UNIVERSIDAD POLITÉCNICA DE VALENCIA
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

EXTENSIONES AL LENGUAJE ADA Y A LOS SERVICIOS POSIX PARA PLANIFICACIÓN EN SISTEMAS DE TIEMPO REAL ESTRICTO.

TESIS DOCTORAL

DIRIGIDA POR:
DRA. DÑA. ANA GARCÍA FORNES
DR. D. ALFONS CRESPO I LORENTE

PRESENTADA POR:
AGUSTÍN RAFAEL ESPINOSA MINGUET

VALENCIA, 2003

A Agustín, mi padre

Agradecimientos

En primer lugar, quisiera agradecer a mis directores de tesis, Ana García Fornes y Alfons Crespo i Lorente, su dedicación. Sin ellos, esta tesis nunca hubiera podido ser terminada y, mucho más importante, comenzada. Mi agradecimiento, como seguro ambos saben, va más allá de la labor que han realizado como mis directores.

Quisiera también agradecer a mis compañeros y compañeras de los grupos Tecnología Informática e Informática Industrial el privilegio de poder trabajar con ellos. A dos de estos compañeros, Andrés Terrasa Barrena y Vicente Botti Navarro, les debo algo muy especial. A Andrés, las incontables ocasiones que ha tenido que soportar que le pidiese ayuda y su infinita paciencia. A Vicente, su constancia en animarme en mi trabajo y su particular forma de hacerlo.

Por último, y sobre todo, gracias a mi mujer Amparo y a mis hijos Víctor y Andrea. Han sido muchos los días que han tenido que apoyarme para que yo pudiera realizar esta tesis, y por ello les estoy y estaré inmensamente agradecido.

Resumen

Esta tesis se ha centrado en el estudio de las políticas basadas en la extracción de holgura y en la asignación dual de prioridades, ambas orientadas a dar servicio a tareas aperiódicas en sistemas de tiempo real estricto. Estas políticas constituyen una interesante alternativa a las políticas basadas en servidores de carga aperiódica y han sido ampliamente estudiadas en numerosos trabajos. No obstante, la posibilidad de ser aplicables en la práctica todavía no había sido evaluada en detalle, y éste ha sido el principal objetivo de esta tesis.

En este trabajo, ambas políticas son revisadas en profundidad. En el caso de la política basada en la extracción de holgura, algunas de sus principales debilidades han sido corregidas. En particular, se muestra que es posible compartir recursos entre tareas críticas y aperiódicas de una forma sencilla y eficiente, utilizando para ello los mismos protocolos de herencia de prioridad que pueden aplicarse en sistemas en los que sólo existen tareas críticas.

La aplicabilidad de estas políticas ha sido valorada incorporándolas en los dos entornos más relevantes hoy en día para la programación de sistemas de tiempo real estricto, el lenguaje de programación Ada y los servicios POSIX. Con este fin se han definido nuevas interfaces para ambas políticas en estos entornos, coherentes con sus principios de diseño y con los servicios que actualmente ofrecen. El diseño de estas interfaces ha supuesto una adaptación de ambas políticas buscando, en todo momento, un equilibrio entre prestaciones y eficiencia. Como parte de este diseño, estas interfaces han sido implementadas en el sistema operativo MaRTE OS. El proceso de implementación ha servido para validar las distintas alternativas que han sido consideradas.

Un aspecto importante de una política de planificación es su eficiencia, por lo que este aspecto ha sido evaluado en detalle. Se ha caracterizado de forma analítica el efecto de la implementación de estas políticas en los tiempos de respuesta de las tareas. Este efecto ha sido cuantificado, y se ha comparado la eficiencia de estas políticas respecto a la de la política basada en prioridades fijas con desalojo, normalmente utilizada para la

programación de sistemas de tiempo real. Los resultados obtenidos muestran que, aunque la sobrecarga que introducen es significativa, ésta se mantiene en un rango aceptable.

También se han estudiado los recientemente aprobados servicios de trazas definidos por el estándar POSIX. La necesidad de disponer de instrumentos de análisis y medición para poder abordar los trabajos realizados en esta tesis, motivó la implementación de estos servicios en el sistema MaRTE OS. En relación a este aspecto, se ha propuesto una interfaz POSIX/Ada para estos servicios y se ha abordado la problemática de obtener métricas temporales del sistema a partir de la información que se recoge en las trazas.

Resum

Aquesta tesi s'ha centrat a l'estudi de les polítiques basades en l'extracció de folgança i en l'assignació dual de prioritats, totes dos orientades a donar un servei a tasques aperiòdiques en sistemes de temps real estricte. Aquestes polítiques constitueixen una interessant alternativa a les polítiques basades en servidors de càrrega aperiòdica i han sigut àmpliament estudiades a nombrosos treballs. No obstant això, la possibilitat d'esser aplicables a la pràctica encara no havia sigut avaluada en detall, i aquest ha sigut el principal objectiu d'aquesta tesi.

En aquest treball, les dues polítiques han sigut revisades en profunditat. En el cas de la política basasaa en l'extracció de folgança, algunes de les seues principals debilitats han sigut corregides. En particular, es mostra que és possible la compartició de recursos entre tasques crítiques i aperiòdiques d'una forma senzilla i eficient, utilitzant per tal de fer-ho els mateixos protocols d'herència de prioritats que poden aplicar-se a sistemes als que només existeixen tasques crítiques.

L'aplicabilitat d'aquestes polítiques ha sigut valorada mitjançant la seua incorporació als dos entorns més relevants hui en dia per a la programació de sistemes de temps real estricte, el llenguatge de programació Ada i els serveis POSIX. Amb aquest fi s'han definit noves interfícies per ambdues polítiques en aquestos entorns, coherents amb els seus principis de diseny i amb els serveis que proporcionen actualment. El diseny de les interficies ha suposat una adaptació de ambdues polítiques, buscant en tot moment un equilibri entre prestacions i eficiència. Com a part d'aquest diseny, aquestes interfícies han sigut implementades al sistema operatiu MaRTE OS. El procés de implementació ha servit per a validar les diferents alternatives que han sigut considerades.

Un aspecte important d'una política de planificació és la seua eficiència, per la qual cosa aquest aspecte has sigut avaluat en detall. S'ha caracteritzat d'una manera analítica l'efecte de la implementació d'aquestes polítiques als temps de resposta de les tasques. Aquest efecte ha sigut quantificat i s'ha comparat l'eficiència d'aquestes polítiques respecte a la de la política basada en prioritas fixes amb desallotjament, normalment utilitza-

da per a la programació de sistemes de temps real estricte. Els resultats obtinguts mostren que, encara que la sobrecàrrega que introdueixen és significativa, aqueste es manté a un rang acceptable.

També s'han estudiat els recentment aprovats serveis de traça definits per l'estàndar POSIX. La necessitat de disposar de instruments d'anàlisi i medició per poder abordar els treballs d'aquesta tesi va motivar la implementació d'aquestos serveis al sistema MaRTE OS. En relació a aquest aspecte, s'ha proposat una interfície POSIX/Ada per aquestos serveis i s'ha abordat la problemàtica d'obtindre métriques temporals del sistema mitjançant la informació que es recull de les traces.

Abstract

This thesis is centered on the study of slack stealing and dual priority scheduling policies, both oriented to serve aperiodic tasks in hard real-time systems. These polices, largely studied in previous work, constitute an interesting alternative to policies based on aperiodic servers. However, the possibility of actually being put into practice had not been yet exhaustively evaluated. This has been the main objective of this thesis.

Both policies are profoundly revised in this dissertation. In the case of the slack stealing policy, some of its weaknesses have been solved. In particular, this thesis shows that it is possible to share resources between hard and aperiodic tasks in a simple and efficient manner. This can be done by using the same priority inheritance protocols which are used in systems comprising hard tasks only.

The applicability of these scheduling policies has been evaluated by means of their incorporation into the two most relevant hard real-time programming environments in use nowadays, the Ada programming language and the POSIX set of services. In order to fulfill this purpose, new interfaces for both policies have been defined in these two environments. These interfaces have been designed in order to be coherent with the environments' design principles and the services they currently support. This design has involved the adaptation of the original scheduling policies, trying to optimize the balance between performance and efficiency. As a part of this design, both interfaces have been implemented in the MaRTE OS operating system. These implementations have been useful for validating the different alternatives which have been considered throughout the design process.

Since efficiency is an important aspect of any scheduling policy, this aspect has been extremely evaluated in this work. The effect of the implementation of both policies to the task response times has been analytically characterized. This effect has also been quantified, and then the efficiency of both policies has been compared with the fixed-priority preemptive scheduling policy, which is the one normally used in hard real-time systems. The results of this comparative study show that, although the overhead introduced by any

of the two new policies is significant, this overhead lies in a reasonable range.

The recently approved POSIX trace services have also been studied in this thesis. The necessity of having some analysis and measurement tools available for the efficiency studies carried out in this thesis led to the incorporation of these services into MaRTE OS. Related to this, a new POSIX/Ada interface for the trace services has also been proposed. In addition, the problem of obtaining temporal metrics of the system from the information of the traces has also been covered.

