones *appschedlib_attr_set_schedhandlers(), appschedlib_set_schedhandlers(), pthread_attr_set_appschedlib_schedhandlers_np()* y *pthread_set_appschedlib_schedhandlers_np()* fallarán si:

[EINVAL]] Los valores especificado por *void (*overrun_handler) (pthread_t *thread)* y *void (*deadline_miss_handler) (pthread_t *thread)* no son válidos.

Hasta aquí se han presentado las funciones que forman la parte obligatoria de la interfaz propuesta. Sin embargo, el modelo debe ser lo suficientemente flexible como para soportar políticas de planificación que requieran de la definición de atributos adicionales. En la siguiente sección se presentarán las guías para extender la interfaz y agregarle funciones optativas.

5.6. Definición de nuevas políticas

En esta sección se explica la manera de implementar las funciones necesarias para definir los atributos de alguna política de planificación particular. El objetivo es el de preservar compatibilidad con la interfaz propuesta.

Algunas políticas de planificación requieren de la definición de atributos adicionales de planificación. En caso de que la política implementada requiera de parámetros específicos, deberá crearse la función correspondiente. Por ejemplo, para la política de planificación *round robin* (APPSCHEDLIB_RR), el *quantum* de ejecución se definirá por medio de las siguientes funciones:

```
int appschedlib_attr_set_rrquantum(
          appschedlib_attr_t * attr,
          struct timespec * quantum);

int appschedlib_set_rrquantum(
          appschedlib_sched_t * appscheduler,
          struct timespec * quantum);
```

Además, deberán definirse las funciones que permitan conocer el valor del nuevo atributo. Por ejemplo, para conocer al valor del *quantum* de la política *round robin*, las siguientes funciones deben estar disponibles:

```
int appschedlib_attr_get_rrquantum(
          appschedlib_t * attr,
          struct timespec * quantum);

int appschedlib_get_rrquantum(
          appschedlib_sched_t * appscheduler,
          struct timespec * quantum);
```

La implementación de estas funciones implica que deba incluirse en el objeto de atributos del planificador uno que represente al nuevo parámetro, que en este caso representará al *quantum* con un atributo del tipo *struct timespec*. En el siguiente capítulo se estudiará con detalle la manera de implementar planificadores a nivel de usuario con el modelo propuesto, y se presentarán los elementos que forman al objeto de atributos, así como las guías para agregar nuevos parámetros.

Por otra parte, alguna de las nuevas políticas de planificación pueden requerir la definición de parámetros de los threads planificados por ella. Por ejemplo, si se implementa la política de planificación de prioridad dual[8] [33], el *instante de promoción*[9] del thread se definiría utilizando las siguientes funciones:

```
int pthread_attr_set_appschedlib_promotiontime_np(
          pthread_attr_t * attr,
          struct timespec * prom_time);

int pthread_set_appschedlib_promotiontime_np(
          pthread_t  thread,
          struct timespec * prom_time);
```

De igual manera, para conocer el valor del se utilizarían las funciones cuyo prototipo se muestra a continuación:

```
int pthread_attr_get_appschedlib_promotiontime_np(
          pthread_attr_t * attr,
          struct timespec * prom_time);

int pthread_get_appschedlib_promotiontime_np(
          pthread_t  thread,
          struct timespec * prom_time);
```

Puede observarse que la aplicación no tiene nunca acceso directo a los atributos de las políticas de planificación y threads. En todo caso, deben existir siempre funciones para definir y conocer tales atributos.

[8] dual priority scheduling
[9] promotion time

5.7. Implementación de la propuesta

La interfaz propuesta en este capítulo supone la existencia de un conjunto de planificadores definidos por el usuario y desarrollados siguiendo la metodología propuesta en esta tesis. La interfaz ha sido implementada en RTLinux y se ha medido su sobrecarga, que resultó ser del 1.15 % [40]. La sobrecarga introducida, como puede verse, es despreciable.

En el siguiente capítulo se presentan las guías a seguir para la implementación de los planificadores definidos por el usuario, y para utilizarlos en el marco de referencia propuesto.

Capítulo 6

Creación de Bibliotecas de Planificadores Definidos por el Usuario

En el capítulo anterior se presentó el modelo propuesto para la definición de planificadores a nivel de usuario, que junto con la estrategia de implementación presentada en el capítulo 4, cumple con los características definidas en esta tesis. La interfaz propuesta tiene como objetivo facilitar la implementación de sistemas de tiempo-real, extendiendo el modelo de la planificación definida para el usuario presentada por Aldea y González-Harbour. Hasta el momento se ha explicado la manera en que los nuevos servicios de planificación pueden ser utilizados, pero no se ha explicado cómo implementar las nuevas políticas de planificación utilizando el marco de referencia propuesto. En este capítulo se presentan algunas guías para la implementación de planificadores definidos por el usuario de tal manera que puedan ser utilizados eficientemente con la interfaz propuesta.

6.1. Definición de planificadores

La idea básica consiste en que las políticas de planificación que se implementen estén integradas, al menos de manera conceptual, en una biblioteca de planificadores definidos por el usuario. Si los planificadores se construyen siguiendo algunas reglas sencillas, se podrán utilizar con la interfaz propuesta en el capítulo anterior y se podrán aprovechar las nuevas prestaciones.

El modelo propuesto busca ser lo más abstracto posible, enfocándose más en aspectos cualitativos que permitan la implementación de sistemas de tiempo-real de manera flexible y eficiente. Sin embargo, en muchas ocasiones se explicarán detalles de su implementación en RTLinux para ilustrar una manera en que los

nuevos servicios pueden ser integrados a un SOTR, sin que esto implique que sea la única manera de hacerlo. Se estudiarán primero los aspectos generales para la definición de un PDU, y en secciones posteriores se tratará la definición del PDU de acuerdo a características particulares del sistema a planificar.

Sin importar la manera en que cada PDU sea implementado, estará caracterizado por un conjunto de datos y un conjunto de operaciones. Los datos necesarios para que el planificador lleve a cabo su trabajo están definidos en una estructura que contiene al menos los atributos que a continuación se presentan:

```
struct appschedlib_sched_attr_t {
        appschedlib_policy_t policy;
        int priority;
        int threads_synch_start;
        int threads_count;
        generic_list_t *  threads_list_start;
        struct rtl_thread_struct * current_thread;
        void (* overrun_handler)
                (struct rtl_thread_struct *thread);
        void (* deadline_miss_handler)
                (struct rtl_thread_struct *thread);
        generic_list_t * servers_list_start;
        generic_list_t * synch_threads_list_start;
};
```

La función de la mayoría de los atributos de esta estructura es obvia, pero sin el ánimo de ser exhaustivo, se explican a continuación:

- *policy*: política de planificación a utilizar
- *priority*: prioridad de la política de planificación, que será asignada a los threads planificados por ella
- *threads_synch_start*: si se ha elegido que el conjunto de threads sea síncrono, contiene el número de threads que iniciarán su ejecución en el mismo instante de tiempo
- *threads_count*: número de threads creados actualmente y que son planificados por el planificador
- *threads_list_start*: puntero al inicio de la lista de threads activos
- *overrun_handler*: puntero a la función que será ejecutada si alguno de sus threads incurre en un cómputo en exceso

- *deadline_miss_handler*: puntero a la función que será ejecutada si alguno de sus thread falla algún plazo
- *servers_list_start*: puntero al inicio de la lista de servidores aperiódicos definidos junto con la política de planificación
- *synch_threads_list_start*: puntero al inicio de la lista de threads que inciciarán ejecución simultánea, justo en su correspondiente instante de sincronía

Se ha dicho que todo PDU está compuesto por un conjunto de atributos, mostrados líneas arriba, y un conjunto de operaciones. La totalidad de operaciones primitivas disponibles se muestran en el Cuadro 4.3 en la página 91, y generalmente es tan sólo necesario definir un subconjunto de ellas. Por cada nueva política de planificación, deberán determinarse las operaciones primitivas correspondientes, que podrán variar de acuerdo a cada política y a cada implementación. Sin embargo, en el modelo propuesto, se ha determinado con precisión el conjunto de operaciones que deberán definirse en todo PDU. Para fines prácticos, los planificadores a nivel de usuario han sido clasificadas en tres grupos, de acuerdo al conjunto de tareas que planifique. Los grupos son: planificadores de tareas periódicas, planificadores de tareas aperiódicas, y planificadores de tareas con recursos compartidos. Siempre que se implemente un nuevo planificador, y tomando en consideración el tipo de sistema que planificará, deberán definirse las operaciones correspondientes siguiendo las guías presentadas en este capítulo. De esta manera se tiene un marco de referencia para la construcción de planificadores, y se garantizará la consistencia en su uso, así como su portabilidad.

En la siguiente sección se analizará la construcción de planificadores definidos por el usuario considerando exclusivamente tareas periódicas. En posteriores secciones se analizarán los casos que incluyan tareas aperiódicas y recursos compartidos, y se concluirá el capítulo presentando los atributos adicionales de los threads *planificados-por-aplicación*.

6.2. Planificadores de tareas periódicas

En esta sección se estudiará la construcción de planificadores a nivel de usuarios, siguiendo el modelo propuesto, y que serán utilizados en sistemas compuestos exclusivamente por tareas periódicas críticas. Un ejemplo de este tipo de planificadores es el EDF. Para la construcción de estos planificadores será necesario la definición de al menos las operaciones que aparecen en el Cuadro 6.1. El objetivo y estructura de cada una de estas funciones se explica a continuación.

Es importante destacar que todos los planificadores utilizarán la misma operación *init()*, y que llevará a cabo el conjunto de operaciones iniciales necesarias por

Operaciones del Planificador
init (...)
new_thread (...)
thread_ready (...)
explicit_call (...)
notification_for_thread (...)
signal (...)

Cuadro 6.1: Operaciones de un *PDU* de tareas periódicas

todo planificador. En la implementación llevada a cabo, a la función se le nombra *appschedlib_init()* y su prototipo se muestra a continuación:

```
void appschedlib_init(
       struct appschedlib_sched_attr_t  * sched_data,
       struct appschedlib_init_data_t * init_data);
```

en donde el primer argumento es una estructura que contiene los datos del planificador estudiados anteriormente.

La función *new_thread()* es ejecutada cuando se crea un thread *planificado-por-aplicación*. Recibe como primer parámetro, al igual que el resto de las operaciones primitivas, un puntero a `sched_data`, que contiene la dirección del inicio del área de memoria en la que se encuentran los datos utilizados por el planificador. La función revisa en primera instancia si el usuario ha definido un conjunto de tareas síncrono, lo que determinará si el valor de `threads_synch_start` es mayor que cero. Si ese es el caso, ejecutará al thread y lo marcará como síncrono, en un atributo del thread definido específicamente para ello. Si el conjunto de tareas no es síncrono, se activará el thread inmediatamente.

Antes activar un thread deben revisarse algunos de sus atributos. En primer lugar, debe determinarse su correcto valor de *urgencia*, que dependerá de la política de planificación utilizada. Además, si se ha definido algún manejador para detección de *cómputo en exceso* o *fallos en plazos*, deben programarse los temporizadores correspondientes. También debe programarse un temporizador para el siguiente periodo de ejecución del thread. Para controlar estos tiempos dos mecanismos son utilizados. Para los tiempos de cómputo de los threads se utiliza el mecanismo *POSIX de medición de los tiempos de ejecución (CPU Time*). Tomando en consideración que POSIX establece en [51] que los mecanismos para medir el tiempo de ejecución deben ser definidos por el implementador[1], se han integrado al núcleo del SOTR los servicios para medir el tiempo de ejecución

[1] implementation-defined

Ejemplo 7 Pseudocódigo de la función *new_thread()* para tareas periódicas

```
void new_thread(sched_data, thread, actions, current_time)
{
  si (threads_synch_start >= 0)  {
      incrementar no_threads en 1;
      marcar thread como síncrono;
  } fin si
  calcular valor de urgencia;
  si (manejador de fallos en plazo definido)
          armar notificación para thread
             en el plazo o periodo,
             lo que sea más cercano;
  de otra manera
           armar notificación para thread
             en el siguiente periodo;
  activar thread;
 }
```

de los threads *planificados-por-aplicación* que utilicen el modelo propuesto. De esta manera, cuando un thread es ejecutado su tiempo de cómputo es registrado siempre. Si el thread incurre en *cómputo en exceso* y se ha definido alguno de los manejadores correspondientes, se genera una señal y el *PDU* será informado cuando se invoca la función primitiva *signal()*. El pseudocódigo de esta función puede verse en el Ejemplo 8. Con este mecanismo no es necesario incluir en el código del thread o de la aplicación función alguna para el control del tiempo de ejecución o los cómputos en exceso. La misma estrategia se utiliza para el control del crédito de los servidores aperiódicos, como se verá más adelante. Por otra parte, si el usuario desea utilizar temporizadores para llevar el control del tiempo de cómputo la interfaz lo permite.

El segundo mecanismo relacionado con el tiempo se utiliza para programar el siguiente periodo del thread y para detectar fallos en plazos. En estos casos debe programarse una notificación para el thread al tiempo más próximo de los dos (periodo o plazo), y esto se hace utilizando internamente la función *apps-ched_notification_for_thread()* en las funciones *new_thread()* y *thread_ready()*, entre otras. El pseudocódigo de la función *new_thread()* se muestra en el Ejemplo 7. La función *thread_ready()* tiene una estructura similar a la de *new_thread()*, con la excepción de que no se valida la ejecución simultánea de las tareas síncronas.

Cuando un thread concluye con su ejecución debe utilizar la función que invo-

Ejemplo 8 Pseudocódigo de la función *signal()* para tareas periódicas

```
void signal(sched_data, siginfo, actions, current_time)
{
   en caso de que siginfo.si_signo igual a {

      APPSCHEDLIB_EXEC_TIME_SIGNAL:
         si (manejador nivel thread definido)
            ejecutar manejador nivel thread;
         de otra manera
            si (manejador nivel global definido)
               ejecutar manejador nivel global;
         salir;
   }
}
```

ca explícitamente a su *PDU, posix_appsched_invoke_scheduler()*. En este caso, la operación primitiva invocada por el núcleo del SOTR será la definida en la función *explicit_call()*. En esta función se suspende la ejecución del thread y en el caso de que se utilice el servicio de detección de fallos en plazos, se desarma o reprograma la notificación correspondiente. Además, si el thread es parte del conjunto de threads síncronos, lo agregará a la *lista de threads síncronos* y decrementará el contador `threads_synch_start` en uno. Si después de esto su valor es igual a 0, significa que todos los threads síncronos han alcanzado su punto de sincronía y por lo tanto pone en ejecución a los threads contenidos en la lista de threads síncronos. Cada thread síncrono deja de ser marcado como tal a partir de este punto. El pseudocódigo de esta función para threads periódicos se muestra en el Ejemplo 9.

Cuando ocurre alguna notificación para un thread significa que algún evento temporal relacionado con él ha ocurrido. Ejemplos de esta clase de eventos son el inicio de un nuevo periodo para el thread o el fallo en cumplir un plazo. En el Ejemplo 10 se muestra el pseudocódigo de esta función, llamada *notification_for_thread()*. Si se trata del instante de una nueva activación del thread, se activa después de programar la siguiente notificación. Si el evento que provocó la ejecución de la función es una fallo en un plazo, se ejecuta el manejador correspondiente.

Con las funciones hasta aquí expuestas se pueden construir planificadores para tareas periódicas. La inclusión de tareas aperiódicas se estudia en la siguiente sección.

Ejemplo 9 Pseudocódigo de la función *explicit_call()* para tareas periódicas

```
void explicit_call(sched_data, thread, actions, current_time)
{
 suspender thread;
 si (manejador de fallos en plazo definido)
    desarmar notificación para el thread
      o re-programarla al siguiente periodo
 si (thread es síncrono) {
    agregar thread a lista de threads síncronos;
    decrementar threads_synch_start en 1;
    si (threads_synch_start = 0) {
       por cada thread en lista de threads síncronos, hacer: {
          marcar thread como no síncrono;
          calcular valor de urgencia;
          si (manejador de fallos en plazo definido)
             armar notificación para thread
                en el plazo o periodo,
                lo que sea más cercano;
             de otra manera
                armar notificación para thread
                  en el siguiente periodo;
          activar thread;
    }  fin si
 }  fin si
}
```

Ejemplo 10 Pseudocódigo de la función *notification_for_thread()* para tareas periódicas

```
void notification_for_thread(sched_data, thread,
                             actions, current_time)
 {
   en caso de que thread.appschedlib_state igual a {

      INACTIVE:
     /* significa que su tiempo de activación ha llegado */
        si (manejador de fallos en plazo definido)
           armar nueva notificación para el thread
             para el próximo plazo o periodo,
             lo que sea menor;
        de otra manera
           armar nueva notificación para el thread
             para el próximo  periodo;
        activar thread
        salir;

      ACTIVE:
         /*  significa que ha fallado su plazo */
         si (manejador nivel thread definido)
            ejecutar manejador nivel thread;
         de otra manera
            si (manejador nivel global definido)
               ejecutar manejador nivel global;
         salir;
   }
}
```

Operaciones del Planificador
init (...)
new_thread (...)
thread_ready (...)
explicit_call (...)
notification_for_thread
timeout (...)
signal (...)