Índice general

Agradecimientos I

Resumen III

Índice General IX

Índice de Figuras XIII

Índice de Tablas XV

Nomenclatura XVII

1. Introducción 1

1.1. Motivación 1

1.2. Objetivos 4

1.3. Estructura 4

2. Planificación en Sistemas de Tiempo Real 7

2.1. Sistemas de Tiempo Real Estricto 7

2.1.1. Análisis de la Planificabilidad 10

2.1.2. Interacción entre Tareas 13

2.1.3. Inclusión de Tareas Aperiódicas 14

2.2. El Algoritmo Extractor de Holgura Dinámico 19

2.3. El Algoritmo de Asignación Dual de Prioridades 26

2.4. Modificaciones al Algoritmo Dinámico Extractor de Holgura 31

2.4.1. Compartición de Recursos entre Tareas Críticas y Aperiódicas . . 31
2.4.2. Coexistencia con Tareas Esporádicas 41
2.5. Resumen y Conclusiones 46

3. Nuevas Interfaces Ada y POSIX 49
3.1. El Lenguaje de Programación Ada 51
3.2. La Interfaz de Aplicación POSIX 54
3.3. Perspectiva General 56
3.4. Nuevas Políticas de Planificación 61
3.4.1. Interfaz Ada 61
3.4.2. Interfaz POSIX 62
3.4.3. Comentarios 62
3.5. Notificación de Instancia Terminada 62
3.5.1. Interfaz Ada 63
3.5.2. Interfaz POSIX 63
3.5.3. Comentarios 63
3.6. Planificación Basada en la Extracción de Holgura 66
3.6.1. Interfaz Ada 66
3.6.2. Interfaz POSIX 69
3.6.3. Comentarios 70
3.7. Planificación Basada en la Asignación Dual de Prioridades 74
3.7.1. Interfaz Ada 74
3.7.2. Interfaz POSIX 75
3.7.3. Comentarios 76
3.8. Modificación de Parámetros de Planificación 77
3.8.1. Interfaz Ada 78
3.8.2. Interfaz POSIX 80
3.8.3. Comentarios 81
3.9. Secciones de Cómputo Opcional 82
3.9.1. Interfaz Ada 83
3.9.2. Interfaz POSIX 84

3.9.3. Comentarios 87
3.10. Implementación en MaRTE OS 91
3.11. Resumen y Conclusiones 95

4. Servicios de Trazas POSIX **99**
4.1. Introducción a los Servicios de Trazas POSIX 100
4.2. Interfaz POSIX/Ada para los Servicios de Trazas POSIX 104
4.2.1. Directrices de Diseño 104
4.2.2. Definición 112
4.3. Adecuación a los Sistemas de Tiempo Real 132
4.4. Implementación de los Servicios de Trazas POSIX en MaRTE OS 134
4.5. Obtención de Métricas Temporales 140
4.5.1. Interfaz 140
4.5.2. Implementación 144
4.6. Resumen y Conclusiones 148

5. Análisis de Eficiencia **151**
5.1. Análisis de Planificabilidad. Prioridades Fijas con Desalojo 152
5.2. Análisis de Planificabilidad. Extracción de Holgura 159
5.3. Análisis de Planificabilidad. Asignación Dual de Prioridades 162
5.4. Comparación de Algoritmos Respecto a la Planificabilidad 163
5.5. Utilización de Interrupciones Diferidas 173
5.6. Optimización del Algoritmo Extractor de Holgura 181
5.7. Comparación de Algoritmos Respecto a la Sobrecarga del Sistema 183
5.8. Secciones de Cómputo Opcional 188
5.9. Resumen y Conclusiones 199

6. Conclusiones y Trabajos Futuros **203**
6.1. Conclusiones 203
6.2. Trabajos Futuros 207

Índice de figuras

2.1. Concepto de Holgura I 19
2.2. Concepto de Holgura II 20
2.3. Servicio Aperiódico de Baja Prioridad 22
2.4. Servicio Aperiódico con el Algoritmo DSSA 22
2.5. Servicio Aperiódico de Baja Prioridad 28
2.6. Servicio Aperiódico con el Algoritmo DP 28
2.7. Recursos Compartidos con el Algoritmo Extractor de Holgura I 36
2.8. Recursos Compartidos con el Algoritmo Extractor de Holgura II 37
2.9. Implementación del Método de Asignación de Prioridades I 41
2.10. Implementación del Método de Asignación de Prioridades II 42
2.11. Derivación de la Holgura en Presencia de Tareas Esporádicas. Método Original 45
2.12. Derivación de la Holgura en Presencia de Tareas Esporádicas. Método Propuesto 46

3.1. Partes Opcionales (Extracción de Holgura) 89
3.2. Partes Opcionales (Asignación Dual de Prioridades) 89
3.3. Partes Opcionales en Tareas (Extracción de Holgura) 92
3.4. Partes Opcionales en Tareas (Asignación Dual de Prioridades) 92

4.1. Símbolos utilizados por Quivi 135
4.2. Visualización en Quivi. 139
4.3. Autómata $Delay\_Until\_Wakeup_i$ 147

5.1. Actividades del Sistema Relacionadas con Cada Instancia. (Valores en μs) 154
5.2. Bloqueo Causado por el Sistema. (Valores en μs) 155
5.3. Activación de Tareas Menos Prioritarias en Ada. (Valores en μs) 157
5.4. Activación de Tareas Menos Prioritarias en POSIX. (Valores en μs) . . . 157
5.5. Variación del Instante de Activación. (Valores en μs) 159
5.6. Detección de Fin de Holgura 161
5.7. Instantes de Promoción 163
5.8. Sobrecarga del Sistema. Caso Ada. (Valores en μs) 167
5.9. Sobrecarga del Sistema. Caso POSIX. (Valores en μs) 168
5.10. Diferencias en Tiempos de Respuesta. Caso Ada.(Valores en μs) 171
5.11. Diferencias en Tiempos de Respuesta. Caso POSIX. (Valores en μs) . . . 172
5.12. Sobrecarga del Sistema. Caso Ada con Interrupciones Diferidas. (Valores en μs) 176
5.13. Sobrecarga del Sistema. Caso POSIX con Interrupciones Diferidas. (Valores en μs) 177
5.14. Diferencias en Tiempos de Respuesta. Caso Ada con Interrupciones Diferidas. (Valores en μs) 179
5.15. Diferencias en Tiempos de Respuesta. Caso POSIX con Interrupciones Diferidas. (Valores en μs) 180
5.16. Utilización del Sistema (%). Caso Ada. 184
5.17. Utilización del Sistema (%). Caso POSIX. 184
5.18. Utilización del Sistema (%). Caso Ada con Interrupciones Diferidas. . . . 185
5.19. Utilización del Sistema (%). Caso POSIX con Interrupciones Diferidas. . 185
5.20. La Parte Opcional es Activada. (Valores en μs) 193
5.21. La Parte Opcional Termina. (Valores en μs) 194
5.22. La Parte Opcional es Abortada. (Valores en μs) 194
5.23. La Tarea Opcional es Activada. (Valores en μs) 197
5.24. La Tarea Opcional Termina. (Valores en μs) 197
5.25. La Tarea Opcional es Abortada. (Valores en μs) 198

Índice de tablas

4.1. Resultado de una Métrica. (Valores en μs) 144

5.1. Conjunto de Tareas de Ejemplo 164
5.2. Tiempos de Respuesta. Caso Ada. (Valores en μs) 169
5.3. Tiempos de Respuesta. Caso POSIX. (Valores en μs) 170
5.4. Tiempos de Respuesta. Caso Ada con Interrupciones Diferidas. (Valores en μs) 175
5.5. Tiempos de Respuesta. Caso POSIX con Interrupciones Diferidas. (Valores en μs) 178
5.6. Descripción de los Términos Utilizados 188
5.7. Mediciones (μs). Carga Aperiódica del 0 %. Caso Ada. 189
5.8. Mediciones (μs). Carga Aperiódica del 50 %. Caso Ada. 189
5.9. Mediciones (μs). Carga Aperiódica del 100 %. Caso Ada. 189
5.10. Mediciones (μs). Carga Aperiódica del 0 %. Caso Posix. 190
5.11. Mediciones (μs). Carga Aperiódica del 50 %. Caso Posix. 190
5.12. Mediciones (μs). Carga Aperiódica del 100 %. Caso Posix. 190
5.13. Mediciones (μs). Carga Aperiódica del 0 %. Caso Ada con Interrupciones Diferidas. 191
5.14. Mediciones (μs). Carga Aperiódica del 50 %. Caso Ada con Interrupciones Diferidas. 191
5.15. Mediciones (μs). Carga Aperiódica del 100 %. Caso Ada con Interrupciones Diferidas. 191
5.16. Mediciones (μs). Carga Aperiódica del 0 %. Caso POSIX con Interrupciones Diferidas. 192

5.17. Mediciones (μs). Carga Aperiódica del 50 %. Caso POSIX con Interrupciones Diferidas. 192
5.18. Mediciones (μs). Carga Aperiódica del 100 %. Caso POSIX con Interrupciones Diferidas. 192
5.19. Sobrecarga Partes Opcionales. (Valores en μs) 195
5.20. Sobrecarga Partes Opcionales en Tareas. (Valores en μs) 198

Nomenclatura

i	Prioridad.
τ_i	Tarea. Normalmente con prioridad base igual a i.
T_i	Periodo de la tara τ_i.
C_i	Tiempo de cómputo en el peor caso de la tarea τ_i.
D_i	Plazo máximo de ejecución para la tarea τ_i.
B_i	Factor de bloqueo de la tarea τ_i.
J_i	Variación entre activaciones de la tarea τ_i.
w_i	Ventana temporal que mide el tiempo de respuesta de la tarea τ_i.
R_i	Tiempo de respuesta de la tarea τ_i.
$lp(i)$	Conjunto de tareas cuya prioridad es inferior a i.
$hp(i)$	Conjunto de tareas cuya prioridad es superior a i.
$hep(i)$	Conjunto de tareas cuya prioridad es igual o superior a i.
$S_i^{max}(t)$	Holgura teórica al nivel de prioridad i en el instante t.
$S_i(t)$	Holgura calculada al nivel de prioridad i en el instante t.
$I_j(t,e)$	Interferencia causada por la tarea j a partir del instante t hasta el instante e.
$x_i(t)$	Instante en que se producirá la siguiente activación de la tarea τ_i.
$d_i(t)$	Siguiente instante de tiempo en que se producirá el vencimiento del plazo de la tarea τ_i
$c_i(t)$	Tiempo de ejecución restante de la instancia actual de la tarea τ_i.
$(exp)_0$	El valor de la expresión si es mayor que cero o cero en otro caso.
Y_i	Tiempo de promoción para la tarea τ_i.
Y_i^{max}	Máximo tiempo de promoción para que la tarea τ_i no pierda su plazo.
$y_i(t)$	Próximo instante de promoción de la tarea τ_i.