Cuadro 6.2: Operaciones de un *PDU* de tareas aperiódicas

6.3. Planificadores de tareas aperiódicas

En esta sección se estudiará la manera de construir planificadores para sistemas formados por tareas periódicas críticas y tareas aperiódicas. Por ejemplo, las políticas EDF y CBS pueden ser utilizadas en sistemas con estas características.

Como se estudió en la Sección 2.3.3, los métodos más comúnmente utilizados para planificar este tipo de sistemas consisten en el uso de *servidores aperiódicos*, que reservan un ancho de banda para la ejecución de las tareas asignadas a ellos. Si bien es cierto que un servidor aperiódico es definido conceptualmente como una tarea, en la práctica y por razones de eficiencia no tiene por que ser así. En el marco de referencia propuesto, cada PDU planifica a los servidores que le han sido asignados, sin necesidad que que sean implementados como threads.

Para el PDU, cada servidor está definido por medio de un conjunto de datos, que son administrados internamente por él y a los que las aplicaciones no tienen acceso. En el modelo, todo servidor debe definirse con una estructura de datos que contenga al menos los valores que se muestran a continuación:

```
struct appschedlib_server_attr_t {
        struct appschedlib_server_attr_t * next;
        int id;
        int threads_count;
        struct timespec urgency;
        int priority;
        posix_appsched_scheduler_id_t * appscheduler;
        struct timespec period;
        struct timespec budget;
        struct timespec current_budget;
        struct timespec current_deadline;
        generic_list_t * threads_list_start;
```

```
        struct rtl_thread_struct * current_thread;
        appschedlib_server_state_t state;
};
```

La función de cada atributo se explica brevemente:

- *next*: utilizado por el planificador para ordenar a los servidores en su lista
- *id*: identificador del servidor
- *threads_count*: contador del número de threads asignados al servidor
- *urgency*: valor de urgencia del servidor, que será asignado a los threads que active
- *priority*: prioridad fija del servidor, utilizada por algunas políticas
- *appscheduler*: planificador del servidor
- *period*: periodo del servidor
- *budget*: crédito máximo del servidor
- *current_budget*: crédito actual del servidor
- *current_deadline*: plazo actual del servidor
- *threads_list_start*: puntero a la lista de threads activos del servidor
- *current_thread*: de los threads activos del servidor, el que actualmente esta siendo planificado por el PDU
- *state*: estado del servidor

Atributos adicionales podrán agregarse, en caso de que alguna política de planificación lo requiriese.

Además de los datos del planificador presentados en la sección anterior, y de los datos del servidor, el PDU requerirá de la definición de un conjunto de operaciones primitivas. Las operaciones necesarias para la planificación de threads periódicos que se presentaron en el Cuadro 6.1 en la página 146 deberán modificarse para planificar un sistema con tareas periódicas y aperiódicas, y deberá agregarse la operación *timeout()*, como se muestra en el Cuadro 6.2. Esta función se usará para llevar el control de las activaciones periódicas de los servidores, como se explica a continuación.

Ejemplo 11 Pseudocódigo de la función `timeout()` para tareas aperiódicas

```
void timeout(sched_data, actions, current_time)
{
   por cada servidor  en lista, hacer;
   si (next_server) {
      activar sus threads aperiódicos
        controlando el consumo de su crédito
        hasta consumir crédito del servidor
        o no haya más threads aperiódicos
        por ejecutar;
   }
   determinar siguiente periodo de next_server;
   colocar next_server en lista:
   next_server = servidor que encabece lista;
   armar timeout en
      siguiente periodo de next_server:
 }
```

Debido a que cada PDU debe administrar el conjunto de servidores definidos bajo su política, cada uno de ellos será agregado a la lista de servidores del planificador, que estará ordenada por periodo o instante de activación. Una de las funciones del planificador es la de llevar control de las activaciones periódicas de cada servidor. Para esto, programará temporizadores globales utilizando la función *posix_appsched_actions_addtimeout()*. Como sólo puede utilizarse un temporizador global por planificador, deberá programarse para el instante de activación del servidor que encabece la lista. Cuando el temporizador expire, se ejecutará la operación *timeout()*, en la que el planificador activará al servidor correspondiente. El pseudocódigo de esta función se muestra en el Ejemplo 11.

Por otra parte, cada servidor debe llevar un control sobre el consumo del crédito de los threads que planifique. Para vigilar que las restricciones de crédito de ejecución se satisfagan, cada servidor sólo podrá activar uno de sus thread a la vez, lo que significa que en cada instante la cola de threads activos del sistema operativo contendrá a lo más un thread aperiódico por servidor. Esto no ocurre con las tareas periódicas, ya que pueden activarse inmediatamente con un valor de urgencia adecuado, pues es utilizado por el planificador del sistema como segundo criterio de prioridad. Sin embargo, para el caso de las tareas aperiódicas, el control del tiempo de ejecución obliga al servidor a llevar una lista de sus threads activos, de tal manera que cuando el PDU correspondiente los planifique, lo haga con los parámetros correctos, a saber: crédito, plazo y valor de urgencia.

Ejemplo 12 Pseudocódigo de t`hread_ready()` para tareas aperiódicas

```
void thread_ready(sched_data, thread, actions, current_time)
{
   si (el thread es aperiódico)
      agregar thread a lista
        del servidor correspondiente;
        si (servidor inactivo)
           activarlo en caso de que su
             algoritmo lo especifique;
   de otra manera {
      calcular valor de urgencia;
      si (manejador de fallos en plazo definido)
              armar notificación para thread
                 en el plazo o periodo,
                 lo que sea más cercano;
      de otra manera
               armar notificación para thread
                 en el siguiente periodo;
      activar thread;
   } fin si - de otra manera
}
```

Ejemplo 13 Pseudocódigo de la función *signal()* para tareas aperiódicas

```
void signal(sched_data, siginfo, actions, current_time)
{
   en caso de que siginfo.si_signo igual a {

      APPSCHEDLIB_EXEC_TIME_SIGNAL:
         si (thread es aperiódico) {
           suspender thread;
           agregar thread a lista de threads
               de su servidor;
           aplicar reglas de consumo y recarga
               de su servidor;
           actualizar lista de servidores
              y re-programar temporizador global
              en caso de ser necesario;
         } de otra manera {
            si (manejador nivel thread definido)
               ejecutar manejador nivel thread;
            de otra manera
               si (manejador nivel global definido)
                  ejecutar manejador nivel global;
         } fin si - de otra manera
         salir;
}
```

Para llevar control del consumo de crédito, siempre que un thread aperiódico se active será colocado en la lista de threads del servidor respectivo. Cuando el servidor sea activado, elegirá el thread aperiódico por ejecutar de acuerdo a su algoritmo. Una vez que lo haya seleccionado, deberá calcular su valor de urgencia, que dependerá de la política de planificación de cada PDU. Por ejemplo, para el caso de un PDU que utilice las políticas EDF y CBS, cada servidor CBS utilizará el inverso del valor de su plazo como valor de urgencia del thread que active, mismo que será utilizado por el EDF para planificar al thread.

Se ha dicho que las funciones que llevan a cabo las activaciones de los threads periódicos deben ser modificadas para que, cuando una tarea aperiódica llegue al sistema, en lugar de ser activada directamente por el PDU, sea agregada a la cola de threads del servidor correspondiente. Las funciones que son afectadas con estos cambios son *new_thread()* y *thread_ready()*. El pseudocódigo de esta última se muestra en el Ejemplo 12. Puede observarse que se verifica si el thread es aperiódico, para que en ese caso sea agregado a la lista de threads activos de su servidor. Además, hay que considerar el caso de algunos algoritmos que especifican que si el servidor está inactivo cuando alguno de sus threads se libera, debe generarse un nuevo plazo para el servidor y activarse de acuerdo a las reglas del PDU. Considerando nuevamente el ejemplo del EDF y CBS, el servidor será planificado por el EDF de acuerdo a su nuevo plazo. El pseudocódigo de la función *new_thread()* no se presenta ya que es muy similar al de *thread_ready()*, con la diferencia de que debe considerar el caso de la activación de tareas síncronas, que se estudió en la sección anterior. Para el caso de la función *explicit_call()*, el único cambio consiste en que cuando un thread aperiódico concluye con su ejecución, deben ajustarse los valores de crédito y plazo del servidor correspondiente, de acuerdo a lo estipulado en su propio algoritmo.

Por otra parte, si se utilizan temporizadores CPU-Time para el control del consumo del crédito, la función *signal()* debe también modificarse para que ejecute las operaciones necesarias cuando un thread consume su crédito. Su pseudocódigo se muestra en el Ejemplo 13. Si una tarea aperiódica consume su crédito, se genera una señal y se ejecuta la función *signal()*. En ella, se suspende al thread aperiódico y se aplican las reglas de consumo y recarga del servidor. Una vez hecho esto, el servidor puede activar alguno de sus threads con los nuevos valores. Otra alternativa mas sencilla, es la de utilizar los mecanismos de la interfaz propuesta para el control del tiempo de ejecución. De esta manera, utilizando la función *pthread_set_appschedlib_maxexectime()* es posible establecer el crédito de ejecución disponible para el thread, y en caso de que se consuma, se ejecutará la operación *thread_block()*. El código de esta función sería similar al de *signal()*.

Es importante mencionar que el uso de un servidor aperiódico es una estrategia para controlar el tiempo de cómputo de un thread, por lo que no tendría sentido definir manejadores de cómputo en exceso para tareas aperiódicas, así que en

Operaciones del Planificador
init (...)
new_thread (...)
thread_ready (...)
explicit_call (...)
notification_for_thread
timeout (...)
signal (...)
lock_mutex(...)
trylock_mutex(...)
unlock_mutex(...)

Cuadro 6.3: Operaciones de un *PDU* con recursos compartidos

caso de ser definidos se ignoran y se ejecutan en su lugar las reglas de consumo y reposición del servidor correspondiente.

6.4. Sincronización

La sincronización de tareas puede gestionarse utilizando el protocolo SRP y las versiones POSIX de los protocolos PIP y PCP. Para los casos particulares del PCP y SRP, es necesario la definición *off-line* de los valores de *techo de prioridad* y de *techo de nivel de expulsión* de los monitores, y una vez definidos deben ser asignados utilizando las funciones correspondientes que se presentaron en la Sección 4.9.3.6 en la página 103. Sin embargo, este enfoque no es flexible y puede introducir errores si no se asignan correctamente los valores de techo de prioridad o de nivel de expulsión. En la interfaz propuesta se ha integrado una nueva función, presentada por Balbastre y Ripoll en [13], para eliminar este tipo de errores y para flexibilizar el uso de monitores. La función se muestra a continuación:

```
int pthread_mutex_register_np(pthread_t thread,
                              pthread_mutex_t *mutex);
```

La función deberá ser utilizada para *registrar* todos los threads que utilizarán el monitor, y determinará automáticamente los valores de techo de prioridad y de techo de nivel de expulsión correctos.

Debido a que los protocolos de sincronización arriba descritos han sido integrados al núcleo del sistema operativo al implementar el modelo para la planificación definida por el usuario, no es necesario administrar el acceso a recursos

en el código de los planificadores definidos por el usuario, a excepción de cuando se utilizan tareas aperiódicas. Si tal es el caso, deberán programarse las funciones primitivas que se invocarán cuando un thread *planificado-por-aplicación* invoque a alguna de las funciones relacionadas con los monitores. Las operaciones que deberán ser implementadas para planificar conjuntos de tareas periódicas y aperiódicas que compartan recursos entre sí se muestra en el Cuadro 6.3. El código de dichas funciones, en caso de ser necesario definirlas, dependerá de la política de sincronización implementada.

La interfaz propuesta, como se ha explicado, incluye funciones que permiten implementar la gran mayoría de algoritmos de planificación para tareas aperiódicas con recursos compartidos publicadas a la fecha. Es posible llevar control del consumo de crédito de los threads, y es posible administrar la entrada a las secciones críticas.

Una vez que se han estudiado las guías para la implementación de planificadores, en la siguiente sección se presentan los atributos adicionales definidos para los threads *planificados-por-aplicación* del modelo propuesto. Con ellos, el PDU podrá planificarlos correctamente.

6.5. Threads planificados por aplicación

Los atributos para threads que se muestran en el Cuadro 6.4, han sido definidos para permitir su planificación utilizando el modelo propuesto.

Se ha explicado que el modelo busca ser lo suficientemente flexible como para permitir la implementación de la mayoría de las políticas de planificación disponibles. Por esta razón, la lista de atributos presentada en esta sección puede incrementarse para incluir atributos adicionales, en caso de que alguna implementación los requiera. Lo importante a destacar es que en todo caso deben incluirse las funciones que permitan establecer o conocer el valor de tales atributos, de tal manera que las aplicaciones no puedan acceder a ellos los directamente, sino a través de las funciones correspondientes.

Una vez que se ha presentado el marco de referencia propuesto, en el siguiente capítulo se presentarán algunos ejemplos y casos de estudio, con la finalidad de clarificar su uso.

Tipo	Nombre	Descripción
appschedlib_policy_t	`policy`	Política de planificación
*struct timespec **	`appschedlib_period`	Periodo
*struct timespec **	`deadline`	Plazo relativo
*struct timespec **	`next_deadline`	Plazo absoluto
*struct timespec **	`wcet`	Máximo tiempo de cómputo
*struct timespec **	`exec_time`	Tiempo de ejecución de última activación
int	`appschedlib_priority`	Prioridad
appschedlib_server_t	`server`	Servidor
int	`synch_flag`	Bandera para thread síncrono
*void **	`(*deadline_miss_handler) (pthread_t *thread)`	Puntero al manejador de fallos en plazos
*void **	`(*overrun_handler) (pthred_t *thread)`	Puntero al manejadors de fallos en cómputo en exceso

Cuadro 6.4: Atributos de los threads *planificados-por-aplicación*

Capítulo 7

Ejemplos y Casos de Estudio

7.1. Ejemplo: EDF

En este capítulo se mostrarán algunos ejemplos que muestren la implementación de planificadores definidos por el usuario, así como el uso de los servicios de planificación construidos con las propuestas estudiadas en esta tesis. En primer lugar, se presentará la implementación de la política de planificación EDF utilizando el modelo para la planificación definida por el usuario implementada en RTLinux.

Como se estudió en la Sección 2.3.2, la política EDF asigna una prioridad dinámica a las tareas inversamente proporcional al valor de sus plazos absolutos. Para implementarla, sin considerar servicios de tolerancia a fallos, tan sólo es necesario definir las funciones primitivas que aparecen en el Cuadro 7.1. En la implementación existente en RTLinux del modelo para la planificación definida por el usuario [38] se hace una distinción en la función primitiva utilizada para invocar al planificador una vez que ha concluido su ejecución. Si se hace utilizando la función *posix_appsched_invoke_scheduler()* se usarán las operaciones mostradas en el Cuadro 7.1. De otra manera, si se utilizan funciones POSIX como *clock_nanosleep()*, o del API de RTLinux como *pthread_wait_np()*, se sustituiría la operación *explicit_call()* y su código se ejecutaría por la operación *thread_block()*. El pseudo-código de la función *new_thread()* se muestra en el Ejemplo 14, que es el mismo para la función *thread_ready()*. La función ex*plicit_call()* incluye tan sólo la instrucción para suspender la ejecución del thread.

Una vez definidas las operaciones primitivas necesarias es posible utilizar la política de planificación EDF. El código de la función *init_module()* de una aplicación que utilice las operaciones del EDF se muestra en el Ejemplos 15 y 16.

Utilizando el modelo propuesto en esta tesis, la implementación del sistema

Operaciones del Planificador
new_thread (...)
thread_ready (...)
explicit_call (...)

Cuadro 7.1: Operaciones para implementar la política EDF

Ejemplo 14 Función `new_thread()` para el EDF

```
void new_thread(sched_data. thread, actions, current_time)
{
   obtener parámetros definidos por
     el usuario del thread;
   hacer next_deadline = current_time
                       + parámetros_thread.period
   hacer urgencia = ~(next_deadline);
   activar thread con valor de urgencia;
 }
```

de tiempo-real se simplifica muchísimo, como puede observarse en el código de la función *init_module()* que se muestra en el Ejemplo 17.