Capítulo 1

Introducción

1.1. Motivación

Los sistemas de tiempo real se definen normalmente como sistemas en los cuales su corrección depende no sólo del resultado lógico de la computación, sino también del instante en que dichos resultados se producen. Esto es debido a que estos sistemas suelen interactuar con un proceso o entorno real, y sus acciones deben ser realizadas cumpliendo los requisitos temporales que el entorno real impone. El diseño de sistemas de tiempo real que sean capaces de cumplir con sus requisitos temporales no es una tarea sencilla y por ello, en las dos últimas décadas, se ha realizado un importante esfuerzo de investigación en este campo.

Los resultados más relevantes se han obtenido en relación al paradigma de planificación basada en prioridades fijas. Utilizando este paradigma, una aplicación de tiempo real puede construirse como un conjunto de tareas periódicas, donde cada tarea debe tener un comportamiento temporal predecible. El comportamiento temporal de cada tarea es determinado por varios atributos temporales, tales como su periodo, su tiempo máximo de ejecución y su plazo. Bajo ciertas restricciones, a partir de estos atributos puede demostrarse de forma analítica si una tarea se ejecutará dentro de su plazo. De esta forma, es posible programar aplicaciones de tiempo real capaces de interactuar con su entorno de una forma predecible en el tiempo.

Disponer de teorías sólidas y bien asentadas en el ámbito de la programación de sistemas de tiempo real es, sin lugar a dudas, crucial, pero también lo es la existencia de entornos de programación donde estas teorías puedan ser aplicadas. Los sistemas operativos y los lenguajes de programación han evolucionado de forma notable para ofrecer entornos donde programar sistemas de tiempo real. En la actualidad, los entornos de programa-

ción que ofrecen por un lado el lenguaje de programación Ada, y por otro, los servicios definidos por el estándar POSIX, son, sin duda, una referencia para la programación de sistemas de tiempo real.

El lenguaje de programación Ada fue creado en 1983 a petición del Departamento de Defensa de los EEUU, con el objetivo de ser utilizado para la programación de sistemas empotrados. Es precisamente en esa misma década cuando se producen los resultados más importantes en el campo de los sistemas de tiempo real, resultados que mostraron las debilidades de este lenguaje respecto al fin para el cual había sido creado. En 1988 comenzó el proceso de revisión del lenguaje, con el objetivo de disponer de una nueva versión en la década de los 90. Una de las áreas más mejorada fue la de la programación de sistemas de tiempo real. En este ámbito, se redefinió el lenguaje con el fin de que ofreciese los servicios necesarios para poder aplicar las nuevas teorías que se habían desarrollado. Esta revisión culminó con la publicación en 1995 de la nueva definición del lenguaje, al que comúnmente se le denomina Ada 95 [2].

El estándar POSIX surgió para unificar las múltiples versiones existentes de interfaces de acceso a los servicios del sistema operativo Unix. Su primera versión, publicada en 1988, no contemplaba aspectos relacionados con los sistemas de tiempo real pero, pocos años más tarde, POSIX fue ampliado y modificado para abordar este tipo de sistemas. Al igual que en el caso del lenguaje Ada, la influencia de las investigaciones realizadas es muy notable. La evolución de POSIX hacia los sistemas de tiempo real ha sido casi frenética en estos últimos años y, en la última versión de este estándar [55], publicada en el año 2001, se dispone de un abanico muy amplio de servicios orientados a la programación de sistemas de tiempo real. Los sistemas operativos de tiempo real más importantes ofrecen una interfaz conforme a POSIX, entre los que cabría destacar VxWorks, pSOSystem, QNX y LynxOS, sistemas que han sido utilizados en multitud de aplicaciones de tiempo real.

No obstante, aunque la programación de sistemas de tiempo real ha alcanzado un grado elevado de madurez, posiblemente quede todavía mucho por conseguir. Una de las áreas que hasta la fecha se ha abordado con cierta timidez, es la de los sistemas de tiempo real en los que coexisten tareas con requisitos temporales estrictos (tareas críticas) junto con tareas cuyos requisitos no son tan estrictos. A este último tipo de tareas se les conoce generalmente como tareas aperiódicas o no críticas.

Una tarea es aperiódica cuando sus características temporales no son predecibles o, si lo son, el grado de pesimismo de estas características es excesivamente alto. En estos casos no es posible garantizar un tiempo de respuesta acotado para estas tareas, pero es importante que este tiempo de respuesta sea el mejor posible.

Las tareas aperiódicas pueden ser gestionadas por servidores de carga aperiódica. Estos servidores son tareas periódicas o pseudo-periódicas, que coexisten con el resto de

tareas de la aplicación, y su función es la de atender las peticiones aperiódicas que se producen en el sistema. Estos servidores tienen garantizada una cierta capacidad de procesamiento a un determinado nivel de prioridad, y consiguen de esta forma que las tareas aperiódicas que atienden obtengan buenos tiempos de respuesta. Dado que estos servidores tienen un comportamiento temporal predecible, pueden ser analizados conjuntamente con el resto de tareas de la aplicación. Gracias a esto, es posible determinar si las tareas críticas cumplirán sus plazos a pesar de competir con tareas aperiódicas.

Existen varios tipos de servidores de tareas aperiódicas pero, de todos ellos, cabe destacar el denominado Servidor Esporádico [62], ya que sus cualidades han hecho que haya sido incorporado en los servicios POSIX para sistemas de tiempo real. No está disponible en el lenguaje Ada, pero parece razonable pensar que lo estará en un futuro próximo.

Existe, no obstante, un método alternativo de tratar las tareas aperiódicas. Este método no está basado en la incorporación de servidores, sino en políticas de planificación que permiten retrasar la ejecución de las tareas críticas sin que éstas pierdan sus plazos. De esta forma, las tareas aperiódicas pueden ejecutarse con preferencia sobre las tareas críticas mientras esto no comprometa sus plazos. Estas políticas son la extracción de holgura [20] y la asignación dual de prioridades [22], y aportan ciertas ventajas sobre el método basado en servidores. En primer lugar, permiten que las tareas aperiódicas obtengan mejores tiempos de respuesta. Por otro lado, su forma de operar es función de las características temporales de las tareas críticas, lo que evita tener que ajustar los parámetros del servidor de tareas aperiódicas para obtener los mejores resultados. Finalmente, facilitan que las tareas aperiódicas puedan ser ejecutadas bajo cualquier política de planificación. Tienen, no obstante, sus inconvenientes. No son capaces de garantizar una capacidad de procesamiento, por lo que en situaciones de mucha carga, el tiempo de respuesta de las tareas aperiódicas puede alargarse. Por otra parte, su implementación puede llegar a introducir una sobrecarga elevada, especialmente cuando se utiliza la extracción de holgura. En definitiva, estas políticas forman una familia distinta de soluciones para el mismo problema, con enfoques, ventajas e inconvenientes diferentes.

Estas políticas no han abandonado el campo estrictamente académico y, por tanto, no están disponibles para ser utilizadas en ningún entorno conocido de programación de sistemas de tiempo real. Es precisamente en este aspecto en el que se centran los trabajos realizados en esta tesis, en la que se pretende valorar cómo estas políticas pueden ser adaptadas e incorporadas en los entornos de referencia para la programación de sistemas de tiempo real, el lenguaje Ada y los servicios del estándar POSIX.

Este trabajo ha sido desarrollado en el marco de las investigaciones realizadas por los grupos de Tecnología Informática e Informática Industrial de la Universidad Politécnica de Valencia. Esta tesis es, en gran medida, fruto de los trabajos realizados en torno a la arquitectura de sistemas inteligentes de tiempo real ARTIS [25], en la que la política

basada en la extracción de holgura es utilizada para permitir la coexistencia de tareas con requisitos temporales estrictos y actividades inteligentes no críticas.

1.2. Objetivos

El objetivo fundamental de esta tesis es evaluar con detalle si las políticas basadas en la extracción de holgura y en la asignación dual de prioridades pueden ser aplicables en la práctica. Este objetivo general se particulariza en los siguientes objetivos específicos:

- Revisar en profundidad las políticas objeto de estudio e intentar corregir sus principales debilidades.
- Diseñar una interfaz para estas políticas en los dos principales entornos de programación de sistemas de tiempo real, el lenguaje Ada y los servicios POSIX.
- Implementar esta interfaz en un sistema operativo de tiempo real, para garantizar la viabilidad de las interfaces propuestas.
- Caracterizar y cuantificar el impacto de la implementación de estas políticas, comparándolo con la política estándar basada en prioridades fijas con desalojo.
- Desarrollar instrumentos de análisis y medida basados en los servicios de trazas POSIX, así como valorar la aplicación de estos servicios en el campo de los sistemas de tiempo real.

1.3. Estructura

El resto de esta tesis se estructura como sigue:

- En el capítulo 2 se revisan las políticas basadas en la extracción de holgura y en la asignación dual de prioridades. Como paso previo se ofrece un breve resumen de los principales resultados obtenidos relacionados con el paradigma de planificación basada en prioridades fijas con desalojo. Por otra parte, en este capítulo se proponen dos mejoras a la política basada en la extracción de holgura, relacionadas con la compartición de recursos entre tareas críticas y aperiódicas y la coexistencia con tareas esporádicas.
- En el capítulo 3 se diseñan nuevas interfaces POSIX y Ada para las políticas objeto de estudio. Estas interfaces son validadas mediante su implementación en el sistema operativo de tiempo real MaRTE OS.

- En el capítulo 4 se presentan diversos trabajos relacionados con los servicios de trazas POSIX. Con el fin de disponer de un instrumento de análisis y medida, estos servicios son implementados en el sistema MaRTE OS. Se diseña una interfaz POSIX/Ada para estos servicios y se propone una interfaz para obtener métricas temporales a partir de trazas POSIX.
- En el capítulo 5 se aborda el análisis de la eficiencia de las dos políticas estudiadas. Se caracteriza de forma analítica la sobrecarga que introduce su implementación. Esta sobrecarga es medida y se utiliza para valorar cómo estas políticas afectan a los tiempos de respuesta y al rendimiento global del sistema. Se analiza también el efecto de gestionar de forma diferida las interrupciones del temporizador hardware y se propone una optimización de la política basada en la extracción de holgura.
- Por último, en el capítulo 6, se resumen las principales conclusiones de esta tesis y se describen posibles líneas de trabajo futuro.

Capítulo 2

Planificación en Sistemas de Tiempo Real

En este capítulo se realiza un estudio detallado de los métodos y características de los algoritmos de planificación basados en la extracción dinámica de la holgura y la asignación dual de prioridades, en los cuales se centran los trabajos realizados en esta tesis.

Con el fin de enmarcar en su contexto estos algoritmos, en la sección 2.1 se revisan los aspectos más relevantes de las investigaciones realizadas en el paradigma de planificaciones fijas y su aplicación a la programación de sistemas de tiempo real. En las secciones 2.2 y 2.3 se describen, respectivamente, los dos algoritmos objeto de estudio y se analiza cómo en ellos se abordan cuestiones relevantes, tales como la compartición de recursos entre tareas críticas y aperiódicas, la coexistencia con tareas esporádicas, la variación entre instantes de activación, la posibilidad de utilizar plazos arbitrarios y la recuperación del tiempo de cómputo no producido. En la sección 2.4 se proponen nuevos métodos para corregir las deficiencias del algoritmo extractor de holgura dinámico relativas a la compartición de recursos entre tareas periódicas y aperiódicas, y la coexistencia con tareas esporádicas.

2.1. Sistemas de Tiempo Real Estricto

Los sistemas de tiempo real se definen normalmente como sistemas en los cuales su corrección depende no sólo del resultado lógico de la computación, sino también del instante en que dichos resultados se producen. Este segundo tipo de corrección se expresa como un conjunto de requisitos temporales que el sistema debe cumplir en tiempo de

ejecución. De acuerdo con estos requisitos, los sistemas de tiempo real pueden ser clasificados en dos categorías: sistemas de tiempo real estricto y no estricto. En los sistemas de tiempo real estricto, todos sus requisitos deben cumplirse bajo cualquier circunstancia ya que, de no ser así, pueden producirse consecuencias que pueden llegar a ser catastróficas. Por contra, en los sistemas de tiempo real no estricto, se permite que sus requisitos temporales puedan no cumplirse de forma ocasional, dado que lo que se produce con ello es una degradación de la respuesta del sistema, pero no un resultado catastrófico.

Los sistemas de tiempo real estricto son sistemas que normalmente interactúan con un proceso o entorno real. En este sentido, un sistema de tiempo real se ejecuta continuamente en un bucle en el cual se obtiene información del entorno, se calculan soluciones basadas en los datos de entrada y finalmente se actúa sobre el entorno aplicando dichas soluciones. Todas estas acciones deben ser realizadas cumpliendo los requisitos temporales que el entorno real impone. La variedad de procesos presentes en los entornos reales y el inherente paralelismo de dichos entornos conduce normalmente a diseñar e implementar los sistemas de tiempo real como un conjunto de tareas concurrentes. En esta clase de diseño, cada tarea resuelve un subconjunto del problema, y se le asigna un requisito temporal. Este requisito temporal suele expresarse como un plazo máximo de ejecución, plazo que debe ser cumplido cada vez que la tarea se ejecuta. De esta forma, la corrección temporal de todo el sistema se alcanza en la práctica cuando las tareas de la aplicación son planificadas y ejecutadas de tal forma que siempre cumplen con sus plazos de ejecución. Por esta razón, una gran parte del esfuerzo realizado en el campo de los sistemas de tiempo real se ha centrado en el desarrollo de paradigmas de planificación apropiados para conseguir la corrección temporal del sistema.

Los paradigmas de planificación para sistemas de tiempo real estricto se centran en garantizar los requisitos temporales del sistema a priori. Debe poder demostrarse que el sistema de tiempo real cumple sus requisitos temporales antes de que sea puesto en ejecución. Esta garantía se expresa normalmente en términos de viabilidad. La viabilidad del sistema corresponde a la certeza de que el sistema será capaz de cumplir con sus requisitos temporales. La viabilidad se basa en la predecibilidad del sistema, la cual se define como el conocimiento preciso de las acciones que el sistema realizará bajo cualquier situación una vez esté en ejecución.

Actualmente existen tres paradigmas de planificación fundamentales: planificación basada en ejecutivos cíclicos, planificación basada en prioridades fijas y planificación basada en prioridades dinámicas.

Cuando se utiliza la planificación basada en ejecutivos cíclicos [12, 43], se dispone la ejecución de las tareas que forman parte de la aplicación en un determinado orden conforme a un plan que se repite de forma cíclica. Este paradigma es muy sencillo de implementar y, por tanto, muy eficiente. La viabilidad del sistema está garantizada por la

propia construcción del plan. Su principal inconveniente es su rigidez, ya que cualquier modificación en la aplicación puede requerir diseñar de nuevo todo el plan. En cualquier caso, su uso está ampliamente aceptado y es utilizado en numerosas aplicaciones de tiempo real.

Con la planificación basada en prioridades fijas [11], a cada tarea de la aplicación de tiempo real se le asigna una prioridad. Esta prioridad se establece a priori, y por tanto es fija. Con este paradigma, los recursos del sistema son asignados a las tareas en función de su prioridad. De esta forma, los problemas de contención de recursos se resuelven de forma dinámica, evitándose así la necesidad de establecer un plan de ejecución a priori. La principal ventaja de este paradigma estriba en que el diseño de la aplicación de tiempo real es mucho más sencilla y natural. Por contra, se requiere la existencia de un sistema operativo multitarea que se encargue de aplicar esta política de planificación, lo que introduce sobrecarga de cómputo y, por tanto, menor eficiencia. La viabilidad del sistema se estudia a priori, aplicando para ello un análisis de planificabilidad para todas las tareas del sistema. Este análisis es función de las características temporales de las tareas y de su prioridad asignada. Este paradigma es también ampliamente utilizado. Prueba de ello es el gran número de sistemas operativos de tiempo real que lo implementan, así como ser el paradigma utilizado por los servicios de tiempo real definidos en los estándares POSIX y Ada.

La planificación basada en prioridades dinámicas [63] es similar al paradigma anterior, pero en este caso la asignación de la prioridad de cada tarea se realiza por parte del sistema de forma dinámica. Esta prioridad es función de la proximidad al vencimiento del plazo de la tarea. Al igual que en el caso anterior, la viabilidad del sistema se estudia aplicando un análisis de planificabilidad a priori. Su principal ventaja reside en que permite conseguir un aprovechamiento mayor del procesador respecto a los otros paradigmas citados. Por contra, su implementación es más compleja e ineficiente, especialmente en lo referente al acceso a recursos compartidos. Debido a esto, son muy pocos los sistemas operativos de tiempo real que lo implementan, y no forma parte tampoco de los servicios ofrecidos por POSIX y Ada.

El trabajo realizado en esta tesis se enmarca en el paradigma de planificación basada en prioridades fijas, por lo que a continuación se describen los aspectos más relevantes, relacionados con esta tesis, de la investigación realizada en torno a este paradigma:

- *Análisis de la planificabilidad*. Los resultados obtenidos en este campo permiten analizar si en un sistema de tiempo real todas las tareas son planificables, es decir, si ninguna de estas tareas perderá su plazo como consecuencia de las características temporales de la aplicación.

- *Interacción entre tareas*. La interacción entre tareas introduce el problema de la

inversión de prioridades. El bloqueo que causa la inversión de prioridades puede acotarse y analizarse gracias al desarrollo de diversos protocolos de acceso a los recursos compartidos, denominados protocolos de herencia de prioridades.

- *Inclusión de tareas aperiódicas*. La presencia de tareas aperiódicas en aplicaciones de tiempo real ha sido también un tópico ampliamente estudiado. Estas tareas no cumplen con las restricciones que exige el análisis de la planificabilidad del sistema, pero pueden ser importantes para la aplicación y, por tanto, es necesario ofrecer tiempos de respuesta adecuados para esta clase de tareas.

2.1.1. Análisis de la Planificabilidad

Bajo el paradigma de planificación basado en prioridades fijas, se asume que una aplicación de tiempo real está formada por un conjunto de n tareas. A cada tarea se le asigna una prioridad en el rango 1 a n [1]. Desde el punto de vista de notación, una tarea es nombrada como τ_i, siendo su prioridad igual a i.