Ejemplo 15 Función *init_module()* utilizando el EDF

```
. . .
pthread_t thread;
 /* variable que contiene parámetros
    definidos por el usuario        */
edf_th_t edf_th;
 /* se crea estructura con punteros
    a las funciones primitivas      */
 static posix_appsched_scheduler_ops_t edf_scheduler_ops =
 {
   NULL,                  /*  init()                     */
   (void *) new_thread,   /*  new_thread()               */
   NULL,                  /*  thread_terminate()         */
   (void *) thread_ready, /*  thread_ready()             */
   (void *) thread_block, /*  thread_block()             */
   NULL,                  /*  thread_yield()             */
   NULL,                  /*  change_sched_param_thread()*/
   NULL,                  /*  explicit_call()            */
   NULL,                  /*  explicit_call_with_data()  */
   NULL,                  /*  notification_for_thread()  */
   NULL,                  /*  timeout()                  */
   NULL,                  /*  signal()                   */
   NULL,                  /*  priority_inherit()         */
   NULL,                  /*  priority_uninherit()       */
   NULL,                  /*  lock_mutex()               */
   NULL,                  /*  trylock_mutex()            */
   NULL,                  /*  unlock_mutex()             */
   NULL                   /*  appsched_error()           */
};
 /* se define identificador de planificador */
static posix_appsched_scheduler_id_t edf_sched_id;
. . .
```

Ejemplo 16 Continuación de la función *init_module()* utilizando el EDF

```
int init_module(void)
{
   pthread_attr_t attr;
   struct sched_param sched_param

 /* se crea el planificador */
   posix_appsched_scheduler_create(&edf_scheduler_ops,
                           0, NULL, 0, &edf_sched_id);

   pthread_attr_init (&attr);
   sched_param.sched_priority = 10;

/* se define el periodo del thread */
  edf_th.period = 30*ONEMILI;

 /* se asignan parámetros definidos
    por el usuario (periodo)            */
   pthread_attr_setappschedparam(
                &attr,(void *) &edf_t, sizeof(edf_th));

   pthread_attr_setschedparam (&attr, &sched_param);
 /* se define el PDU del thread  */
  pthread_attr_setappscheduler(&attr, edf_sched_id);

/* se crea el thread */
   pthread_create (&thread, &attr, thread_code, NULL);
   return 0;
 }
```

Ejemplo 17 Función *init_module()* usando el EDF con el modelo propuesto

```
. . .
pthread_t thread;

 /* se define identificador de planificador */
static appschedlib_sched_t edf_sched_id;
. . .
 int init_module(void)
{
   pthread_attr_t attr;
   srtuct timespec period;

 /* se define la política de planificación */
   appschedlib_init_schedpolicy(&edf_sched_id,
                  NULL, APPSCHEDLIB_EDF, 10);

   pthread_attr_init (&attr);

/* se define el periodo del thread */
   period.tv_sec = 0;
   period.tv_nsec = 30*ONEMILI;
   pthread_attr_set_appschedlib_period_np(
                          &attr, &period);

 /* se define la política de planificación del thread */
   pthread_attr_set_appschedlib_scheduler_np(
                          &attr, edf_sched_id);

 /* se crea el thread */
   pthread_create (&thread, &attr, thread_code, NULL);
   return 0;
 }
```

Capítulo 8

Conclusiones y líneas de trabajo futuras

En la Sección 1.2 se establecieron los objetivos de esta tesis. A continuación se presentan los resultados obtenidos en cada uno de estos objetivos.

8.1. Planificación Definida por el Usuario

La teoría de planificación de sistemas de tiempo-real ha madurado lo suficiente como para permitir la implementación de sistemas complejos, que no son correctamente planificados utilizando exclusivamente planificación basada en prioridades fijas. Sin embargo, es poco común encontrar otro esquema de planificación en los sistemas operativos de tiempo-real existentes. En esta tesis se ha abordado la problemática de añadir políticas adicionales a un sistema operativo, sin modificar su estructura interna. Se buscó definir un modelo que incluyera una interfaz completa y que permitiera la implementación de la mayoría de las políticas de planificación existentes a la fecha. También se buscó que el modelo fuese portátil, flexible, eficiente, y preferentemente compatible con algún estándar de tiempo-real. Después de llevar a cabo una investigación del estado del arte de la planificación definida por el usuario, se concluyó que el modelo propuesto por Aldea y González-Harbour [7] representa un excelente punto de partida para definir el modelo buscado.

En esta tesis se ha extendido el modelo para la planificación definida por el usuario compatible con POSIX, propuesto por Aldea y González-Harbour, para obtener un modelo que permita agregar servicios de planificación cumpliendo con los objetivos definidos inicialmente. Además, se han integrado servicios adicionales que permiten el desarrollo de sistemas de tiempo-real más completos.

El modelo propuesto fue implementado en RTLinux de tal manera que las

operaciones de planificación sean ejecutadas en el contexto del núcleo. Con esta estrategia, se evita el uso de colas de eventos de planificación o la creación de tareas especiales para implementar los algoritmos de planificación.

En conclusión, se ha propuesto un modelo para la planificación definida por el usuario que permite agregar políticas de planificación a los sistemas operativos de tiempo-real existentes, y que contiene una interfaz orientada a los desarrolladores de sistemas de tiempo-real, que aisla los aspectos relacionados con la planificación del código de las aplicaciones. Además, la implementación hecha en RTLinux presenta una sobrecarga despreciable.

8.2. Marco de Referencia

Otra de las aportaciones de esta tesis consiste en que el modelo propuesto sirve como marco de referencia para la planificación definida por el usuario. Además de extender la interfaz para el desarrollo de planificadores, y de presentar una interfaz para el desarrollo de aplicaciones de tiempo real, y de proponer una estrategia de implementación que reduzca la sobrecarga, se han presentado las guías para la construcción de nuevas políticas de planificación, de tal manera que su uso sea consistente y favorecer así la portabilidad de las aplicaciones, su complejidad, el uso del modelo para la planificación definida por el usuario puede provocar que se implementen planificadores cuyo uso no sea consistente.

Otro aspecto interesante lo constituye la posibilidad que ofrece el marco de referencia propuesto para agregar servicios no existentes en POSIX, o que si existen su uso es complejo o en el peor de los casos, no están presentes en todo sistema operativo. Ejemplos de este tipo son el uso de relojes, temporizadores y señales de tiempo-real, o prestaciones para medir el tiempo de cómputo real de los threads, entre otros, con los que se pueden implementar mecanismos para la definición de conjuntos de tareas síncronas, para la definición de tareas periódicas o servicios de tolerancia a fallos. Además, se han integrado nuevos servicios que serán de gran utilidad en la implementación de sistemas de tiempo-real. Por ejemplo, puede medirse fácilmente el tiempo de procesador utilizado por cada tarea, lo que permite la evaluación de sistemas de tiempo-real de manera sencilla. También se han integrado servicios de detección de *fallos en plazos* y *cómputo en exceso*, y es posible definir manejadores globales y locales que se ejecutarán cuando alguno de estos fallos ocurra.

En el Capítulo 6 se definieron las guías para construir planificadores definidos por el usuario, que se integrarán al menos conceptualmente en bibliotecas, y que serán utilizados con la interfaz propuesta. Se explicó la manera de implementar nuevas políticas de planificación considerando diferentes modelos de sistema.

Las propuestas hechas en esta tesis han sido implementadas con éxito en RTLi-

nux.

8.3. Líneas de trabajo futuras

El marco de referencia propuesto ofrece un número ilimitado de posibilidades para facilitar la implementación de sistemas de tiempo-real. El uso de estándares existentes, como POSIX, para este propósito exige de conocimientos profundos de programación, teoría de planificación y sistemas de tiempo-real. Por otra parte, quienes implementan sistemas de tiempo-real deben incluir este tipo de mecanismos en el código de las aplicaciones y se ha demostrado que su uso no es trivial.

Las prestaciones de tolerancia a fallos integradas al modelo son limitadas ya que un fallo en el hardware del sistema no puede ser recuperado, por lo que se propone la integración de servicios distribuidos de tolerancia a fallos, o de cambios de modo.

Apéndice A

Definiciones de Datos

Errores

Los nombres simbólicos que se muestran a continuación, representan los números de error que deben ser definidos en <*errno.h*>.

[EREJECT] El thread que solicita ser planificado por un PDU ha sido rechazado por ese PDU. Esta nueva condición de error puede generarse por una llamada a la función *pthread_create()* al crear un thread *planificado-por-aplicación*, o cuando un thread llama a la función *pthread_setschedparam()* para cambiar su PDU.

[EPOLICY] La política de planificación o el atributo de estado del PDU del thread que hace la llamada no es válida para esta operación.

Definiciones de tipos de datos

Los siguientes definiciones de tipos de datos deben hacerse en <*sched.h*>. El tipo *n/a* indica que es definido por la implementación:

```
typedef long long posix_appsched_urgency_t;
typedef n/a posix_appsched_actions_t;
typedef n/a posix_appsched_scheduler_id_t;
```

Política de Planificación y Atributos

La siguiente política de planificación debe ser definida en <*sched.h*>. Esta política no debe ser utilizada para planificar al proceso:

Símbolo	Descripción
SCHED_APP	Política de planificación definida por el usuario

En <*sched.h*> los siguientes nuevos atributos del thread deben ser definidos *:*

Tipo	Nombre	Descripción
posix_appsched_scheduler_t	`appscheduler`	PDU que planificará al thread
n/a	`appsched_param`	Parámetros para la planificación definida por el usuario
posix_appsched_urgency_t	`urgency`	Urgencia
unsigned short int	`level`	Nivel de expulsión

En <*appschedlib.h*> los siguientes nuevos atributos del thread deben ser definidos:

Tipo	Nombre	Descripción
appschedlib_policy_t	`policy`	Política de planificación
*struct timespec **	`appschedlib_period`	Periodo
*struct timespec **	`deadline`	Plazo relativo
*struct timespec **	`next_deadline`	Plazo absoluto
*struct timespec **	`wcet`	Máximo tiempo de cómputo
*struct timespec **	`exec_time`	Tiempo de ejecución de última activación
int	`appschedlib_priority`	Prioridad
appschedlib_server_t	`server`	Servidor
int	`synch_flag`	Bandera para thread síncrono
*void **	`deadline_miss_handler`	Puntero al manejador de fallos en plazos
*void **	`overrun_handler`	Puntero al manejador de fallos en cómputo en exceso

Objeto PDU

Los siguientes tipos de datos deben estar definidos en <*sched.h*>:

```
typedef struct {
   void (*init) (void * sched_data, void * arg);

   void (*new_thread) (
            void * sched_data,
            pthread_t thread,
            posix_appsched_actions_t * actions,
            struct timespec *current_time);

   void (*thread_terminate) (
            void * sched_data,
            pthread_t thread,
            posix_appsched_actions_t * actions,
            struct timespec *current_time);

   void (*thread_ready) (
            void * sched_data,
            pthread_t thread,
            posix_appsched_actions_t * actions,
            struct timespec *current_time);

   void (*thread_block) (
            void * sched_data,
            pthread_t thread,
            posix_appsched_actions_t * actions,
            struct timespec *current_time);

   void (*thread_yield) (
            void * sched_data,
            pthread_t thread,
            posix_appsched_actions_t * actions,
            struct timespec *current_time);

   void (*change_sched_param_thread) (
            void * sched_data,
            pthread_t thread,
            posix_appsched_actions_t * actions,
```

```
            struct timespec *current_time);

        void (*explicit_call) (
            void * sched_data,
            pthread_t thread,
            int user_event_code,
            posix_appsched_actions_t * actions,
            struct timespec *current_time);

        void (*explicit_call_with_data) (
            void * sched_data,
            pthread_t thread,
            const void * msg,
            size_t msg_size,
            void *reply, size_t *reply_size,
            posix_appsched_actions_t * actions,
            struct timespec *current_time);

        void (*notification_for_thread) (
            void * sched_data,
            pthread_t thread,
            posix_appsched_actions_t * actions,
            struct timespec *current_time);

        void (*timeout) (
            void * sched_data,
            posix_appsched_actions_t * actions,
            struct timespec *current_time);

        void (*signal) (
            void * sched_data,
            siginfo_t siginfo,
            posix_appsched_actions_t * actions,
            struct timespec *current_time);

        void (*priority_inherit) (
            void * sched_data,
            pthread_t thread,
            int sched_priority,
            posix_appsched_actions_t * actions,
            struct timespec *current_time);
```

```
   void (*priority_uninherit) (
           void * sched_data,
           pthread_t thread,
           int sched_priority,
           posix_appsched_actions_t * actions,
           struct timespec *current_time);
void (*lock_mutex) (
           void * sched_data,
           pthread_t thread,
           pthread_mutex_t * mutex,
           posix_appsched_actions_t * actions,
           struct timespec *current_time);

   void (*trylock_mutex) (
           void * sched_data,
           pthread_t thread,
           pthread_mutex_t * mutex,
           posix_appsched_actions_t * actions,
           struct timespec *current_time);

   void (*unlock_mutex) (
           void * sched_data,
           pthread_t thread,
           pthread_mutex_t * mutex,
           posix_appsched_actions_t * actions,
           struct timespec *current_time);

   void (*appsched_error) (
           void * sched_data,
           pthread_t thread,
           posix_appsched_error_cause_t cause,
           posix_appsched_actions_t * actions);

} posix_appsched_scheduler_ops_t;
```

Apéndice B

Funciones

appschedlib_attr_init

Nombre

appschedlib_attr_init - Inicialización del objeto de atributos.

Sinopsis

```
#include <appschedlib.h>

int appschedlib_attr_init(
                appschedlib_attr_t * attr);
```

Descripción

La función *appschedlib_attr_init()* inicializa un objeto de atributos *attr* del planificador con los valores por defecto, para todos los atributos individuales utilizados de una implementación determinada. Los atributos del objeto, que pueden ser modificados por medio de las funciones correspondientes, cuando sea utilizado por la función *appschedlib_init_schedpolicy()*, definirán los atributos de la política de planificación. El mismo objeto de atributos podrá ser utilizado en llamadas simultáneas a la función *appschedlib_init_schedpolicy()*.

Valor de retorno

Si se completa exitosamente, la función *appschedlib_attr_init()* retornará el valor de 0.

Errores

La función *appschedlib_attr_init()* fallará si:

[ENOMEM] No existe suficiente memoria para inicializar el objetos de atributos del planificador.

appschedlib_attr_destroy

Nombre

appschedlib_attr_destroy - Destrucción del objeto de atributos.

Sinopsis

```
#include <appschedlib.h>

int appschedlib_attr_destroy(
                appschedlib_attr_t * attr);
```

Descripción

La función *appschedlib_attr_destroy()* destruirá un objeto de atributos *attr* del planificador. Una implementación determinada puede establecer como inválidos los valores de los atributos del objeto después de utilizar esta función. Un objeto de atributos *attr* puede ser reinicializado utilizando la función *appschedlib_attr_init()*. Los resultados de utilizar un objeto después de que ha sido destruido no están definidos.

Valor de retorno

Si se completa exitosamente, la función *appschedlib_attr_destroy()* retornará el valor de 0.

Errores

La función *appschedlib_attr_destroy()* fallará si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado.

appschedlib_attr_set_synchstart

Nombre

appschedlib_attr_set_synchstart - Establece el atributo de inicio síncrono.

Sinopsis

```
#include <appschedlib.h>

int appschedlib_attr_set_synchstart(
                    appsched_attr_t * attr,
                    int n);
```

Descripción

La función *appschedlib_attr_set_synchstart()* establece el valor del atributo de inicio síncrono del planificador en el objeto de atributos *attr*. El uso de la función provocará que los primeros *n* threads inicien su ejecución de manera simultánea, a partir del punto de sincronía. En el código del thread, el punto de sincronía se establece llamando a la función *posix_appsched_invoke_scheduler()*.

Valor de retorno

Si se completa exitosamente, la función *appschedlib_attr_set_synchstart()* retornará el valor de 0.

Errores

La función *appschedlib_attr_set_synchstart()* fallará si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado, o el valor especificado por *n* no es válido.

appschedlib_attr_get_synchstart

Nombre

appschedlib_attr_get_synchstart - Obtiene el atributo de inicio síncrono.

Sinopsis

```
#include <appschedlib.h>

int appschedlib_attr_get_synchstart(
                    appsched_attr_t * attr,
                    int * n);
```

Descripción

La función *appschedlib_attr_get_synchstart() obtiene* el valor del el atributo de inicio síncrono del planificador del objeto de atributos *attr*. El uso de la función provocará que *n* contenga el número de threads que iniciarán ejecución de manera simultánea, a partir del punto de sincronía.

Valor de retorno

Si se completa exitosamente, la función *appschedlib_attr_get_synchstart()* retornará el valor de 0.

Errores

La función *appschedlib_attr_get_synchstart()* fallará si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado.

appschedlib_attr_set_schedhandlers

Nombre

appschedlib_attr_set_schedhandlers - Establece el atributo que especifica los manejadores de tolerancia a fallos.