Una tarea puede estar preparada o suspendida. En el primer caso está lista para ser ejecutada, mientras que en el segundo está esperando a que ocurra algún evento y, por tanto, no es candidata para ejecución. En cualquier instante, la tarea preparada que tiene la mayor prioridad es la tarea que está en ejecución. De esta forma, cuando una tarea pasa a estar preparada, ésta desaloja a la tarea que está actualmente en ejecución si su prioridad es mayor.

Cada tarea genera durante su ejecución un número ilimitado de instancias. Estas instancias pueden activarse de forma periódica (a intervalos regulares de tiempo) o de forma esporádica (por causa de eventos externos o de otras tareas). T_i es el periodo de una tarea periódica o el intervalo mínimo entre activaciones de una tarea esporádica. Cada instancia de una tarea se ejecuta durante un tiempo acotado por C_i (tiempo de ejecución en el peor caso), y debe completarse antes de su plazo D_i, relativo al instante de su activación.

Este modelo de tareas fue estudiado en el ya clásico trabajo de Liu y Layland [42]. En este trabajo se propuso un análisis suficiente que permite determinar si un conjunto de tareas es planificable, es decir, que ninguna tarea pierde su plazo. Las restricciones impuestas al modelo, así como el ser un análisis suficiente pero no necesario, hacen que este análisis sea poco aplicable en la práctica. No obstante, este trabajo sentó las bases para la mayoría de desarrollos que se realizaron años después.

Los resultados más relevantes obtenidos desde los trabajos de Liu y Layland son los análisis de planificabilidad basados en la cantidad de carga admisible y en los tiempos de respuesta. Estos análisis proponen condiciones suficientes y necesarias para que una

[1] En este apartado, 1 es la prioridad más alta y n la más baja del sistema

aplicación de tiempo real sea planificable, siempre que se cumplan las siguientes restricciones:

- El plazo de cada tarea es inferior o igual a su periodo ($D_i \leq T_i$).
- Las tareas son independientes entre sí.
- Las tareas comparten un instante crítico, es decir, en algún momento de la ejecución del sistema se activan de forma simultánea.
- Una instancia de una tarea no se suspende hasta que se completa.
- Las tareas son activadas tan pronto como se hacen preparadas.
- La sobrecarga del sistema es ignorada (se asume igual a 0).

El análisis basado en la cantidad de carga fue desarrollado por Lehoczky y otros en [38]. La cantidad de carga $W_i(t)$, en un instante dado t, generada por tareas con prioridad igual o superior a i se define como:

$$W_i(t) = \sum_{j=1}^{i} C_j \left\lceil \frac{t}{T_j} \right\rceil \tag{2.1}$$

Esta ecuación calcula la suma del tiempo de cómputo de todas las instancias de tareas con prioridad igual o superior a i ejecutadas en el intervalo $[0, t]$. Una tarea τ_i es planificable si y sólo si la siguiente condición es cierta:

$$min_{(0<t\leq D_i)} \left(\frac{W_i(t)}{t} \leq 1 \right) \tag{2.2}$$

Un análisis alternativo al anterior fue desarrollado por Harter en [31] y [32]. El mismo análisis fue publicado también de forma independiente por Joseph y Pandya en [33] y [34] y por Audsley y otros en [7]. A diferencia del caso anterior, en este análisis se propone la siguiente ecuación, la cual permite calcular el peor tiempo de respuesta para una tarea.

$$R_i = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{2.3}$$

De esta forma, una tarea es planificable si su tiempo de respuesta es inferior a su plazo:

$$R_i \leq D_i \tag{2.4}$$

Ambos análisis son, en esencia, equivalentes. El primer tipo de análisis es en cierta forma la continuación de los trabajos originales de Liu y Layland basados en la utilización del procesador. El segundo tipo de análisis aporta la ventaja de poder extenderse con mayor facilidad a sistemas más complejos, en los que ciertas de las restricciones expuestas anteriormente pueden ser eliminadas. Los trabajos más significativos en este campo han sido los siguientes:

- Cuando las tareas no comparten un instante crítico, los análisis anteriores siguen siendo suficientes pero dejan de ser necesarios. Esta situación puede producirse cuando las tareas se activan con un retardo relativo al inicio de la aplicación. Este tipo de sistemas puede analizarse de forma efectiva gracias a los trabajos realizados en [69].

- Otro tipo de sistemas analizados han sido aquellos en los que el plazo de una tarea puede ser superior a su periodo. En [70] se presenta un análisis suficiente y necesario para este tipo de sistemas.

- Por último, en [69] se amplia el análisis de tiempos de respuesta descrito anteriormente para contemplar la posible variación entre los instantes de activación de una tarea *(jitter)*.

Los análisis de planificabilidad suficientes y necesarios descritos no imponen ningún criterio para asignar prioridades a las tareas que forman parte de una aplicación de tiempo real. El criterio más utilizado, denominado *Rate Monotonic* o RM, consiste en asignar mayores prioridades a las tareas con mayor frecuencia de activación. Este es el criterio utilizado en el trabajo original de Liu y Layland, y su influencia ha sido tal que el conjunto de técnicas que permiten analizar la planificabilidad de sistemas de tiempo real suele conocerse como *Rate Monotonic Analysis*. Otro criterio utilizado, más general que el anterior, se denomina *Deadline Monotonic* o DM, propuesto por Leung y Whitehead en [41]. En este caso, la prioridad de una tarea es mayor cuanto menor sea su plazo D_i. De esta forma, RM es un caso particular de DM cuando los periodos y plazos de las tareas coinciden. En sus ámbitos de aplicación, ambos criterios son óptimos en el sentido de que si existe una asignación de prioridades capaz de hacer que un conjunto de tareas sea planificable, entonces la asignación de prioridades dictada por estos criterios también es viable. Esta propiedad no se mantiene cuando las tareas tienen plazos de ejecución arbitrarios o no comparten un instante crítico. En estos casos puede utilizarse el método de asignación de prioridades propuesto por Audsley en [8], método que sí es óptimo en estas situaciones.

2.1.2. Interacción entre Tareas

La restricción más inconveniente que fue impuesta en un primer momento a las aplicaciones de tiempo real fue la de independencia entre sus tareas. Cuando las tareas utilizan recursos de forma compartida, dejan de ser independientes entre sí. La compartición de recursos es común en aplicaciones de tiempo real, pero su uso origina problemas en este tipo de aplicaciones. Normalmente los recursos compartidos se utilizan de forma exclusiva, lo que puede hacer que una tarea interesada en un recurso deba esperar a que otra tarea de menor prioridad abandone dicho recurso. Es posible también que, mientras dura esta espera, se activen una o varias tareas de prioridad intermedia. Esto puede producir que la espera de la tarea más prioritaria se prolongue aun más. Esta situación, en la que una tarea puede verse bloqueada por tareas de menor prioridad, se conoce como inversión de prioridad [36].

Aunque no es posible eliminar por completo la inversión de prioridad, si es posible limitar la duración de este tipo de situaciones. Esta limitación es fundamental para poder aplicar los análisis de planificabilidad descritos en el apartado anterior. Para ello, pueden utilizarse los protocolos de herencia de prioridades.

El protocolo más básico utilizado se denomina Protocolo de Herencia de Prioridad (PIP) [61]. Cuando se utiliza este protocolo, si una tarea solicita un recurso que pertenece a otra tarea, esta última tarea hereda la prioridad de la primera, hasta que abandona el recurso compartido. Esta técnica limita el tiempo que una tarea puede ser bloqueada por tareas de menor prioridad. En [61] se demuestra que una tarea τ_i puede ser bloqueada a lo sumo n veces, siendo n el número de tareas de menor prioridad que τ_i y a lo sumo m veces, siendo m el número de semáforos que pueden causar bloqueo a τ_i. De esta forma, puede calcularse la duración máxima que puede durar un bloqueo, denominada normalmente B_i, e introducir este factor en el análisis de planificabilidad. Este protocolo tiene como desventajas, frente a los protocolos que se describen a continuación, que la duración del bloqueo máximo puede ser superior y no previene la aparición de interbloqueos. Este protocolo es ampliamente utilizado, ya que está presente en muchos sistemas operativos de tiempo real. Su relevancia es tal que forma parte de los servicios de tiempo real de POSIX.

Otro protocolo también muy utilizado se denomina comúnmente Protocolo de Techo de Prioridad Inmediato. Fue sugerido en el contexto del entorno de programación Mesa [36] y su aplicación en sistemas de tiempo real fue estudiada por Rajkumar en [57] bajo la denominación de *Ceiling Semaphore Protocol* (CSP). Forma parte de los servicios de tiempo real tanto de POSIX como de Ada. En este protocolo se asocia una prioridad a cada recurso compartido, denominándose techo a esta prioridad. Este techo es igual a la mayor prioridad de las tareas que utilizan dicho recurso. Cuando una tarea solicita un

recurso compartido, su prioridad se hace igual a la del techo del recurso. Este cambio de prioridad se mantiene hasta que la tarea abandona el recurso. La duración máxima del bloqueo B_i es igual a la duración de la sección crítica más larga que puede bloquear a τ_i. De esta forma, una tarea puede ser bloqueada a lo sumo una vez por tareas menos prioritarias. Con este protocolo se obtienen valores de B_i mucho menores que en el caso del protocolo PIP. Otras de sus ventajas son que permite una implementación mucho más eficiente y evita la aparición de interbloqueos, ya que se impone un orden estricto en la utilización de los recursos compartidos.

Otro protocolo ampliamente conocido es el protocolo denominado Protocolo de Techo de Prioridad o PCP [61]. Para cada recurso se establece un techo utilizando el mismo criterio que en el protocolo CSP. Una tarea puede adquirir un recurso si el recurso está libre, y su prioridad es estrictamente mayor que el techo global del sistema, el cual es igual al mayor de los techos de los recursos que están actualmente adquiridos. Una tarea que posee un recurso hereda la prioridad de cualquier tarea bloqueada por ella, bien de forma directa o indirecta. La duración máxima del bloqueo B_i es idéntica a la obtenida con el protocolo CSP, y también se evitan los interbloqueos. Los tiempos de respuesta que obtienen las tareas cuando utilizan este protocolo tienden a ser mejores en promedio que los obtenidos con el protocolo CSP, ya que con el protocolo PCP los cambios de prioridad sólo se producen cuando son estrictamente necesarios. Por contra, su implementación es mucho menos eficiente y genera más cambios de contexto. Dados estos inconvenientes, este protocolo no forma parte ni de POSIX ni de Ada, y por tanto, no es utilizado en la práctica.

Una vez determinada la duración máxima del bloqueo B_i que puede sufrir una tarea τ_i, esta duración puede aplicarse en las ecuaciones 2.1 y 2.3 para analizar si un conjunto de tareas que comparten recursos puede ser planificable. Este factor B_i se introduce en estas ecuaciones de la forma siguiente:

$$W_i(t) = B_i + \sum_{j=1}^{i} C_j \left\lceil \frac{t}{T_j} \right\rceil \tag{2.5}$$

$$R_i = C_i + B_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{2.6}$$

2.1.3. Inclusión de Tareas Aperiódicas

El concepto de tarea aperiódica resulta algo confuso en la teoría de sistemas de tiempo real. Originalmente una tarea era considerada aperiódica cuando su activación se producía a intervalos irregulares. Este tipo de tareas no podían ser contempladas por los primeros

análisis de planificación propuestos, y debido a ello, se realizaron numerosos trabajos orientados a poder considerar este tipo de tareas. Hoy en día se conoce que una tarea aperiódica puede ser tratada como si fuese una tarea periódica siempre que su tiempo de cómputo y la distancia mínima entre activaciones estén acotados. A este caso particular de tarea aperiódica se le denomina normalmente tarea esporádica. En la actualidad se considera una tarea como aperiódica cuando no puede ser tratada por las técnicas de análisis de planificabilidad, por alguno o varios de los siguientes motivos:

- Su tiempo de cómputo en el peor caso no está acotado.
- Su activación es irregular y no está acotada la distancia mínima entre activaciones.
- Su tiempo de cómputo en el peor caso está acotado, pero es muy superior a su tiempo de cómputo promedio, lo que provoca que esta tarea u otras de inferior prioridad no sean planificables.
- La distancia mínima entre activaciones está acotada pero es excesivamente pequeña, lo que produce, al igual que en el caso anterior, que esta tarea u otras de inferior prioridad no sean planificables.

Cuando en una aplicación de tiempo real coexisten tareas críticas (periódicas o esporádicas) y tareas aperiódicas, se pretende que las tareas críticas no pierdan sus plazos bajo ninguna circunstancia y que las tareas aperiódicas obtengan buenos tiempos de respuesta. Toda la investigación realizada en este campo se ha realizado en torno a este objetivo, y en este apartado se revisan los resultados más relevantes obtenidos.

El método más simple para planificar tareas aperiódicas consiste en asignar a estas tareas prioridades inferiores a las de las tareas críticas de la aplicación, método conocido como servicio en segundo plano. Si la carga de tareas periódicas es alta, la utilización que queda para el servicio en segundo plano es baja y las oportunidades para realizar este servicio son relativamente infrecuentes, lo que ocasiona tiempos de respuesta muy elevados para las tareas aperiódicas.

Un método alternativo consiste en la utilización de un servidor de tareas aperiódicas. Este servidor se ejecuta en un cierto nivel de prioridad, generalmente a la prioridad más alta del sistema, y su función consiste en ejecutar las peticiones realizadas por las tareas aperiódicas. Los servidores de tareas aperiódicas tienen una capacidad de procesamiento acotada, lo que permite integrarlos en el análisis de planificabilidad del sistema. Cuando un servidor agota su capacidad, las tareas aperiódicas deben esperar a que esta capacidad se recupere o a que todas las tareas críticas del sistema dejen de estar preparadas para ejecución. Cuando las tareas aperiódicas son ejecutadas por un servidor, pueden obtener mejores tiempo de respuesta respecto al servicio en segundo plano, ya que utilizan, mientras la capacidad lo permite, una prioridad que puede ser superior a la de varias de las

tareas críticas de la aplicación. En el marco de la planificación por prioridades fijas, se han propuesto varios tipos de servidores, los cuales se revisan a continuación.

El servidor más sencillo es el servidor de consulta, propuesto por Sha y otros en [60]. Este servidor se activa de forma periódica y su capacidad es utilizada para atender tareas aperiódicas. Cuando la capacidad se agota, el servidor se suspende. La capacidad del servidor se repone cuando éste es activado de nuevo en el siguiente periodo. Este método mejora los tiempos de respuesta de las tareas aperiódicas. Su principal inconveniente reside en que, si no hay tareas aperiódicas preparadas cuando el servidor está activo, éste se suspende hasta el próximo periodo, desperdiciando por tanto su capacidad.

Los algoritmos de intercambio de prioridades (PE) y el servidor diferido (DS), introducidos por Lehoczky y otros en [37], superan el inconveniente del servidor de consulta. Al igual que en el caso anterior, los servidores PE y DS crean una tarea periódica (usualmente de prioridad alta) para servir las peticiones aperiódicas. Sin embargo, a diferencia del servidor de consulta, estos servidores preservan el tiempo de ejecución asignado al servicio aperiódico si, al invocar la tarea servidora, no hay peticiones aperiódicas pendientes. Estos servidores pueden producir mejores tiempos medios de respuesta para las peticiones aperiódicas debido a su capacidad para proporcionarles servicio inmediato. Los servidores PE y DS difieren en la forma en que preservan su capacidad.

El servidor DS mantiene su capacidad disponible durante el periodo del servidor. Así, las peticiones aperiódicas pueden servirse a la prioridad del servidor en cualquier momento, mientras la capacidad para el periodo actual no haya sido agotada. Al comienzo del siguiente periodo se repone la capacidad completa del servidor. A diferencia del servidor DS, el servidor PE preserva la capacidad, intercambiándola por el tiempo de ejecución de una tarea menos prioritaria. El servidor DS es más sencillo que implementar, pero ofrece peores tiempos de respuesta que el servidor PE [37].

Un nuevo tipo de servidor, denominado Servidor Esporádico (SS) fue introducido por Sprunt en [62], con el fin de ofrecer las ventajas de los servidores anteriores. El funcionamiento de este servidor es el siguiente. La capacidad del servidor SS se repone T_s unidades de tiempo tras el instante en que el nivel de prioridad del servidor se hace activo, siendo T_s el periodo del servidor. En particular, se reponen c unidades de tiempo, que corresponden con el tiempo consumido por las tareas aperiódicas atendidas por el servidor. La implementación del servidor SS es relativamente sencilla, y ofrece mejores tiempos de respuesta que el servidor PE [62], dado que permite obtener una utilización máxima mayor, aunque esta propiedad es cuestionada en [13].

Las cualidades del servidor esporádico han hecho que este método haya sido recientemente adoptado por POSIX [52] como la política de planificación denominada `SCHED_SPORADIC`. El enfoque dado en POSIX a esta política es diferente al enfoque original. Las diferencias principales son las siguientes:

- En POSIX no existe un servidor encargado de atender peticiones provenientes de distintas tareas aperiódicas, sino que cada tarea aperiódica es en sí misma un servidor esporádico. De esta forma, cada tarea aperiódica de la aplicación que utiliza esta política tiene asignados un periodo y una capacidad.

- El número de operaciones de reposición pendientes está limitado, con el fin de preservar los recursos del sistema así como de acotar la inversión de prioridad que puede ocasionar la operación de reposición.

- Las reglas de esta política se enuncian en función de los instantes en los que una tarea accede o abandona la cola conceptual de tareas preparadas y la definición de esta política es algo distinta a la propuesta originalmente.

Un método alternativo de planificar tareas aperiódicas está basado en los algoritmos extractores de holgura. Estos algoritmos permiten conocer, en tiempo de ejecución, el tiempo de procesamiento disponible (holgura) que permite que una tarea crítica se ejecute sin perder su plazo. Si existe holgura para todas las tareas críticas, es posible utilizar este tiempo para ejecutar tareas aperiódicas.

El primero de los algoritmos extractores de holgura fue propuesto por Lehoczky y Ramos-Thuel en [39]. Este algoritmo consiste en calcular a priori la holgura que puede obtenerse en un hiperperiodo para cada instancia de una tarea. Estos valores se almacenan en una tabla y son consultados en tiempo de ejecución para determinar si es posible ejecutar tareas aperiódicas. Este algoritmo es aplicable sólo si todas las tareas críticas son estrictamente periódicas y no comparten recursos entre sí. Otro de sus inconvenientes reside en el tamaño de la tabla, ya que su tamaño depende del número de instancias que se ejecuten en un hiperperiodo, y éste puede ser relativamente grande en función de las características de las tareas.

Muchas de las limitaciones del algoritmo extractor de holgura son eliminadas por la versión dinámica desarrollada por Davis y otros en [20]. En este caso, la holgura no se calcula a priori y se almacena en una tabla, sino que se calcula de forma dinámica. Este algoritmo permite tareas esporádicas y sincronización entre tareas críticas. Su principal inconveniente reside en que el coste para calcular la holgura es muy elevado, ya que es función del número de tareas y de cómo estas tareas intercalan sus ejecuciones. En [21] se propone un algoritmo aproximado cuyo coste es mucho menor, lo que posibilita su aplicación en la práctica.