Sinopsis

```
#include <appschedlib.h>

int appschedlib_attr_set_schedhandlers(
        appsched_attr_t * attr,
          void (*overrun_handler)
                  (pthread_t *thread),
          void (*deadline_miss_handler)
                  (pthread_t *thread);
```

Descripción

La función *appschedlib_attr_set_schedhandlers()* establece en el objeto de atributos *attr* los punteros a las funciones manejadoras de cómputo en exceso y fallos en plazo del planificador. Las funciones manejadores serán ejecutadas si alguno de los threads planificados por el PDU experimenta alguno de los fallos descritos.

Valor de retorno

Si se completa exitosamente, la función *appschedlib_attr_set_schedhandlers()* retornará el valor de 0.

Errores

La función *appschedlib_attr_set_schedhandlers()* fallará si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado, o los valores especificado por *void (*overrun_handler) (pthread_t *thread)* y *void (*deadline_miss_handler) (pthread_t *thread)* no son válidos.

appschedlib_attr_get_schedhandlers

Nombre

appschedlib_attr_get_schedhandlers - Obtiene el atributo que especifica los manejadores de tolerancia a fallos.

Sinopsis

```
#include <appschedlib.h>

int appschedlib_attr_get_schedhandlers(
         const appsched_attr_t * attr,
           void (*overrun_handler)
                   (pthread_t *thread),
           void (*deadline_miss_handler)
                   (pthread_t *thread);
```

Descripción

La función *appschedlib_attr_get_schedhandlers()* obtiene los valores de los punteros a las funciones manejadoras de cómputo en exceso y fallos en plazo del planificador, contenidos en el objeto de atributos *attr*.

Valor de retorno

Si se completa exitosamente, la función *appschedlib_attr_get_schedhandlers()* retornará el valor de 0.

Errores

La función *appschedlib_attr_get_schedhandlers()* fallará si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado.

appschedlib_attr_set_server

Nombre

appschedlib_attr_set_server - Establece el atributo que especifica un servidor.

Sinopsis

```
#include <appschedlib.h>

int appschedlib_attr_set_server(
        const appsched_attr_t * attr,
          appschedlib_server_t * server,
          struct timespec * period
          struct timespec * budget);
```

Descripción

La función *appschedlib_attr_set_server()* establece en el objeto de atributos *attr*, el servidor *server*, así como su de periodo *period* y crédito *budget*. El servidor será planificado con la política de planificación definida en el PDU. Puede definirse más de un servidor por PDU. Si se completa con éxito, la función *appschedlib_attr_set_sever()* almacenará en el lugar al que hace referencia *server*, el valor del identificador del servidor.

Valor de retorno

Si se completa exitosamente, la función *appschedlib_attr_set_server()* retornará el valor de 0.

Errores

La función *appschedlib_attr_set_server()* fallará si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado, o el valor especificado por *server* no hace referencia a un objeto identificador de servidor válido, o los valores especificado por *period* o *budget* no son válidos.

appschedlib_attr_get_server

Nombre

appschedlib_attr_get_server - Obtiene el atributo que especifica un servidor.

Sinopsis

```
#include <appschedlib.h>

int appschedlib_attr_get_server(
          const appsched_attr_t * attr,
          const appschedlib_server_t * server,
          struct timespec * period
          struct timespec * budget);
```

Descripción

La función *appschedlib_attr_get_server()* obtiene del objeto de atributos *attr* y del servidor *server*, los valores del periodo *period* y del crédito *budget*.

Valor de retorno

Si se completa exitosamente, la función *appschedlib_attr_get_server()* retornará el valor de 0.

Errores

La función *appschedlib_attr_get_server()* fallará si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado, o el valor especificado por *server* no hace referencia a un objeto identificador de servidor válido.

appschedlib_init_schedpolicy

Nombre

appschedlib_init_schedpolicy - Establece la política de planificación.

Sinopsis

```
#include <appschedlib.h>

int appschedlib_init_schedpolicy(
      appschedlib_sched_t * appscheduler,
      appsched_attr_t * attr,
      appschedlib_policy_t policy,
      int prio);
```

Descripción

La función *appschedlib_init_schedpolicy()* inicializa la política de planificación especificada en *policy*, con los atributos definidos en el objeto de atributos *attr*. La prioridad *prio* de la política de planificación será la prioridad del sistema de los threads planificados por *appscheduler*. Los valores soportados de *policy* dependerán de la implementación, y deberán estar definidos en <*appschedlib.h*>. Si *attr* es NULL, los valores por defecto de los atributos deberán ser utilizados. Si posterior al llamado de la función, los valores de *attr* son modificados, no afectará a los valores de los atributos de la política de planificación. Si se completa con éxito, la función *appschedlib_init_schedpolicy()* almacenará en el lugar al que hace referencia *appscheduler*, el valor del identificador del PDU creado.

Valor de retorno

Si se completa exitosamente, la función *appschedlib_init_schedpolicy()* el valor de 0.

Errores

La función *appschedlib_init_schedpolicy()* fallará si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado, o los valores especificado por *policy* o *prio* no son válidos.

[EAGAIN] El sistema no cuenta con recursos suficientes para crear un nuevo PDU, o el número de planificadores ha alcanzado el límite impuesto por el sistema.

[ENOTSUP] La política especificada por *policy* no está soportada.

appschedlib_get_schedpolicy

Nombre

appschedlib_init_schedpolicy - Obtiene los parámetros de la política de planificación.

Sinopsis

```
#include <appschedlib.h>

int appschedlib_get_schedpolicy(
      const appschedlib_sched_t * appscheduler,
      appschedlib_policy_t * policy,
      int * prio);
```

Descripción

La función *appschedlib_get_schedpolicy()* obtiene los valores de la política de planificación *policy* y el nivel de prioridad *prio* del PDU al que hace referencia *appscheduler.*

Valor de retorno

Si se completa exitosamente, la función *appschedlib_get_schedpolicy()* el valor de 0.

Errores

La función *appschedlib_get_schedpolicy()* fallará si:

[ESRCH] El valor especificado por *appscheduler* no hace referencia a un identificador de alguna política de planificación previamente inicializada.

[EAGAIN] El sistema no cuenta con recursos suficientes para crear un nuevo PDU, o el número de planificadores ha alcanzado el límite impuesto por el sistema.

[ENOTSUP] La política especificada por *policy* no está soportada.

appschedlib_set_schedprio

Nombre

appschedlib_set_schedprio - Establece la prioridad de la política de planificación.

Sinopsis

```
#include <appschedlib.h>

int appschedlib_set_schedprio(
          appschedlib_sched_t * appscheduler,
          int * prio);
```

Descripción

La función *appschedlib_set_schedprio()* establece el valor del nivel de prioridad *prio* del PDU al que hace referencia por *appscheduler.*

Valor de retorno

Si se completa exitosamente, la función *appschedlib_set_schedprio()* el valor de 0.

Errores

La función *appschedlib_set_schedprio()* fallará si:

[EINVAL] El valor especificado por *prio* no es válido.

[ESRCH] El valor especificado por *appscheduler* no hace referencia a un identificador de alguna política de planificación previamente inicializada.

appschedlib_set_synchstart

Nombre

appschedlib_set_synchstart - Establece el valor de inicio síncrono.

Sinopsis

```
#include <appschedlib.h>

int appschedlib_set_synchstart(
          appschedlib_sched_t * appscheduler,
          int n);
```

Descripción

La función *appschedlib_set_synchstart()* establece el valor de inicio síncrono del planificador *appscheduler*. El uso de la función provocará que los primeros *n* threads inicien su ejecución de manera simultánea, a partir del punto de sincronía. En el código del thread, el punto de sincronía se establece llamando a la función *posix_appsched_invoke_scheduler()*.

Valor de retorno

Si se completa exitosamente, la función *appschedlib_set_synchstart()* retornará el valor de 0.

Errores

La función *appschedlib_set_synchstart()* fallará si:

[EINVAL] El valor especificado por *n* no es válido.

[ESRCH] El valor especificado por *appscheduler* no hace referencia a un identificador de alguna política de planificación previamente inicializada.

appschedlib_get_synchstart

Nombre

appschedlib_get_synchstart - Obtiene el valor de inicio síncrono.

Sinopsis

```
#include <appschedlib.h>

int appschedlib_get_synchstart(
          appschedlib_sched_t * appscheduler,
          int * n);
```

Descripción

La función *appschedlib_get_synchstart()* obtiene el valor de inicio síncrono del planificador *appscheduler*. El uso de la función provocará que *n* contenga el número de threads inicien su ejecución de manera simultánea, a partir del punto de sincronía.

Valor de retorno

Si se completa exitosamente, la función *appschedlib_get_synchstart()* retornará el valor de 0.

Errores

La función *appschedlib_get_synchstart()* fallará si:

[ESRCH] El valor especificado por *appscheduler* no hace referencia a un identificador de alguna política de planificación previamente inicializada.

appschedlib_set_schedhandlers

Nombre

appschedlib_set_schedhandlers - Establece los punteros a las funciones manejadores de tolerancia a fallos.

Sinopsis

```
#include <appschedlib.h>

int appschedlib_set_schedhandlers(
        appschedlib_sched_t * appscheduler,
        void (*overrun_handler)
                (pthread_t *thread),
        void (*deadline_miss_handler)
                (pthread_t *thread);
```

Descripción

La función *appschedlib_set_schedhandlers()* establece los punteros a las funciones manejadoras de cómputo en exceso y fallos en plazo del planificador identificado por *appscheduler*. Las funciones manejadores serán ejecutadas si alguno de los threads planificados por el PDU experimenta alguno de los fallos descritos.

Valor de retorno

Si se completa exitosamente, la función *appschedlib_set_schedhandlers()* retornará el valor de 0.

Errores

La función *appschedlib_set_schedhandlers()* fallará si:

[EINVAL] Los valores especificado por *void (*overrun_handler) (pthread_t *thread)* y *void (*deadline_miss_handler) (pthread_t *thread)* no son válidos.

[ESRCH] El valor especificado por *appscheduler* no hace referencia a un identificador de alguna política de planificación previamente inicializada.

appschedlib_get_schedhandlers

Nombre

appschedlib_get_schedhandlers - Obtiene los punteros a las funciones manejadores de tolerancia a fallos.

Sinopsis

```
#include <appschedlib.h>

int appschedlib_get_schedhandlers(
      const appschedlib_sched_t * appscheduler,
      void (*overrun_handler)
              (pthread_t *thread),
      void (*deadline_miss_handler)
              (pthread_t *thread);
```

Descripción

La función *appschedlib_get_schedhandlers()* obtiene los valores los punteros a las funciones manejadoras de cómputo en exceso y fallos en plazo del planificador identificado por *appscheduler*.

Valor de retorno

Si se completa exitosamente, la función *appschedlib_get_schedhandlers()* retornará el valor de 0.

Errores

La función *appschedlib_get_schedhandlers()* fallará si:

[ESRCH] El valor especificado por *appscheduler* no hace referencia a un identificador de alguna política de planificación previamente inicializada.

appschedlib_set_server

Nombre

appschedlib_set_server - Establece un servidor.

Sinopsis

```
#include <appschedlib.h>

int appschedlib_set_server(
      const appschedlib_sched_t * appscheduler,
      appschedlib_server_t * server,
      struct timespec * period
      struct timespec * budget);
```

Descripción

La función *appschedlib_set_server()* establece en el planificador *appscheduler*, el servidor *server*, así como su de periodo *period* y crédito *budget*. El servidor será planificado con la política de planificación definida en el PDU. Puede definirse más de un servidor por PDU. Si se completa con éxito, la función *appschedlib_set_sever()* almacenará en el lugar al que hace referencia por *server*, el valor del identificador del servidor.

Valor de retorno

Si se completa exitosamente, la función *appschedlib_set_server()* retornará el valor de 0.

Errores

La función *appschedlib_set_server()* fallará si:

[EINVAL] El valor especificado por *server* no hace referencia a un objeto identificador de servidor válido, o los valores especificado por *period* o *budget* no son válidos.

[ESRCH] El valor especificado por *appscheduler* no hace referencia a un identificador de alguna política de planificación previamente inicializada.

appschedlib_get_server

Nombre

appschedlib_get_server - Obtiene los valores de un servidor.

Sinopsis

```
#include <appschedlib.h>

int appschedlib_get_server(
      const appschedlib_sched_t * appscheduler,
      const appschedlib_server_t * server,
      struct timespec * period
      struct timespec * budget);
```

Descripción

La función *appschedlib_get_server()* obtiene del planificador *appscheduler* y del servidor *server*, los valores del periodo *period* y del crédito *budget*.

Valor de retorno

Si se completa exitosamente, la función *appschedlib_get_server* retornará el valor de 0.

Errores

La función *appschedlib_get_server()* fallará si:

[ESRCH] El valor especificado por *appscheduler* no hace referencia a un identificador de alguna política de planificación previamente inicializada.

posix_appsched_scheduler_create

Nombre

posix_appsched_scheduler_create - Crea un PDU.

Sinopsis

```
#include <sched.h>

int posix_appsched_scheduler_create (
     const posix_appsched_scheduler_ops_t
                              * scheduler_ops,
     size_t scheduler_data_size,
     void * arg, size_t arg_size,
     posix_appsched_scheduler_id_t
                              * scheduler_id);
```

Descripción

La función *posix_appsched_scheduler_create()* crea un planificador definido por el usuario. El objeto *scheduler_ops* hace referencia a la estructura que contiene los punteros a las operaciones primitivas del PDU. El valor contenido en *scheduler_data_size* se utiliza para que el PDU solicite al sistema una área de memoria del tamaño indicado por este parámetro. El PDU puede recibir un conjunto de parámetros al momento de ser creado, almacenados en el área de memoria de tamaño *arg_size*, a la que hace referencia *arg*. Si la función *posix_appsched_scheduler_create()* se completa con éxito, almacenará en el lugar al que hace referencia *scheduler_id*, el valor del identificador del PDU.

Valor de retorno

Si se completa exitosamente, la función *posix_appsched_scheduler_create()* retornará el valor de 0.

Errores

La función *posix_appsched_scheduler_create()* fallará si:

[EINVAL] Los valores especificados por *scheduler_ops*, *arg* o *scheduler_id* hacen referencia a objetos no válidos, o los valores especificados por *scheduler_data_size* o *arg_size* no son válidos.

posix_appsched_actions_addaccept

Nombre

posix_appsched_actions_addaccept - Agrega la acción de aceptación de thread.

Sinopsis

```
#include <sched.h>

int posix_appsched_actions_addaccept (
     posix_appsched_actions_t * sched_actions,
     pthread_t thread);
```

Descripción

La función *posix_appsched_actions_addaccept()* agrega la acción de aceptación de un thread *planificado-por-aplicación*, al objeto de acciones al que hace referencia *sched_actions*, y que servirá para notificar que el thread identificado por *thread* ha sido aceptado por el PDU.

Valor de retorno

Si se completa exitosamente, la función *posix_appsched_actions_addaccept()* retornará el valor de 0.

Errores

La función *posix_appsched_actions_addaccept()* fallará si:

[EINVAL] El valor especificados por *sched_actions* hace referencia a un objeto de acciones no válido, o el valor de *thread* no es válido.

[ENOMEM] No existe memoria suficiente para agregar la acción al objeto de acciones.

posix_appsched_actions_addreject

Nombre

posix_appsched_actions_addreject - Agrega la acción de rechazo de thread.

Sinopsis

```
#include <sched.h>

int posix_appsched_actions_addreject (
     posix_appsched_actions_t * sched_actions,
     pthread_t thread);
```

Descripción

La función *posix_appsched_actions_addreject()* agrega la acción de rechazo de un thread *planificado-por-aplicación*, al objeto de acciones al que hace referencia *sched_actions*, y que servirá para notificar que el thread identificado por *thread* ha sido rechazado por el PDU.

Valor de retorno

Si se completa exitosamente, la función *posix_appsched_actions_addreject* retornará el valor de 0.

Errores

La función *posix_appsched_actions_addreject()* fallará si:

[EINVAL] El valor especificados por *sched_actions* hace referencia a un objeto de acciones no válido, o el valor de *thread* no es válido.

[ENOMEM] No existe memoria suficiente para agregar la acción al objeto de acciones.

posix_appsched_actions_addactivate

Nombre

posix_appsched_actions_addactivate - Agrega la acción de activación de thread.

Sinopsis

```
#include <sched.h>

int posix_appsched_actions_addactivate (
     posix_appsched_actions_t * sched_actions,
     pthread_t thread,
     posix_appsched_urgency_t urgency);
```

Descripción

La función *posix_appsched_actions_addactivate()* agrega la acción de activación de un thread *planificado-por-aplicación*, al objeto de acciones al que hace referencia *sched_actions*, y que servirá para notificar que el thread identificado por *thread* ha sido activado por su PDU, con el valor de urgencia especificado por *urgency*.