Finalmente, es posible utilizar también la asignación dual de prioridades propuesto por Davis en [22]. Cada tarea crítica del sistema utiliza dos prioridades, una baja y otra alta. Para cada tarea se determina su tiempo de promoción, que es función del tiempo de respuesta de la tarea en el peor caso. Cuando una tarea se activa utiliza su prioridad baja

y, una vez se alcanza el tiempo de promoción, su prioridad pasa a ser la alta. Las tareas aperiódicas utilizan una prioridad media, por lo que se ejecutan con preferencia cuando no hay ninguna tarea crítica promocionada. Este algoritmo es aplicable bajo un gran número de supuestos posibles y su implementación es sencilla y eficiente.

Los algoritmos extractor de holgura y asignación dual de prioridades aportan ciertas ventajas sobre el método basado en servidores. Permiten que las tareas aperiódicas obtengan mejores tiempos medios de respuesta [20, 26]. Por otro lado, su forma de operar es función de las características temporales de las tareas críticas, lo que evita tener que ajustar los parámetros del servidor de tareas aperiódicas para obtener los mejores resultados. Finalmente facilitan que las tareas aperiódicas puedan ser ejecutadas bajo cualquier política de planificación. Tienen, no obstante, sus inconvenientes. No son capaces de garantizar una capacidad de procesamiento, por lo que en situaciones de mucha carga, el tiempo de respuesta de las tareas aperiódicas puede alargase. Por otra parte, su implementación puede llegar a introducir una sobrecarga elevada, especialmente cuando se utiliza la extracción de holgura.

Ambos algoritmos han sido utilizados para implementar modelos de tareas en los que intervienen componentes opcionales [9, 10]. Un componente es opcional cuando su ejecución no es estrictamente necesaria, pero puede aportar mayor calidad a la respuesta del sistema. El algoritmo extractor de holgura ha sido utilizado en el sistema ARTIS [24] y en *Flexible RT-Linux* [67] para ofrecer un modelo de tareas en el que cada tarea está formado por partes obligatorias y opcionales. Las partes obligatorias tienen garantizados sus tiempos de respuesta, mientras que las partes opcionales sólo se ejecutan si hay capacidad de procesamiento disponible. Por otra parte, el algoritmo de asignación dual de prioridades ha sido utilizado en el sistema Jorvik [14] para implementar un modelo de tareas similar.

Dado que esta tesis se centra en el estudio de estos dos algoritmos, ambos son descritos con mayor nivel de detalle en las siguientes secciones.

Con la excepción del servidor esporádico, las políticas citadas en este apartado no forman parte de los estándares POSIX y Ada. No obstante, se han realizado diversas propuestas para implementarlas en el nivel de aplicación, utilizando los servicios definidos por estos estándares. En [27] y [29] se implementa la política del servidor esporádico en el lenguaje Ada. La implementación en Ada de la política basada en la extracción de holgura se describe en [23]. Otro ejemplo es la implementación de la política basada en la asignación dual de prioridades [18, 14].

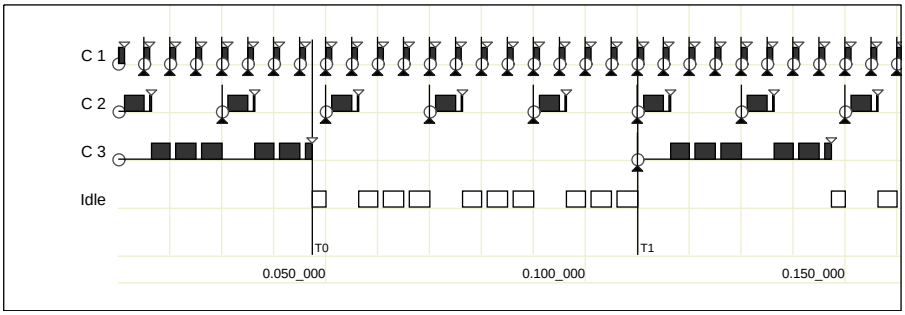
La implementación de estas políticas a nivel de aplicación presenta algunos inconvenientes. En algunos casos es necesario imponer restricciones adicionales para poder implementar la política, tal como sucede con los casos del servidor esporádico y la extracción de holgura. Por otro lado, la sobrecarga que generan estas soluciones es generalmente muy elevada. Finalmente, se requiere incluir en el código de las tareas de la aplicación

llamadas explícitas para que el planificador implementado a nivel de aplicación se ejecute en los instantes oportunos. Esto dificulta la legibilidad del código y hace este código susceptible de incorporar errores. Estos inconvenientes determinan que el único método realmente adecuado para implementar estas políticas sea a nivel del sistema, ofreciendo una interfaz de acceso adecuada para el nivel de aplicación. Esta ha sido, por ejemplo, la solución adoptada para la política del servidor esporádico en POSIX.

2.2. El Algoritmo Extractor de Holgura Dinámico

El algoritmo Extractor de Holgura Dinámico (DSSA), propuesto por Davis en [20], permite mantener la garantía de los tiempos de respuesta de las tareas críticas a la vez que mejora los tiempos de respuesta de las tareas aperiódicas.

Este algoritmo se fundamenta en el concepto de holgura. La holgura se define como la cantidad de tiempo de procesamiento que puede ser arrebatada a una tarea crítica sin que ésta pierda su plazo. En la figura 2.1 se ilustra este concepto [2]. En ella puede observarse como, desde la finalización de la tarea C3 ($t0$) hasta su plazo ($t1$) se suceden varios intervalos de tiempo en los cuales el procesador está ocioso.



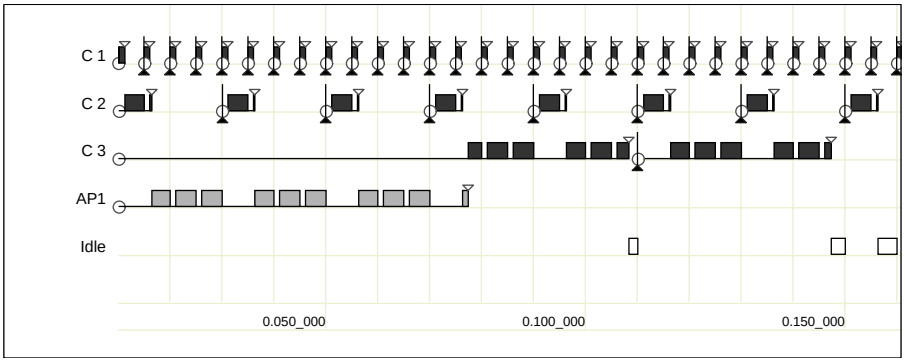
$C1 : (T = D = 5ms, C = 1ms)$ $C2 : (T = D = 20ms, C = 4ms)$
$C3 : (T = D = 100ms, C = 20ms)$

Figura 2.1: Concepto de Holgura I

La suma de esta sucesión de intervalos de tiempo es lo que se denomina holgura. Esta holgura puede destinarse a atender carga aperiódica, la cual se ejecutará con preferencia sobre la carga crítica mientras exista holgura disponible. En la figura 2.2 puede observar-

[2] Este cronograma, al igual que el resto de cronogramas que aparecen en esta tesis, ha sido generado con la aplicación Quivi. El significado de los símbolos utilizados puede consultarse en la figura 4.1

se como la tarea aperiódica AP1 puede ser ejecutada con preferencia sobre la tarea C3 apropiándose de su holgura sin provocar la pérdida de su plazo.



$C1 : (T = D = 5ms, C = 1ms)\ C2 : (T = D = 20ms, C = 4ms)$
$C3 : (T = D = 100ms, C = 20ms)$

Figura 2.2: Concepto de Holgura II

El algoritmo DSSA permite calcular la holgura de forma exacta en tiempo de ejecución. Para ello se supone que se pueden obtener de cada tarea τ_i los siguientes datos [3]:

$x_i(t)$ instante en que se producirá la siguiente activación de τ_i.

$d_i(t)$ siguiente instante de tiempo en que se producirá el vencimiento del plazo de τ_i (Si la instancia actual ha finalizado, corresponderá al vencimiento de la siguiente instancia)

$c_i(t)$ tiempo de ejecución restante de la instancia actual de τ_i. (Si la instancia actual ha finalizado, entonces será igual a cero).

El objetivo será conocer, para un cierto instante t, lo que denotaremos por $S_i^{max}(t)$, que será la cantidad máxima de tiempo que se puede arrebatar a las tareas pertenecientes al nivel de prioridad i-ésimo en el intervalo $[t, d_i(t))$, sin que ello afecte al cumplimiento de sus plazos máximos de ejecución. Básicamente el método considera este intervalo como constituido por una sucesión de periodos pertenecientes a dos tipos:

- Periodo ocupado de nivel i, que es un intervalo de tiempo continuo en el una o más tareas con prioridad igual o superior a i están preparadas para ejecución.

[3] En la formulación original estos datos son relativos al instante t. Por contra, en este caso se definen como instantes absolutos de tiempo, con el fin de simplificar la notación empleada en esta tesis.

- Periodo ocioso de nivel i, en el cual no hay ejecutándose tareas de prioridad igual o mayor a i.

El cálculo de $S_i^{max}(t)$ se basa en la identificación de esa sucesión de periodos, y así, la suma de periodos ociosos de nivel i en el intervalo de estudio nos dará el valor deseado. El cálculo de la holgura exacta disponible se realiza por un algoritmo de complejidad pseudo-polinomial $O(mn^2)$, donde m es el número de iteraciones, que depende de los periodos y plazos de las tareas, y n es el número de tareas.

Las características de este algoritmo lo convierten en óptimo, condición que se establece en el siguiente teorema [20].