Valor de retorno

Si se completa exitosamente, la función *posix_appsched_actions_addactivate* retornará el valor de 0.

Errores

La función *posix_appsched_actions_addactivate()* fallará si:

[EINVAL] El valor especificados por *sched_actions* hace referencia a un objeto de acciones no válido, o el valor de *thread* no es válido.

[ENOMEM] No existe memoria suficiente para agregar la acción al objeto de acciones.

posix_appsched_actions_addtimedactivation

Nombre

posix_appsched_actions_addtimedactivation - Agrega la acción de activación programada de thread.

Sinopsis

```
#include <sched.h>

int posix_appsched_actions_addtimedactivation (
     posix_appsched_actions_t * sched_actions,
     pthread_t thread,
     posix_appsched_urgency_t urgency,
     const struct timespec * at_time);
```

Descripción

La función *posix_appsched_actions_addtimedactivation()* agrega la acción de activación de un thread *planificado-por-aplicación*, al objeto de acciones al que hace referencia *sched_actions*, y que servirá para notificar que el thread identificado por *thread* ha sido activado por su PDU, con el valor de urgencia especificado por *urgency*. La activación deberá hacerse en el instante de tiempo especificado por *at_time*.

Valor de retorno

Si se completa exitosamente, la función *posix_appsched_actions_addtimedactivation()* retornará el valor de 0.

Errores

La función *posix_appsched_actions_addtimedactivation()* fallará si:

[EINVAL] El valor especificados por *sched_actions* hace referencia a un objeto de acciones no válido, o el valor de *thread* no es válido, o el valor especificado por *at_time* no es válido.

[ENOMEM] No existe memoria suficiente para agregar la acción al objeto de acciones.

posix_appsched_actions_addsuspend

Nombre

posix_appsched_actions_addsuspend - Agrega la acción de suspensión de thread.

Sinopsis

```
#include <sched.h>

int posix_appsched_actions_addsuspend (
     posix_appsched_actions_t * sched_actions,
     pthread_t thread);
```

Descripción

La función *posix_appsched_actions_addsuspend()* agrega la acción de suspensión de un thread *planificado-por-aplicación*, al objeto de acciones al que hace referencia *sched_actions*, y que servirá para notificar que el thread identificado por *thread* ha sido suspendido por su PDU.

Valor de retorno

Si se completa exitosamente, la función *posix_appsched_actions_addsuspend* retornará el valor de 0.

Errores

La función *posix_appsched_actions_addsuspend()* fallará si:

[EINVAL] El valor especificados por *sched_actions* hace referencia a un objeto de acciones no válido, o el valor de *thread* no es válido.

[ENOMEM] No existe memoria suficiente para agregar la acción al objeto de acciones.

posix_appsched_actions_addtimeout

Nombre

posix_appsched_actions_addtimeout - Agrega la acción de temporización.

Sinopsis

```
#include <sched.h>

int posix_appsched_actions_addtimeout (
     posix_appsched_actions_t * sched_actions,
     posix_appsched_scheduler_id_t
                             * scheduler_id,
     clock_id clock_id,
     int flags,
     const struct timespec * at_time);
```

Descripción

La función *posix_appsched_actions_addtimeout()* agrega la acción de temporización al objeto de acciones al que hace referencia *sched_actions*. Si la bandera TIMER_ABSTIME no está establecida en el argumento *flags*, la función suspenderá al PDU identificado por *scheduler_id* hasta que el intervalo de tiempo especificado en *at_time* haya transcurrido. El reloj utilizado para medir el tiempo es el especificado por *clock_id*.

Si la bandera TIMER_ABSTIME está establecida en el argumento *flags*, la función suspenderá al PDU identificado por *scheduler_id* hasta que el reloj especificado por *clock_id* haya alcanzado el tiempo absoluto especificado por *at_time*.

Valor de retorno

Si se completa exitosamente, la función *posix_appsched_actions_addtimeout()* retornará el valor de 0.

Errores

La función *posix_appsched_actions_addtimeout()* fallará si:

[EINVAL] El valor especificados por *sched_actions* hace referencia a un objeto de acciones no válido, o el identificador *scheduler_id* no es válido, o los argumentos *flags*, *clock_id* o *at_tim*e no son válidos.

[ENOMEM] No existe memoria suficiente para agregar la acción al objeto de acciones.

[ENOTSUP] El reloj especificado por *clock_id* no está soportado.

posix_appsched_actions_addthreadnotification

Nombre

posix_appsched_actions_addthreadnotification - Agrega la acción de notificación para thread.

Sinopsis

```
#include <sched.h>

int posix_appsched_actions_addthreadnotification (
     posix_appsched_actions_t * sched_actions,
     pthread_t thread,
     const struct timespec * at_time);
```

Descripción

La función *posix_appsched_actions_addthreadnotification()* agrega la acción de notificación para thread *planificado-por-aplicación*, al objeto de acciones al que hace referencia *sched_actions*, y que servirá para notificar que el thread identificado por *thread* debe recibir una notificación en el instante de tiempo especificado por *at_time*.

Valor de retorno

Si se completa exitosamente, la función *posix_appsched_actions_addthreadnotification()* retornará el valor de 0.

Errores

La función *posix_appsched_actions_addthreadnotification()* fallará si:

[EINVAL] El valor especificado por *sched_actions* hace referencia a un objeto de acciones no válido, o el valor de *thread* no es válido, o el valor especificado por *at_time* no es válido.

[ENOMEM] No existe memoria suficiente para agregar la acción al objeto de acciones.

posix_appsched_actions_addlockmutex

Nombre

posix_appsched_actions_addlockmutex - Agrega la acción de asignación de monitor.

Sinopsis

```
#include <sched.h>

int posix_appsched_actions_addlockmutex (
    posix_appsched_actions_t * sched_actions,
    pthread_t thread,
    pthread_mutex_t * mutex);
```

Descripción

La función *posix_appsched_actions_addlockmutex()* agrega la acción de asignación de monitor a un thread *planificado-por-aplicación*, al objeto de acciones al que hace referencia *sched_actions*, y que servirá para notificar que al thread identificado por *thread* se le ha asignado el monitor especificado por *mutex*.

Valor de retorno

La función *posix_appsched_actions_addlockmutex()*, si se completa exitosamente, retornará el valor de 0.

Errores

La función *posix_appsched_actions_addlockmutex()* fallará si:

[EINVAL] El valor especificados por *sched_actions* hace referencia a un objeto de acciones no válido, o el valor de *thread* no es válido, o el monitor identificado por *mutex* no es válido.

[ENOMEM] No existe memoria suficiente para agregar la acción al objeto de acciones.

posix_appsched_invoke_scheduler

Nombre

posix_appsched_invoke_scheduler - Agrega la acción de invocación explícita del PDU.

Sinopsis

```
#include <sched.h>

int posix_appsched_invoke_scheduler (
       int user_event_code);
```

Descripción

La función *posix_appsched_invoke_scheduler()* invoca explícitamente al PDU que planifica al thread que hace la llamada a la función. Si se completa exitosamente, la función *posix_appsched_invoke_scheduler()* provocará que se ejecute la operación primitiva *explicit_call()* del PDU correspondiente. El parámetro *thread* de la operación primitiva *explicit_call()* será igual al identificador del thread que haya invocado explícitamente al PDU, y el parámetro *user_event_code* será igual al parámetro *user_event_code* de la función *posix_appsched_invoke_scheduler()*.

Valor de retorno

Si se completa exitosamente, la función *posix_appsched_invoke_scheduler()* retornará el valor de 0.

Errores

La función *posix_appsched_invoke_scheduler()* fallará si:

[EPOLICY] El thread que llama a la función no es un thread *planificado-por-aplicación*.

posix_appsched_invoke_withdata

Nombre

posix_appsched_invoke_withdata - Agrega la acción de invocación explícita con datos.

Sinopsis

```
#include <sched.h>

int posix_appsched_invoke_withdata (
     const void * msg, size_t msg_size,
     void * reply, size_t reply_size);
```

Descripción

La función *posix_appsched_invoke_withdata()* invoca explícitamente al PDU que planifica al thread que hace la llamada a la función. Si se completa exitosamente, la función *posix_appsched_invoke_withdata()* provocará que se ejecute la operación primitiva *explicit_call_withdata()* del PDU correspondiente. El parámetro *thread* de la operación primitiva *explicit_call_withdata()* será igual al identificador del thread que haya invocado explícitamente al PDU. Si el valor de *msg_size* es mayor que cero, la función hará disponible para el planificador una área de memoria cuyo contenido es idéntico al área de memoria referida por *msg* y de tamaño *msg_size*. Cuando el planificador es notificado de este evento y si el valor de *reply* es diferente a NULL, podrá responder al thread copiando en el área de memoria referida por *reply* un mensaje de tamaño *reply_size*.

Valor de retorno

Si se completa exitosamente, la función *posix_appsched_invoke_withdata()* retornará el valor de 0.

Errores

La función *posix_appsched_invoke_withdata()* fallará si:

[EPOLICY] El thread que llama a la función no es un thread *planificado-por-aplicación*.

[EINVAL] El valor especificados por *msg_size* o *msg* no son válidos.

posix_appschedattr_setwaitsignalset

Nombre

posix_appschedattr_setwaitsignalset - Establece la máscara de señales del PDU.

Sinopsis

```
#include <sched.h>

int posix_appschedattr_setwaitsignalset(
        posix_appsched_scheduler_id_t
                              * scheduler_id,
        const sigset_t *set);
```

Descripción

La función *posix_appschedattr_setwaitsignalset()* establece la máscara de señales del PDU identificado por *scheduler_id*.

Valor de retorno

Si se completa exitosamente, la función *posix_appschedattr_setwaitsignalset()* retornará el valor de 0.

Errores

La función *posix_appschedattr_setwaitsignalset()* fallará si:

[EINVAL] El PDU identificado por *scheduler_id* no existe, o el argumento *set* no es válido.

posix_appschedattr_getwaitsignalset

Nombre

posix_appschedattr_getwaitsignalset - Obtiene la máscara de señales del PDU.

Sinopsis

```
#include <sched.h>

int posix_appschedattr_getwaitsignalset(
        const posix_appsched_scheduler_id_t
                              * scheduler_id,
        sigset_t * set);
```

Descripción

La función *posix_appschedattr_getwaitsignalset()* obtiene la máscara de señales del PDU identificado por *scheduler_id*.

Valor de retorno

Si se completa exitosamente, la función *posix_appschedattr_getwaitsignalset()* retornará el valor de 0.

Errores

La función *posix_appschedattr_getwaitsignalset()* fallará si:

[EINVAL] El PDU identificado por *scheduler_id* no existe.

posix_appsched_geturgency

Nombre

posix_appsched_geturgency - Obtiene el valor de urgencia.

Sinopsis

```
#include <sched.h>

int posix_appsched_geturgency(
      pthread_t thread,
      posix_appsched_urgency_t * urgency);
```

Descripción

La función *posix_appsched_geturgency()* obtiene el valor de urgencia del thread identificado por t*hread*.

Valor de retorno

Si se completa exitosamente, la función *posix_appsched_geturgency()* retornará el valor de 0.

Errores

La función *posix_appsched_geturgency()* fallará si:

[EINVAL] El thread identificado por *thread* no existe.

pthread_attr_setappscheduler

Nombre

pthread_attr_setappscheduler - Establece el atributo que especifica el PDU de un thread *planificado-por-aplicación.*

Sinopsis

```
#include <pthread.h>

int pthread_attr_setappscheduler(
     pthread_attr_t * attr,
     posix_appsched_scheduler_id_t scheduler);
```

Descripción

La función *pthread_attr_setappscheduler()* establece el valor del atributo *scheduler* del objeto de atributos *attr*. Si el atributo *scheduler* es diferente de NULL, significa que el thread es *planificado-por-aplicación*, y que será planificado por el PDU identificado por *scheduler*. El atributo *scheduler* hace referencia a un PDU creado utilizando la función *posix_appsched_scheduler_create()*. Después de que han sido establecidos, un thread puede ser creado con los atributos especificados en *attr* utilizando la función *pthread_create().*

Valor de retorno

La función *pthread_attr_setappscheduler()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_attr_setappscheduler()* fallará si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado, o el valor especificado por *scheduler* no es válido.

pthread_attr_getappscheduler

Nombre

pthread_attr_getappscheduler - Obtiene el atributo que especifica el PDU de un thread *planificado-por-aplicación.*

Sinopsis

```
#include <pthread.h>

int pthread_attr_getappscheduler(
     pthread_attr_t * attr,
     posix_appsched_scheduler_id_t *scheduler);
```

Descripción

La función *pthread_attr_getappscheduler* obtiene el valor del atributo *scheduler* del objeto de atributos *attr*. Si el atributo *scheduler* es diferente de NULL, significa que el thread es *planificado-por-aplicación*, y que será planificado por el PDU identificado por *scheduler*. El atributo *scheduler* hace referencia a un PDU creado utilizando la función *posix_appsched_scheduler_create()*.

Valor de retorno

La función *pthread_attr_getappscheduler()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_attr_getappscheduler()* fallará si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado.

pthread_attr_setappschedparam

Nombre

pthread_attr_setappschedparam - Establece los atributos que especifican los parámetros de planificación de un thread *planificado-por-aplicación.*

Sinopsis

```
#include <pthread.h>

int pthread_attr_setappschedparam(
     pthread_attr_t * attr,
     const void * param,
     size_t paramsize);
```

Descripción

La función *pthread_attr_setappschedparam()* establece el valor del atributo *appsched_param* del objeto de atributos *attr*. Para un thread *planificado-por-aplicación*, el atributo *appsched_param* representa sus atributos de planificación definida por el usuario. Después de que han sido establecidos, un thread puede ser creado con los atributos especificados en *attr* utilizando la función *pthread_create()*.

Valor de retorno

La función *pthread_attr_setappschedparam()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_attr_setappschedparam()* fallará si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado, o los valores especificados por *param* o *paramsize* no son válidos.

pthread_attr_getappschedparam

Nombre

pthread_attr_getappschedparam - Obtiene los atributos que especifican los parámetros de planificación de un thread *planificado-por-aplicación.*

Sinopsis

```
#include <pthread.h>

int pthread_attr_getappschedparam(
     pthread_attr_t * attr,
     const void * param,
     size_t * paramsize);
```

Descripción

La función *pthread_attr_getappschedparam()* obtiene el valor del atributo *appsched_param* del objeto de atributos *attr*. Para un thread *planificado-por-aplicación*, el atributo *appsched_param* representa sus atributos de planificación definida por el usuario.

Valor de retorno

La función *pthread_attr_getappschedparam()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_attr_getappschedparam()* fallará si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado.

pthread_attr_setpreemptionlevel

Nombre

pthread_attr_setpreemptionlevel - Establece el atributo que especifica el nivel de expulsión de un thread.

Sinopsis

```
#include <pthread.h>

int pthread_attr_setpreemptionlevel(
     pthread_attr_t * attr,
     unsigned short int level);
```

Descripción

La función *pthread_attr_setpreemptionlevel()* establece el valor del atributo *level* del objeto de atributos *attr*. El atributo *level* representa el nivel de expulsión del thread. Después de que han sido establecidos, un thread puede ser creado con los atributos especificados en *attr* utilizando la función *pthread_create()*.

Valor de retorno

La función *pthread_attr_setpreemptionlevel()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_attr_setpreemptionlevel()* fallará si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado, o el valor especificado por *level* no es válido.

pthread_attr_getpreemptionlevel

Nombre

pthread_attr_getpreemptionlevel - Obtiene el atributo que especifica el nivel de expulsión de un thread.

Sinopsis

```
#include <pthread.h>

int pthread_attr_getpreemptionlevel(
     pthread_attr_t * attr,
     unsigned short int * level);
```

Descripción

La función *pthread_attr_getpreemptionlevel()* obtiene el valor del atributo *level* del objeto de atributos *attr*. El atributo *level* representa el nivel de expulsión del thread.

Valor de retorno

La función *pthread_attr_getpreemptionlevel()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_attr_getpreemptionlevel()* fallará si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado.

pthread_mutexattr_setpreemptionlevel

Nombre

pthread_mutexattr_setpreemptionlevel - Establece el atributo que especifica el nivel de expulsión de un monitor.

Sinopsis

```
#include <pthread.h>

int pthread_mutexattr_setpreemptionlevel(
     pthread_mutexattr_t * attr,
     unsigned short int level);
```

Descripción

La función *pthread_mutexattr_setpreemptionlevel()* establece el valor del atributo *level* del objeto de atributos *attr*. El atributo *level* representa el nivel de expulsión del monitor.

Valor de retorno

La función *pthread_mutexattr_setpreemptionlevel()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_mutexattr_setpreemptionlevel()* fallará si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado, o el valor especificado por *level* no es válido.

pthread_mutexattr_getpreemptionlevel

Nombre

pthread_mutexattr_getpreemptionlevel - Obtiene el atributo que especifica el nivel de expulsión de un monitor.

Sinopsis

```
#include <pthread.h>

int pthread_mutexattr_getpreemptionlevel(
     pthread_mutexattr_t * attr,
     unsigned short int * level);
```

Descripción

La función *pthread_mutexattr_getpreemptionlevel()* obtiene el valor del atributo *level* del objeto de atributos *attr*. El atributo *level* representa el nivel de expulsión del monitor.

Valor de retorno

La función *pthread_mutexattr_getpreemptionlevel()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_mutexattr_getpreemptionlevel()* fallará si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado.

pthread_attr_set_appschedlib_scheduler_np

Nombre

pthread_attr_set_appschedlib_scheduler_np - Establece el atributo que especifica el PDU de un thread *planificado-por-aplicación*.

Sinopsis

```
#include <pthread.h>

int pthread_attr_set_appschedlib_scheduler_np(
     pthread_attr_t * attr,
     appschedlib_sched_t appscheduler);
```

Descripción

La función *pthread_attr_set_appschedlib_scheduler_np()* establece el valor del atributo *appscheduler* del objeto de atributos *attr*. Si el atributo *appscheduler* es diferente de NULL, significa que el thread es *planificado-por-aplicación*, y que será planificado por el PDU identificado por *appscheduler*. El atributo *appscheduler* hace referencia a un PDU creado utilizando la función *appschedlib_init_schedpolicy()*. Después de que han sido establecidos, un thread puede ser creado con los atributos especificados en *attr* utilizando la función *pthread_create()*.

Valor de retorno

La función *pthread_attr_set_appschedlib_scheduler_np()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_attr_set_appschedlib_scheduler_np()* fallará si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado, o el valor especificado por *appscheduler* no es válido.

pthread_attr_get_appschedlib_scheduler_np

Nombre

pthread_attr_get_appschedlib_scheduler_np - Obtiene el atributo que especifica el PDU de un thread *planificado-por-aplicación*.

Sinopsis

```
#include <pthread.h>

int pthread_attr_get_appschedlib_scheduler_np(
     const pthread_attr_t * attr,
     appschedlib_sched_t * appscheduler);
```

Descripción

La función *pthread_attr_get_appschedlib_scheduler_np()* obtiene el valor del atributo *appscheduler* del objeto de atributos *attr*. El atributo *appscheduler* hace referencia al PDU que planificará al thread *planificado-por-aplicación*. El atributo *appscheduler* hace referencia a un PDU previamente creado utilizando la función *appschedlib_init_schedpolicy()*.

Valor de retorno

La función *pthread_attr_get_appschedlib_scheduler_np()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_attr_get_appschedlib_scheduler_np()* fallará si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado.

pthread_attr_set_appschedlib_server_np

Nombre

pthread_attr_set_appschedlib_server_np - Establece el atributo que especifica el servidor aperiódico de un thread *planificado-por-aplicación*.

Sinopsis

```
#include <pthread.h>

int pthread_attr_set_appschedlib_server_np(
     pthread_attr_t * attr,
     appschedlib_server_t server);
```

Descripción

La función *pthread_attr_set_appschedlib_server_np()* establece el valor del atributo *server* del objeto de atributos *attr*. El atributo *server* hace referencia al servidor aperiódico que planificará al thread *planificado-por-aplicación*. Después de que han sido establecidos, un thread puede ser creado con los atributos especificados en *attr* utilizando la función *pthread_create()*.

Valor de retorno

La función *pthread_attr_set_appschedlib_server_np()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_attr_set_appschedlib_server_np()* fallará si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado, o el valor especificado por *server* no hace referencia a un servidor previamente inicializado.

pthread_attr_get_appschedlib_server_np

Nombre

pthread_attr_get_appschedlib_server_np - Obtiene el atributo que especifica el servidor aperiódico de un thread *planificado-por-aplicación*.

Sinopsis

```
#include <pthread.h>

int pthread_attr_get_appschedlib_server_np(
     pthread_attr_t * attr,
     appschedlib_server_t * server);
```

Descripción

La función *pthread_attr_get_appschedlib_server_np()* obtiene el valor del atributo *server* del objeto de atributos *attr*. El atributo *server* hace referencia al servidor aperiódico que planificará al thread *planificado-por-aplicación*.

Valor de retorno

La función *pthread_attr_get_appschedlib_server_np()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_attr_get_appschedlib_server_np()* fallará si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado.

pthread_attr_set_appschedlib_period_np

Nombre

pthread_attr_set_appschedlib_period_np - Establece el atributo que especifica el periodo de un thread *planificado-por-aplicación*.

Sinopsis

```
#include <pthread.h>

int pthread_attr_set_appschedlib_period_np(
     pthread_attr_t * attr,
     struct timespec * period);
```

Descripción

La función *pthread_attr_set_appschedlib_period_np()* establece el valor del atributo *period* del objeto de atributos *attr*. El atributo *period* especifica el periodo del thread *planificado-por-aplicación*. Después de que han sido establecidos, un thread puede ser creado con los atributos especificados en *attr* utilizando la función *pthread_create()*.

Valor de retorno

La función *pthread_attr_set_appschedlib_period_np()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_attr_set_appschedlib_period_np()* fallará si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado, o el valor especificado por *period* no es válido.

pthread_attr_get_appschedlib_period_np

Nombre

pthread_attr_get_appschedlib_period_np -Obtiene el atributo que especifica el periodo de un thread *planificado-por-aplicación*.

Sinopsis

```
#include <pthread.h>

int pthread_attr_get_appschedlib_period_np(
     pthread_attr_t * attr,
     struct timespec * period);
```

Descripción

La función *pthread_attr_get_appschedlib_period_np()* obtiene el valor del atributo *period* del objeto de atributos *attr*. El atributo *period* especifica el periodo del thread *planificado-por-aplicación*.

Valor de retorno

La función *pthread_attr_get_appschedlib_period_np()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_attr_get_appschedlib_period_np()* fallará si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado.

pthread_attr_set_appschedlib_deadline_np

Nombre

pthread_attr_set_appschedlib_deadline_np - Establece el atributo que especifica el plazo relativo de un thread *planificado-por-aplicación*.

Sinopsis

```
#include <pthread.h>

int pthread_attr_set_appschedlib_deadline_np(
     pthread_attr_t * attr,
     struct timespec * deadline);
```

Descripción

La función *pthread_attr_set_appschedlib_deadline_np()* establece el valor del atributo *deadline* del objeto de atributos *attr*. El atributo *deadline* especifica el plazo relativo del thread *planificado-por-aplicación*. Después de que han sido establecidos, un thread puede ser creado con los atributos especificados en *attr* utilizando la función *pthread_create()*.

Valor de retorno

La función *pthread_attr_set_appschedlib_deadline_np()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_attr_set_appschedlib_deadline_np()* fallará si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado, o el valor especificado por *deadline* no es válido.

pthread_attr_get_appschedlib_deadline_np

Nombre

pthread_attr_get_appschedlib_deadline_np - Obtiene el atributo que especifica el plazo relativo de un thread *planificado-por-aplicación*.

Sinopsis

```
#include <pthread.h>

int pthread_attr_get_appschedlib_deadline_np(
     pthread_attr_t * attr,
     struct timespec * deadline);
```

Descripción

La función *pthread_attr_get_appschedlib_deadline_np()* obtiene el valor del atributo *deadline* del objeto de atributos *attr*. El atributo *deadline* especifica el plazo relativo del thread *planificado-por-aplicación*.

Valor de retorno

La función *pthread_attr_get_appschedlib_deadline_np()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_attr_get_appschedlib_deadline_np()* fallará si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado.

pthread_attr_set_appschedlib_maxexectime_np

Nombre

pthread_attr_set_appschedlib_maxexectime_np - Establece el atributo que especifica el tiempo de cómputo máximo de un thread *planificado-por-aplicación*.

Sinopsis

```
#include <pthread.h>

int pthread_attr_set_appschedlib_maxexectime_np(
     pthread_attr_t * attr,
     struct timespec * maxexectime);
```

Descripción

La función *pthread_attr_set_appschedlib_maxexectime_np()* establece el valor del atributo *maxexectime* del objeto de atributos *attr*. El atributo *maxexectime* especifica el tiempo de cómputo máximo del thread *planificado-por-aplicación*. Si el thread se ejecuta por más tiempo que *maxexectime*, y si está definida, se ejecutará la función manejadora de cómputo en exceso. Si el thread es planificador por algún servidor, y si esta definida, entonces se ejecuta la operación del planificador *thread_block()*. Después de que han sido establecidos, un thread puede ser creado con los atributos especificados en *attr* utilizando la función *pthread_create()*.

Valor de retorno

La función *pthread_attr_set_appschedlib_maxexectime_np()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_attr_set_appschedlib_maxexectime_np()* fallará si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado, o el valor especificado por *maxexectime* no es válido.

pthread_attr_get_appschedlib_maxexectime_np

Nombre

pthread_attr_get_appschedlib_maxexectime_np - Obtiene el atributo que especifica el tiempo de cómputo máximo de un thread *planificado-por-aplicación*.

Sinopsis

```
#include <pthread.h>

int pthread_attr_get_appschedlib_maxexectime_np(
     pthread_attr_t * attr,
     struct timespec * maxexectime);
```

Descripción

La función *pthread_attr_get_appschedlib_maxexectime_np()* obtiene el valor del atributo *maxexectime* del objeto de atributos *attr*. El atributo *maxexectime* especifica el tiempo de cómputo máximo del thread *planificado-por-aplicación*. Si el thread se ejecuta por más tiempo que *maxexectime*, y si está definida, se ejecutará la función manejadora de cómputo en exceso. Si el thread es planificador por algún servidor, y si esta definida, entonces se ejecuta la operación del planificador *thread_block()*.

Valor de retorno

La función *pthread_attr_get_appschedlib_maxexectime_np()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_attr_get_appschedlib_maxexectime_np()* fallará si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado, o el valor especificado por *maxexectime* no es válido.

pthread_attr_set_appschedlib_schedhandlers_np

Nombre

pthread_attr_set_appschedlib_schedhandlers_np - Establece los atributos que especifican los manejadores de cómputo en exceso y fallos en plazo de un thread *planificado-por-aplicación*.

Sinopsis

```
#include <pthread.h>

int pthread_attr_set_appschedlib_schedhandlers_np(
     pthread_attr_t * attr,
     void (*overrun_handler)
             (pthread_t *thread),
     void (*deadline_miss_handler)
              pthread_t *thread);
```

Descripción

La función *pthread_attr_set_appschedlib_schedhandlers_np()* establece los valores de los atributos *overrun_handler()* y *deadline_miss_handler()* del objeto de atributos *attr*. Los valores de los atributos *overrun_handler()* y *deadline_miss_handler()* hacen referencia a los punteros a las funciones manejadoras de cómputo en exceso y fallo en plazo, respectivamente, del thread *planificado-por-aplicación*. Después de que han sido establecidos, un thread puede ser creado con los atributos especificados en *attr* utilizando la función *pthread_create()*.

Valor de retorno

La función *pthread_attr_set_appschedlib_schedhandlers_np()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_attr_set_appschedlib_schedhandlers_np()* fallará si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado, o los valores especificado por *void (*overrun_handler) (pthread_t *thread)* y *void (*deadline_miss_handler) (pthread_t *thread)* no son válidos.

pthread_attr_get_appschedlib_schedhandlers_np

Nombre

pthread_attr_get_appschedlib_schedhandlers_np - Obtiene los atributos que especifican los manejadores de cómputo en exceso y fallos en plazo de un thread *planificado-por-aplicación*.

Sinopsis

```
#include <pthread.h>

int pthread_attr_get_appschedlib_schedhandlers_np(
     pthread_attr_t * attr,
     void (*overrun_handler)
            (pthread_t *thread),
     void (*deadline_miss_handler)
             pthread_t *thread);
```

Descripción

La función *pthread_attr_get_appschedlib_schedhandlers_np()* establece los valores de los atributos *overrun_handler()* y *deadline_miss_handler()* del objeto de atributos *attr*. Los valores de los atributos *overrun_handler()* y *deadline_miss_handler()* hacen referencia a los punteros a las funciones manejadoras de cómputo en exceso y fallo en plazo, respectivamente, del thread *planificado-por-aplicación*.

Valor de retorno

La función *pthread_attr_get_appschedlib_schedhandlers_np()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_attr_get_appschedlib_schedhandlers_np()* fallará si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado.

pthread_setspecific_for

Nombre

pthread_setspecific_for - Establece el valor especifico asociado a un thread.

Sinopsis

```
#include <pthread.h>

int pthread_setspecific_for(
          pthread_key_t key,
          pthread_t thread,
          const void * value);
```

Descripción

La función *pthread_setspecific_for()* establece un valor específico *value* del thread identificado por *thread*, que a su vez esta asociado con el valor de la llave *key* obtenida con una llamada previa a la función *pthread_key_create()*. La función *pthread_setspecific_for()* permite el intercambio de información entre un thread y su PDU.

Valor de retorno

La función *pthread_setspecific_for()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_setspecific_for()* fallará si:

[EINVAL] El valor especificado por *key* no es válido.

[ESRCH] El valor especificado por *thread* no hace referencia a un thread existente.

[ENOMEM] No existe memoria suficiente para asociar el valor no nulo de *value* con la llave *key*.

pthread_getspecific_for

Nombre

pthread_getspecific_for - Obtiene el valor especifico asociado a un thread.

Sinopsis

```
#include <pthread.h>

int pthread_getspecific_for(
          pthread_key_t key,
          pthread_t thread,
          const void ** value);
```

Descripción

La función *pthread_getspecific_for()* obtiene un valor específico *value* del thread identificado por *thread*, que a su vez esta asociado con el valor de la llave *key* obtenida con una llamada previa a la función *pthread_key_create()*. La función *pthread_getspecific_for()* permite el intercambio de información entre un thread y su PDU.

Valor de retorno

La función *pthread_getspecific_for()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_getspecific_for()* fallará si:

[EINVAL] El valor especificado por *key* no es válido.

[ESRCH] El valor especificado por *thread* no hace referencia a un thread existente.

pthread_setappscheduler

Nombre

pthread_setappscheduler - Establece el PDU de un thread *planificado-por-aplicación*.

Sinopsis

```
#include <pthread.h>

int pthread_setappscheduler(
     pthread_t thread,
     posix_appsched_scheduler_id_t scheduler);
```

Descripción

La función *pthread_setappscheduler()* establece el PDU que planifica al thread identificado por *thread*. El identificador *scheduler* hace referencia a un PDU creado utilizando la función *posix_appsched_scheduler_create()*.

Valor de retorno

La función *pthread_setappscheduler()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_setappscheduler()* fallará si:

[EINVAL] El valor especificado por *scheduler* no es válido.

[ESRCH] El valor especificado por *thread* no hace referencia a un thread existente.

pthread_getappscheduler

Nombre

pthread_getappscheduler - Obtiene el PDU de un thread *planificado-por-aplicación.*

Sinopsis

```
#include <pthread.h>

int pthread_getappscheduler(
     pthread_t thread,
     posix_appsched_scheduler_id_t * scheduler);
```

Descripción

La función *pthread_getappscheduler* obtiene el PDU que planifica al thread identificado por *thread*. El atributo *scheduler* hace referencia a un PDU creado utilizando la función *posix_appsched_scheduler_create().*

Valor de retorno

La función *pthread_getappscheduler()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_getappscheduler()* fallará si:

[ESRCH] El valor especificado por *thread* no hace referencia a un thread existente.

pthread_setappschedparam

Nombre

pthread_setappschedparam - Establece los parámetros de planificación de un thread *planificado-por-aplicación*.

Sinopsis

```
#include <pthread.h>

int pthread_setappschedparam(
     pthread_t thread,
     const void * param,
     size_t paramsize);
```

Descripción

La función *pthread_setappschedparam()* establece los atributos de planificación definida por el usuario del thread identificado por *thread*.

Valor de retorno

La función *pthread_setappschedparam()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_setappschedparam()* fallará si:

[EINVAL] Los valores especificados por *param* o *paramsize* no son válidos.

[ESRCH] El valor especificado por *thread* no hace referencia a un thread existente.

pthread_getappschedparam

Nombre

pthread_getappschedparam - Obtiene los parámetros de planificación de un thread *planificado-por-aplicación*.

Sinopsis

```
#include <pthread.h>

int pthread_getappschedparam(
     pthread_t thread,
     const void * param,
     size_t * paramsize);
```

Descripción

La función *pthread_getappschedparam()* obtiene los atributos de planificación definida por el usuario del thread identificado por *thread*.

Valor de retorno

La función *pthread_getappschedparam()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_getappschedparam()* fallará si:

[EINVAL] El valor especificado por *attr* no hace referencia a un objeto de atributos previamente inicializado.

pthread_set_appschedlib_scheduler_np

Nombre

pthread_set_appschedlib_scheduler_np - Establece el PDU de un thread *planificado-por-aplicación*.

Sinopsis

```
#include <pthread.h>

int pthread_set_appschedlib_scheduler_np(
     pthread_t thread,
     appschedlib_sched_t appscheduler);
```

Descripción

La función *pthread_set_appschedlib_scheduler_np()* establece el valor de *appscheduler* de un thread. El valor de *appscheduler* hace referencia al PDU que planificará al thread.

Valor de retorno

La función *pthread_set_appschedlib_scheduler_np()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_set_appschedlib_scheduler_np()* fallará si:

[EINVAL] El valor especificado por *appscheduler* no es válido.

[ESRCH] El valor especificado por *thread* no hace referencia a un thread existente.

pthread_get_appschedlib_scheduler_np

Nombre

pthread_get_appschedlib_scheduler_np - Obtiene el PDU de un thread *planificado-por-aplicación*.

Sinopsis

```
#include <pthread.h>

int pthread_get_appschedlib_scheduler_np(
     const pthread_t thread,
     appschedlib_sched_t * appscheduler);
```

Descripción

La función *pthread_get_appschedlib_scheduler_np()* obtiene el valor de *appscheduler* del thread. El valor de *appscheduler* hace referencia al PDU que planificará al thread *planificado-por-aplicación*.

Valor de retorno

La función *pthread_get_appschedlib_scheduler_np()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_get_appschedlib_scheduler_np()* fallará si:

[ESRCH] El valor especificado por *thread* no hace referencia a un thread existente.

pthread_set_appschedlib_server_np

Nombre

pthread_set_appschedlib_server_np - Establece el servidor aperiódico de un thread *planificado-por-aplicación*.

Sinopsis

```
#include <pthread.h>

int pthread_set_appschedlib_server_np(
     pthread_t thread,
     appschedlib_server_t server);
```

Descripción

La función *pthread_set_appschedlib_server_np()* establece el valor de *server* del thread. El valor de *server* hace referencia al servidor aperiódico que planificará al thread *planificado-por-aplicación*.

Valor de retorno

La función *pthread_set_appschedlib_server_np()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_set_appschedlib_server_np()* fallará si:

[EINVAL] El valor especificado por *server* no es válido

[ESRCH] El valor especificado por *thread* no hace referencia a un thread existente.

pthread_get_appschedlib_server_np

Nombre

pthread_get_appschedlib_server_np - Obtiene el servidor aperiódico de un thread *planificado-por-aplicación*.

Sinopsis

```
#include <pthread.h>

int pthread_get_appschedlib_server_np(
     pthread_t thread,
     appschedlib_server_t * server);
```

Descripción

La función *pthread_get_appschedlib_server_np()* obtiene el valor de *server* del thread. El valor de *server* hace referencia al servidor aperiódico que planificará al thread *planificado-por-aplicación*.

Valor de retorno

La función *pthread_get_appschedlib_server_np()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_get_appschedlib_server_np()* fallará si:

[ESRCH] El valor especificado por *thread* no hace referencia a un thread existente.

pthread_set_appschedlib_period_np

Nombre

pthread_set_appschedlib_period_np - Establece el periodo de un thread *planificado-por-aplicación*.

Sinopsis

```
#include <pthread.h>

int pthread_set_appschedlib_period_np(
     pthread_t thread,
     struct timespec period);
```

Descripción

La función *pthread_set_appschedlib_period_np()* establece el valor de *period* del thread. El valor de *period* especifica el periodo del thread *planificado-por-aplicación*.

Valor de retorno

La función *pthread_set_appschedlib_period_np()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_set_appschedlib_period_np()* fallará si:

[EINVAL] El valor especificado por *period* no es válido.

[ESRCH] El valor especificado por *thread* no hace referencia a un thread existente.

pthread_get_appschedlib_period_np

Nombre

pthread_get_appschedlib_period_np - Obtiene el periodo de un thread *planificado-por-aplicación*.

Sinopsis

```
#include <pthread.h>

int pthread_get_appschedlib_period_np(
     pthread_t thread,
     struct timespec * period);
```

Descripción

La función *pthread_get_appschedlib_period_np()* obtiene el valor de *period* del thread. El valor de *period* especifica el periodo del thread *planificado-por-aplicación*.

Valor de retorno

La función *pthread_get_appschedlib_period_np()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_get_appschedlib_period_np()* fallará si:

[ESRCH] El valor especificado por *thread* no hace referencia a un thread existente.

pthread_set_appschedlib_deadline_np

Nombre

pthread_set_appschedlib_deadline_np - Establece el plazo relativo de un thread *planificado-por-aplicación*.

Sinopsis

```
#include <pthread.h>

int pthread_set_appschedlib_deadline_np(
     pthread_t thread,
     struct timespec * deadline);
```

Descripción

La función *pthread_set_appschedlib_deadline_np()* establece el valor de *deadline* del thread. El valor de *deadline* especifica el plazo relativo del thread *planificado-por-aplicación*.

Valor de retorno

La función *pthread_set_appschedlib_deadline_np()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_set_appschedlib_deadline_np()* fallará si:

[EINVAL] El valor especificado por *deadline* no es válido.

[ESRCH] El valor especificado por *thread* no hace referencia a un thread existente.

pthread_get_appschedlib_deadline_np

Nombre

pthread_get_appschedlib_deadline_np - Obtiene el plazo relativo de un thread *planificado-por-aplicación*.

Sinopsis

```
#include <pthread.h>

int pthread_get_appschedlib_deadline_np(
     pthread_t thread,
     struct timespec * deadline);
```

Descripción

La función *pthread_get_appschedlib_deadline_np()* obtiene el valor de *deadline* del thread. El valor de *deadline* especifica el plazo relativo del thread *planificado-por-aplicación*.

Valor de retorno

La función *pthread_get_appschedlib_deadline_np()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_get_appschedlib_deadline_np()* fallará si:

[ESRCH] El valor especificado por *thread* no hace referencia a un thread existente.

pthread_set_appschedlib_maxexectime_np

Nombre

pthread_set_appschedlib_maxexectime_np - Establece el tiempo de cómputo máximo de un thread *planificado-por-aplicación*.

Sinopsis

```
#include <pthread.h>

int pthread_set_appschedlib_maxexectime_np(
     pthread_t thread,
     struct timespec * wcet);
```

Descripción

La función *pthread_set_appschedlib_maxexectime_np()* establece el valor de *wcet* del thread. El valor de *wcet* especifica el tiempo de cómputo máximo del thread *planificado-por-aplicación*. Si el thread se ejecuta por más tiempo que *wcet*, y si está definida, se ejecutará la función manejadora de cómputo en exceso. Si el thread es planificador por algún servidor, y si esta definida, entonces se ejecuta la operación del planificador *thread_block()*.

Valor de retorno

La función *pthread_set_appschedlib_maxexectime_np()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_set_appschedlib_maxexectime_np()* fallará si:

[EINVAL] El valor especificado por *wcet* no es válido.

[ESRCH] El valor especificado por *thread* no hace referencia a un thread existente.

pthread_get_appschedlib_maxexectime_np

Nombre

pthread_get_appschedlib_maxexectime_np - Obtiene el tiempo de cómputo máximo de un thread *planificado-por-aplicación*.

Sinopsis

```
#include <pthread.h>

int pthread_get_appschedlib_maxexectime_np(
     pthread_t thread,
     struct timespec * wcet);
```

Descripción

La función *pthread_get_appschedlib_maxexectime_np()* obtiene el valor de *wcet* del thread. El valor de *wcet* especifica el tiempo de cómputo máximo del thread *planificado-por-aplicación*. Si el thread se ejecuta por más tiempo que *wcet*, y si está definida, se ejecutará la función manejadora de cómputo en exceso. Si el thread es planificador por algún servidor, y si esta definida, entonces se ejecuta la operación del planificador *thread_block()*.

Valor de retorno

La función *pthread_get_appschedlib_maxexectime_np()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_get_appschedlib_maxexectime_np()* fallará si:

[ESRCH] El valor especificado por *thread* no hace referencia a un thread existente.

pthread_set_appschedlib_schedhandlers_np

Nombre

pthread_set_appschedlib_schedhandlers_np - Establece los manejadores de cómputo en exceso y fallos en plazo de un thread *planificado-por-aplicación*.

Sinopsis

```
#include <pthread.h>

int pthread_set_appschedlib_schedhandlers_np(
     pthread_t thread,
     void (*overrun_handler)
            (pthread_t *thread),
     void (*deadline_miss_handler)
             pthread_t *thread);
```

Descripción

La función *pthread_set_appschedlib_schedhandlers_np()* establece los valores de los atributos *overrun_handler()* y *deadline_miss_handler()* del thread. Los valores de *overrun_handler()* y *deadline_miss_handler()* hacen referencia a los punteros a las funciones manejadoras de cómputo en exceso y fallo en plazo, respectivamente, del thread *planificado-por-aplicación*.

Valor de retorno

La función *pthread_set_appschedlib_schedhandlers_np()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_set_appschedlib_schedhandlers_np()* fallará si:

[EINVAL] Los valores especificado por *void (*overrun_handler) (pthread_t *thread)* y *void (*deadline_miss_handler) (pthread_t *thread)* no son válidos.

[ESRCH] El valor especificado por *thread* no hace referencia a un thread existente.

pthread_get_appschedlib_schedhandlers_np

Nombre

pthread_get_appschedlib_schedhandlers_np - Obtiene los manejadores de cómputo en exceso y fallos en plazo de un thread *planificado-por-aplicación*.

Sinopsis

```
#include <pthread.h>

int pthread_get_appschedlib_schedhandlers_np(
     pthread_t thread,
     void (*overrun_handler)
            (pthread_t *thread),
     void (*deadline_miss_handler)
             pthread_t *thread);
```

Descripción

La función *pthread_get_appschedlib_schedhandlers_np()* establece los valores de *overrun_handler()* y *deadline_miss_handler()* del thread. Los valores de *overrun_handler()* y *deadline_miss_handler()* hacen referencia a los punteros a las funciones manejadoras de cómputo en exceso y fallo en plazo, respectivamente, del thread *planificado-por-aplicación*.

Valor de retorno

La función *pthread_get_appschedlib_schedhandlers_np()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_get_appschedlib_schedhandlers_np()* fallará si:

[ESRCH] El valor especificado por *thread* no hace referencia a un thread existente.

pthread_get_appschedlib_exectime_np

Nombre

pthread_get_appschedlib_exectime_np - Obtiene el tiempo de cómputo utilizado por un thread *planificado-por-aplicación* en su última invocación.

Sinopsis

```
#include <pthread.h>

int pthread_get_appschedlib_exectime_np(
     pthread_t thread,
     struct timespec * exectime);
```

Descripción

La función *pthread_get_appschedlib_exectime_np()* obtiene el valor de *exectime* del thread. El valor de *exectime* especifica el tiempo de cómputo utilizado por el thread *planificado-por-aplicación* en su última invocación.

Valor de retorno

La función *pthread_get_appschedlib_exectime_np()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_get_appschedlib_exectime_np()* fallará si:

[ESRCH] El valor especificado por *thread* no hace referencia a un thread existente.

pthread_mutex_register_np

Nombre

pthread_mutex_register_np - Registra un monitor que será utilizado por un thread.

Sinopsis

```
#include <pthread.h>

int pthread_mutex_register_np(
     pthread_t thread,
     pthread_mutex_t * mutex);
```

Descripción

La función *pthread_mutex_register_np()* notifica al sistema que el monitor identificado por mutex será utilizado por el thread identificado por thread. Si la función se completa exitosamente, el monitor contendrá los valores de nivel de prioridad y nivel de expulsión equivalentes a la mayor prioridad y nivel de expulsión de los thread que utilicen al monitor.

Valor de retorno

La función *pthread_mutex_register_np()* retornará el valor de 0 si se completa exitosamente.

Errores

La función *pthread_mutex_register_np()* fallará si:

[EINVAL] El valor especificado por *mutex* no hace referencia a un monitor válido.

[ESRCH] El valor especificado por *thread* no hace referencia a un thread existente

Bibliografía

[1] *1984 /usr/group Standard*. The /usr/group Standards Committee, Santa Clara, California, USA, Nov, 1984.

[2] Abeni, L. and Buttazzo, G. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 4–13, Madrid, Spain, Dec 1998.

[3] Abeni, L. and Regehr, J. How to rapidly prototype a real-time scheduler. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (WiP)*, pages 4–13, Austin, Texas, Dec 2002.

[4] Aldea, M. and Gonzalez-Harbour, M. Posix-compatible application-defined scheduling in marte os. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, pages 67–75, June 2001.

[5] Aldea, M. and Gonzalez-Harbour, M. Application-defined scheduling. Draft 1.0, July 2002 submmited to the PASC SSWG Realtime Working Group (POSIX), Jul 2002.

[6] Aldea, M. and Gonzalez-Harbour, M. Evaluation of new posix real-time operating systems services for small embedded platforms. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems, ECRTS*, pages 161–168, Porto, Portugal, July 2003.

[7] Aldea, M. and Gonzalez-Harbour, M. A new generalized approach to application-defined scheduling. In *Proceedings of 16th Euromicro Conference on Real-Time Systems (WiP)*, pages 49–52, July 2004.

[8] Aldea, M., Miranda, J., and Gonzalez-Harbour, M. Integrating application-defined scheduling with the new dispatching policies for ada tasks. In *Proceedings of the 10th International Conference on Reliable Software Technologies Ada-Europe*, pages 220–235, York, UK, June 2005.

[9] Anderson, T.E., Bershad, B.N., Lazowska, E.D., and Levy, H.M. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the 13th ACM Symposium on Opearting Systems Design*, pages 95–109, Oct 1991.

[10] Audlsey, N.C, Burns, A., Davis, R.I., and Wellings, A.J. Integrating best effort and fixed priority scheduling. In *Proceedings of the IFAC/IFIP Workshop on Real-Time Programming*, Lake Constance, Germany, 1994.

[11] Audlsey, N.C., Burns, A., Richardson, M.F., and Welings, A.J. Hard real-time scheduling: The deadline-monotonic approach. In *Proceedings of the 8th IEEE Workshop on Real-Time Operating Systems and Software*, pages 133–138, May 1991.

[12] Baker, T. P. Stack-based scheduling of real-time procesess. *Real-Time Systems Journal*, 3(1):67–99, Mar 1991.

[13] Balbastre, P. and Ripoll, I. Integrated dynamic priority scheduler for rtlinux. In *Proceedings of the Third Real-Time Linux Workshop*, 2001.

[14] Baravanov, M. *A Linux-based Real-Time Operating System*. PhD thesis, New Mexico Institute of Mining and Technology, Socorro, New Mexico, USA, June 1997.

[15] Barreto, L.P. and Muller, G. Bossa: A language-based approach to the design of real-time schedulers. In *Proceedings of the 10th International Conference on Real-Time Systems*, pages 19–31, Mar 2002.

[16] Baruah, S. The limited-preemption uniprocessor scheduling of sporadic task. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems, ECRTS*, pages 137–144, Palma de Mallorca, Spain, Jul 2005.

[17] Baruah, S., Rosier, L., and Howell, R.R. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems Journal*, 2(4):301–324, Nov 1990.

[18] Bernat, G., Broster, I., and Burns, A. Rewriting history to exploit gain time. In *Proceedings of the 25th IEEE Real-Time Systems Symposium*, pages 328–335, Lisbon, Portugal, Dec 2004.

[19] Bini, E., Buttazzo, G.C., and Buttazzo, G.M. Rate monotonic analysis: The hyperbolic bound. *IEEE Transactions on Computers*, 52(7):933–942, Jul 2003.

[20] Burns, A. and Audsley, N. Scheduling hard real-time systems: A review. *Software Engineering Journal*, 6(3):116–128, May 1991.

[21] Buttazzo, G.C. Rate monotonic vs. edf: Judgment day. *Journal of Real-Time Systems*, 29(1):5–26, Jan 2005.

[22] Caccamo, M., Buttazzo, G., and Lui Sha. Capacity sharing for overrun control. In *Proceedings of the 21st IEE Real-Time Systems Symposium*, pages 295 – 304, Nov 2000.

[23] Caccamo, M. and Sha, L. Aperiodic servers with resource constraints. In *Proceedings of the 22th IEEE Real Time Systems Symposium*, pages 161–170, London, UK, Dec 2001.

[24] Candea, G.M. and Jones, M.B. Vassal: Loadable scheduler support for multi-policy scheduling. In *Proceedings of the 2nd USENIX Windows NT Symposium*, pages 157–166, Aug 1998.

[25] Carlow, G.D. Architecture of the space shuttle primary avionics software system. *Communications of ACM*, 27(9):926–936, Sep 1984.

[26] Chen, M. and Lin, K. Dynamic priority ceilings: a concurrency control protocol for real-time systems. *Real-Time Systems*, 2(4):325–346, Nov 1990.

[27] Cheng, S., Stankovic, J.A., and Ramamritham, K. Scheduling algorithms for hard real-time systems: a brief survey. *in Stankovic, J.A. and Ramamritham, K. (Eds) 'Tutorial: hard real-time systems' (IEEE)*, pages 150–173, 1988.

[28] Chetto, H. and Chetto, M. Some results of the earliest deadline scheduling algorithm. *IEEE Transactions on Software Engineering*, 10:1261–1269, Oct 1989.

[29] Coffman Jr., E.G. *Introduction to Deterministic Scheduling Theory*. John Wiley and Sons, New York, 1976.

[30] Cornhill, D. and Sha, L. Priority inversion in ada. *ADA Letters*, pages 30–32, Nov 1987.

[31] Crespo, A. and Ripoll, I. Ocera whitepaper. Technical report, OCERA Project and Universidad Politecnica de Valencia, 2003.

[32] Davis, R., Tindell, K, and Burns, A. Scheduling slack time in fixed priority pre-emptive systems. In *Proceedings of the 14th IEEE Real-Time Systems Symposium*, pages 222–231, Dec 1993.

[33] Davis, R. and Wellings, A. Dual priority scheduling. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 100–109, Dec 1995.

[34] de Niz, D., Abeni, L., Saewong, S., and Rajkumar, R. Resource sharing in reservation-based systems. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 171–180, Dec 2001.

[35] Deng, Z. and Liu, J.W.S. Scheduling real-time applications in an open environment. In *Proceedings of the 18th IEEE Real Time Systems Symposium*, pages 308–319, Dec 1997.

[36] Deng, Z. and Liu, J.W.S. A scheme for scheduling hard real-time applications un open systems environments. In *Proceedings of the 9th Euromicro Workshop on Real-Time Systems*, pages 191–199, Jun 1997.

[37] Dertouzos, M. Control robotics: The procedural control of physical processors. In *IFIP Congress*, pages 807–813, 1974.

[38] Diaz, A., Ripoll, I., Balbastre, P., and Crespo, A. A new application defined scheduler implementation in rtlinux. In *Proceedings of the Sixth Real-Time Linux Workshop*, Singapur, Nov 2004.

[39] Diaz, A., Ripoll, I., and Crespo, A. Extending the posix-compatible application-defined scheduling model. In *Proceedings of the 26th IEEE Real-Time Systems Symposium (WiP)*, Miami, Fl, Dec 2005.

[40] Diaz, A., Ripoll, I., and Crespo, A. A library framework for the posix application-defined scheduling proposal. In *Proceedings of the 2nd IEEE International Conference on Electrical and Electronics Engineering*, pages 21–26, Mexico City, Sep 2005.

[41] Diaz, A., Ripoll, I., and Crespo, A. On integrating posix signals into a real-time operating system. In *Proceedings of the Seventh Real-Time Linux Workshop*, Lille, France, Nov 2005.

[42] Ford, B. and Susarla, S. Cpu inheritance scheduling. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 91–105, Seattle, Washington, US, Nov 1996.

[43] Inc. FSMLabs. Open rtlinux patent license. http://www.fsmlabs.com/openpatentlicense.htm, Oct 2005.

[44] Gai,P., Abeni, L., Giorgi, M., and Butazzo, G. A new kernel approach for modular real-time systems development. In *Proceedings of the 13th*

Euromicro Conference on Real-Time Systems, pages 199–208, Delft, The Netherlands, Jun 2001.

[45] Gardner, M.K. and Liu, J.W.S. Performance of algorithms for scheduling real-time systems with overrun and overload. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pages 287–296, York, U.K., Jun 1999.

[46] Garey, M.R and Johnson, D.S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co, 1979.

[47] Ghazalie, T. M. and Baker, T. P. Aperiodic servers in deadline scheduling environment. *Real-Time Systems Journal*, 9(1):31–68, 1995.

[48] Goel, A., Abeni, L., Krasic, C., Snow, J., and Walpole, J. Supporting time-sensitive applications on a commodity os. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 165–180, Dec 2002.

[49] Harmon, M.G., Baker, T.P., and Whalley, D.B. A retargetable technique for predicting execution time. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 68–77, Dec 1992.

[50] IEEE 1003.1-1988. *IEEE Standard Portable Operating System Interface for Computer Environments, IEEE Std 1003.1-1988*. Institute of Electrical and Electronic Engineers, New York, NY, 1988.

[51] IEEE Std 1003.1, 2004 Edition. *The Open Group Technical Standard Base Specifications, Issue 6. Base Definitions*. Institute of Electrical and Electronic Engineers and The Open Group, Apr, 2004.

[52] IEEE Std 1003.1, 2004 Edition. *The Open Group Technical Standard. Base Specifications, Issue 6. Rationale (Informative)*. Institute of Electrical and Electronic Engineers and The Open Group, Apr, 2004.

[53] IEEE Std 1003.1, 2004 Edition. *The Open Group Technical Standard. Base Specifications, Issue 6. Shell and Utilities*. Institute of Electrical and Electronic Engineers and The Open Group, Apr, 2004.

[54] IEEE Std 1003.1, 2004 Edition. *The Open Group Technical Standard. Base Specifications, Issue 6. System Interfaces*. Institute of Electrical and Electronic Engineers and The Open Group, Apr, 2004.

[55] IEEE Std 1003.13-2003. *IEEE Standard for Information Technology - Standardized Application Environment Profile (AEP) - POSIX Realtime and Embedded Application Support*. Institute of Electrical and Electronic Engineers, 2003.

[56] Jazequel, J.M. and Mayer, B. A generalized processor sharing approach to flow control in integrated services networks: The multiplenode case. *IEEE/ACM Transactions on Networking*, 2(2):137–150, Apr 1994.

[57] Jefay, K. Scheduling sporadic tasks with shared resources. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 89–99, Phoenix, USA, Dec 1992.

[58] Klein, M.H. et. al. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis of Real-Time Systems*. Kulwer Academic Publishers, 1993.

[59] Kuo, T.W. and Li, C.H. A fixed-priority-driven open environment for real-time applications. In *Proceedings of the 20th Real-Time Systems Symposium*, pages 256–267, 1999.

[60] Labetoulle, L. Some theorems on real-time scheduling. In E. Gelembe and R. Mahl, editors, *Computer Architectures and Networks*. North Holland Publishing Company, 1974.

[61] Lamastra, G., Lipari, G., and Abeni, L. A bandwidth inheritance algorithm for real-time task synchronization in open systems. In *Proceedings of the 22th IEEE Real Time Systems Symposium*, pages 151–160, London, UK, Dec 2001.

[62] Lawall, J.L., Muller, G., and Duchesne, H. Language design for implementing process scheduling hierarchies. In *Proceedings of the ACM Symposium on Partial Evaluation and Program Manipulation*, pages 80–91, Aug 2004.

[63] Lehoczky, J.P. Fixed priority scheduling of periodic task set with arbitrary deadlines. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 201–209, Dec 1990.

[64] Lehoczky, J.P. and Ramos-Thuel, S. An optimal algorithm for scheduling soft aperiodic tasks in fixed priority preemptive systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 110–123, Dec 1992.

[65] Lehoczky, J.P., Sha, L., and Ding, Y. The rate monotonic scheduling algorithm: Exact characterisation and average case behavior. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 166–171, Dec 1989.

[66] Lehoczky, J.P., Sha, L., and Strosnider, J.K. Enhanced aperiodic responsiveness in hard real-time environments. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 261–270, Dec 1987.

[67] Leung, J.Y. and Merril, M.L. A note on preemptive scheduling of periodic, real-time tasks. *Information Processing Letters*, 3(11), 1980.

[68] Leung, J.Y. and Whitehead, J. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation (Netherlands)*, 4(2):237–250, Dec 1982.

[69] Lin, C. and Brandt, S.A. Improving soft real-time performance through better slack reclaiming. In *Proceedings of the 26th IEEE Real-Time Systems Symposium*, pages 3–14, Miami, Fl, Dec 2005.

[70] Lipari, G. and Barauh, S.K. Efficient scheduling of real-rime multi-task applications in dynamic systems. In *Proceedings of the Real-Time Technology and Applications Symposium*, pages 166–175, Washington, DC., May 2000.

[71] Lipari, G. and Barauh, S.K. A framework for achieving inter-application isolation in multiprogrammed, hard real-time environments. In *Proceedings of the Real-Time Systems Symposium*, pages 217–226, Orlando, Fl., Nov 2000.

[72] Lipari, G. and Barauh, S.K. Greedy reclaimation of unused bandwidth in cosntant bandwidth servers. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, pages 192–200, Stokholm, Sweden, June 2000.

[73] Lipari, G. and Buttazzo, G.C. Schedulability analysis of periodic and aperiodic tasks with resource constraints. *Journal of Systems Architecture: the EUROMICRO Journal*, 46(4):327–338, Feb 2000.

[74] Lipari, G., Lamastra, G., and Abeni, L. Task synchronization in reservation-based real-time systems. *IEEE Transactions on Computers*, 53(12):1591–1601, Dec 2004.

[75] Liu, J. and Lee, E.A. Timed multitasking for real-time embedded software. *IEEE Control Systems, special issue on Software-Enabled Control*, 23(1):65–75, Jan 2003.

[76] Liu, J.W.S. *Real-Time Systems*. Prentice Hall, 2000.

[77] Liu, J.W.S. and Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, May 1973.

[78] Marzario, L., Lipari, G., Balbastre, P., and Crespo, A. Iris: A new reclaiming algorithm for server-based teal-time system. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 211–218, May 2004.

[79] Mejia-Alvarez, P., Aydin, H., Mosse, D., and Melhem, R. Scheduling optional computations in fault-tolerant real-time systems. In *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications*, pages 323–330, Dec 2000.

[80] Mok, A.K. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environments*. PhD thesis, MIT, Cambridge, MA, 1983.

[81] Mok, A.K. and Dertouzos, M. Multiprocessor scheduling in a hard real-time environment. In *Proceedings of the 7th Texas Conference on Computing Systems*, Nov 1978.

[82] Muller, G., Lawall, J.L., and Duchesne, H. A framework for simplifying the development of kernel schedulers: Design and performance evaluation. In *Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 219–230, Feb 2005.

[83] Rajkumar, R., Juvva, K., Molano, A., and Oikawa, S. Resource kernels: A resource-centric approach to real-time and multimedia system. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, San Jose, CA, Jan 1998.

[84] Ramamritham, K. and Stankovic, J.A. Scheduling algorithms and operating systems support for real-time systems. In *Proceedings of IEEE*, pages 56 – 67, Jan 1994.

[85] Ramamurthy, S. *A Lock-Free Approach to Object Sharing in Real-Time Systems*. PhD thesis, The University of North Carolina at Chapel Hill, Chapel Hill, USA, 1997.

[86] Ramos-Thuel, S. and Lehoczky, J.P. On line scheduling of hard deadline aperiodic tasks in fixed-priority systems. In *Proceedings of the 14th IEEE Real-Time Systems Symposium*, pages 160–171, Dec 1993.

[87] Ramos-Thuel, S. and Lehoczky, J.P. Algorithms for scheduling hard aperiodic tasks in fixed-priority systems using slack stealing. In *Proceedings of the 15th IEEE Real-Time Systems Symposium*, pages 22–35, 1994.

[88] Regehr, J. and Stankovic, J.A. Hls: a framework for composing soft real-time schedulers. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 3–14, Dec 2001.

[89] Ripoll, I. *Planificación con Prioridades Dinámicas en Sistemas de Tiempo Real Crítico*. PhD thesis, Universidad Politécnica de Valencia, Valencia, España, 1996.

[90] Ripoll, I., Crespo, A., and Garcia-Fornes, A. An optimal algorithm for scheduling soft aperiodic tasks in dynamic-priority preemptive systems. *IEEE Transactions on Software Engineering*, 23(6):388–400, June 1997.

[91] Ripoll, I., Crespo, A., and Mok, A. Improvement in feasibility testing for real-time tasks. *Real-Time Systems*, 11(1):19 – 39, July 1996.

[92] Ripoll I. et al. Rtos state of the art analysis. Technical report, OCERA Project, 2003.

[93] Ripoll I. et al. Ocera deliverable d5.4rep - scheduling components v2. Technical report, OCERA Project, Feb 2004.

[94] Russinovich, M.E. and Solomon, D.A. *Microsoft Windows Internals*. Microsoft Press, fourth edition edition, 2004.

[95] Santos, R., Lipari, G., and Santos, J. Scheduling open dynamic systems: The clearing fund algorithm. In *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications*, pages 25–27, Gothenburg, Sweden, Aug 2004.

[96] Schwan, K., Bihari, T., Weide, B.W., and Taulbee, G. High-performance opareting systems primitives for robotics and real-time control systems. *ACM Transactions on Computer Systems*, 5(3):189–231, Aug 1987.

[97] Sha, L., Abdelzaher, T, Arzen, K., Cervin, A., Baker, T.P., Burns, A., Buttazzo, G., Caccamo, M., Lehoczky, J., and Mok, A. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28(2):46–61, Nov 2004.

[98] Sha, L., Lehoczky, J.P., and Rajkumar, R. Solutions for some practical problems in prioritizing preemptive scheduling. In *Proceedings of the 7th IEEE Real Time Systems Symposium*, volume 48, pages 181–191, New Orleans, Lousiana, Dec 1986.

[99] Sha, L., Rajkumar, R., and Lehoczky, J.P. Real-time synchronization protocol for multiprocessors. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 259–269, New York, USA, Dec 1988.

[100] Sha, L., Rajkumar, R., and Lehoczky,J.P. Priority inheritance protocols: An approach to realtime synchronisation. *IEEE Transactions on Computers*, 39(9):1175–1185, Sep 1990.

[101] Silberschatz, A., Galvin, P.B., and Gagne, G. *Operating Systems Concepts*. John Wiley and Sons, seventh edition edition, 2005.

[102] Sprunt, B., Sha, L., and Lehoczky, J. P. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems Journal*, 1(1):27–60, 1989.

[103] Spuri, M. *Earliest Deadline Scheduling in Real-Time Systems*. PhD thesis, Scuola Superiore S. Anna, Pisa, Italy, 1995.

[104] Spuri, M. and Buttazzo, G. Scheduling aperiodic tasks under dynamic priority systems. *Real-Time Systems Journal*, 10(2):179–210, Mar 1996.

[105] Stankovic, J.A., Spuri, M., Ramamritham, K., and Buttazzo, G.C. *Deadline Scheduling for Real-Time Systems. EDF and Related Algorithms*. Kluwer Academic Publishers, 1998.

[106] Stesuloff, H.U. Advanced real-time languages for distributed industrial process control. *IEEE Computer*, 17(2):37–46, Feb 1984.

[107] Stewart, D.B. and Khosla, P.K. Mechanisms for detecting and handling timing errors. *Communications of the ACM*, 40(1):87 – 94, Jan 1997.

[108] Stodolsky, D., Chen, J.B., and Bershad, B.N. Fast interrupt priority management in operating system kernels. In *Proceedings of the 2nd USENIX Symposium on Microkernels and Other Kernel Architectures*, pages 105–110, San Diego, CA, USA, Sep 1993.

[109] Strosnide, J.K., Lehockzy, J.P., and Sha, L. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers*, 44(1):73–91, Jan 1995.

[110] Vidal, J., Gonzales, F., and Ripoll, I. Posix signals implementation in rtlinux. Technical report, OCERA Project and Universidad Politecnica de Valencia, 2002.

[111] Vidal, J., Ripoll, I., Balbastre, P., and Crespo, A. Application defined scheduler implementation in rtlinux. In *Proceedings of the Fifth Real-Time Linux Workshop*, Nov 2003.

[112] Wang, Y.C. and Lin, K.J. Enhancing the real-time capability of the linux kernel. In *Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications*, pages 11–20, Oct 1998.

[113] Wolf, V.F. Rapidsched: Static scheduling and analysis for real-time corba. Technical report, University of Rhode Island, Department of Computer Science and Statistics, Nov 2000.

[114] Xu, L. and Parnas, D.L. On satisfying timing constraints in hard-real-time systems. *IEEE Transactions on Software Engineering*, 19:70–84, Jan 1993.

[115] Yodaiken, V. The rtlinux manifesto. In *Proceedings of the 5th Linux Expo*, Raleigh, NC, Mar 1999.