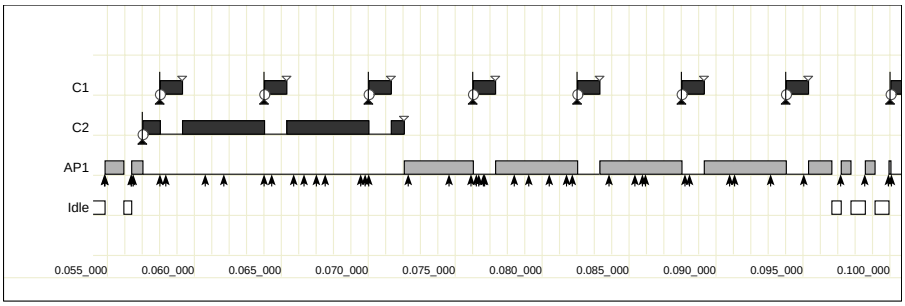
> "Para cualquier conjunto de tareas críticas planificadas mediante un esquema de prioridades fijas con desalojo, y cualquier sucesión de tareas aperiódicas procesadas en orden FIFO, el algoritmo dinámico de extracción de holgura minimiza el tiempo de respuesta de cada tarea aperiódica, entre todos los algoritmos que garantizan todos los plazos máximos de ejecución de las tareas críticas".

El método empleado para decidir cuándo ejecutar trabajo aperiódico es el siguiente. Siempre que haya trabajo aperiódico pendiente, se utiliza el algoritmo para encontrar la holgura disponible en los niveles de prioridad k e inferiores, siendo k el nivel de prioridad más alto de las tareas críticas listas para ejecución. Sólo se podrá ejecutar el trabajo aperiódico al nivel de prioridad k, mientras exista holgura disponible al nivel de prioridad k y todos los niveles inferiores, es decir, si se cumple la siguiente relación:

$$\min_{\forall i \in lp(k)} (S_i^{max}(t)) > 0 \tag{2.7}$$

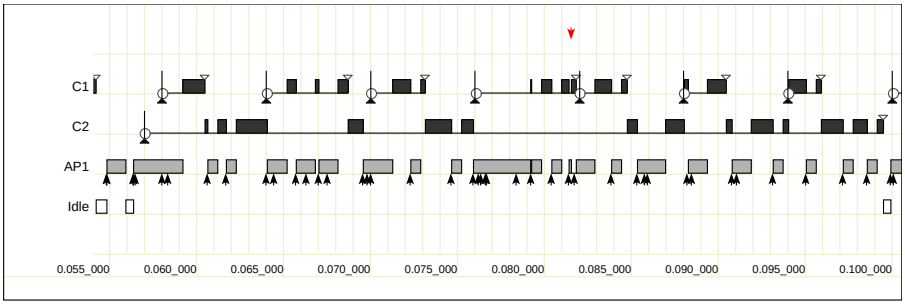
El funcionamiento de este algoritmo se ilustra en las figuras 2.3 y 2.4. En ambas figuras se muestra la ejecución de dos tareas periódicas C1 y C2 y una tarea aperiódica AP1. En la primera figura no se utiliza el algoritmo DSSA, mientras que sí se utiliza en la segunda figura. En estas figuras el símbolo $\uparrow$ representa una activación de la tarea AP1 y el símbolo $\downarrow$ representa el instante en que se agota la holgura del sistema. En la figura 2.4 puede observarse como la tarea aperiódica es atendida inmediatamente mientras hay holgura en el sistema, y por tanto obtiene un mejor tiempo de respuesta promedio.

A continuación se detallan algunas cuestiones importantes relativas a la aplicación de este algoritmo.



$C1 : (T = D = 6ms, C = 1{,}2ms)\ C2 : (T = D = 47ms, C = 11ms)$

Figura 2.3: Servicio Aperiódico de Baja Prioridad



$C1 : (T = D = 6ms, C = 1{,}2ms)\ C2 : (T = D = 47ms, C = 11ms)$

Figura 2.4: Servicio Aperiódico con el Algoritmo DSSA

Cálculo Aproximado de la Holgura

Dado que el coste temporal del algoritmo que calcula la holgura exacta es muy elevado, en [21] se propone calcular una cota inferior de la holgura de una tarea crítica, mediante un algoritmo de menor coste temporal que el exacto.

Para ello se calcula la interferencia que la tarea τ_i puede recibir en el intervalo $[t, d_i(t))$ por parte de cualquier tarea τ_j con prioridad igual o superior. Entonces, una cota inferior de la holgura puede ser la duración del intervalo $[t, d_i(t))$ menos la interferencia calculada.

El cálculo de la interferencia producida por una cierta tarea τ_j, y que denotaremos por $I_j(t, e)$, será la suma de:

- El cómputo de la tarea τ_j pendiente en t.
- El cómputo de ejecuciones completas que la tarea τ_j puede ejecutar en el intervalo considerado, si no tuviera interferencia.

- El cómputo (total o parcial) de una posible última activación de la tarea τ_j al final del intervalo.

Es decir

$$I_j(t,e) = c_j(t) + f_j(t,e)C_j + min(C_j, (e - x_j(t) - f_j(t,e)T_j)_0) \tag{2.8}$$

donde

- e es el instante en que termina el intervalo de estudio, es decir $d_i(t)$.
- $f_j(t,e)$ es el número de instancias completas de la tarea τ_j en $[t,e)$, que podemos calcular así:

$$f_j(t,e) = \left\lfloor \frac{(e - x_j(t))_0}{T_j} \right\rfloor \tag{2.9}$$

A partir de dicha interferencia, el cálculo de una cota inferior de la holgura, se podrá obtener mediante la siguiente ecuación:

$$S_i(t) = \left(e - t - \sum_{\forall j \in hp(i) \cup i} I_j(t,e) \right)_0 \tag{2.10}$$

Esta aproximación es pesimista, pues considera que toda la ejecución de cada tarea de prioridad superior tiene lugar en el intervalo. Por ejemplo, si incluimos en $I_j(t,e)$ la ejecución de la invocación final de la tarea τ_j y resulta que, debido a otra tarea τ_k más prioritaria que τ_j, dicha ejecución final se retrasa hasta fuera del intervalo $[t, d_i(t))$, entonces se está acumulando a la interferencia total un periodo ocupado que no se producirá en ese intervalo, disminuyendo consecuentemente la holgura que se obtendrá. Sin embargo, de esta forma se ha reducido el coste de aplicar el algoritmo de $O(mn)$ a $O(n)$.

Para reducir el grado de pesimismo inherente a esta aproximación, se introduce el concepto de plazo máximo de ejecución efectivo e. e es un instante del intervalo $[t, d_i(t))$ después del cual no existe ningún periodo ocioso de nivel i. El cálculo de e propuesto en [20] y corregido posteriormente por [25] es el siguiente:

$$\begin{array}{l} e = d_i(t) \\ \textbf{loop } \forall j \in hp(i) \\ \quad \textbf{if } f_j(t,e) > 0 \textbf{ and } x_j(t) + f_j(t,e)T_j + C_j \geq e \textbf{ then} \\ \quad\quad e = x_j(t) + f_j(t)T_j \\ \quad \textbf{end if} \\ \textbf{end loop} \end{array} \tag{2.11}$$

Es decir, si la última activación de la tarea τ_j junto con su tiempo de cómputo C_j sobrepasa el final del intervalo, entonces dicha activación es un plazo máximo de ejecución efectivo en la ejecución de la tarea τ_i.

Derivación de la Holgura

Es posible derivar la holgura en un instante t' a partir de la holgura en un instante t. Si en un intervalo $[t, t')$ el procesador ha atendido tareas aperiódicas o ha estado ocioso, la holgura se consume en todos los niveles de prioridad.

$$\forall j : S_j^{max}(t') = S_j^{max}(t) - (t' - t) \tag{2.12}$$

Por contra, si el procesador ha atendido en dicho intervalo una tarea crítica τ_i, entonces la holgura se consume en todos los niveles de prioridad superior.

$$\forall j \in hp(i) : S_j^{max}(t') = S_j^{max}(t) - (t' - t) \tag{2.13}$$

De esta forma, la holgura puede ser calculada sólo cuando la tarea τ_i finaliza, mientras que en cada cambio de contexto la holgura de todos los niveles es actualizada utilizando las ecuaciones anteriores, con lo que se reduce notablemente la sobrecarga que este algoritmo introduce en el sistema.

Tareas Esporádicas

El algoritmo DSSA, cuando se utiliza conjuntamente con el método de derivación de holgura, plantea algunos inconvenientes si existen tareas esporádicas en el sistema. Cuando una tarea esporádica se activa con menor frecuencia de lo posible, se produce un aumento de la holgura, ya que se reduce la interferencia que esta tarea causa en tareas de prioridad inferior. Este aumento de la holgura no puede ser detectado por el método de derivación de la holgura. De hecho, en estos casos es necesario calcular la holgura en cada tick de reloj para conocer su valor exacto, método que es inaplicable en la práctica. Para aproximar mejor la holgura en presencia de tareas esporádicas, en [21] se proponen dos variantes del algoritmo extractor de holgura, las cuales combinan la técnica estática de cálculo de la holgura con el cálculo dinámico de la holgura de forma periódica. En estas variantes se utiliza el cálculo estático de la holgura cada vez que una tarea finaliza y se introduce una tarea que recalcula la holgura para todos los niveles de prioridad utilizando el método dinámico. Estas variantes difieren en que la tarea que calcula la holgura se activa de forma periódica en un caso o cuando existe holgura en el sistema en el otro.

Recursos Compartidos

Asumiendo que las tareas que se ejecutan en tiempo de holgura no utilizan los mismos recursos que las tareas críticas, es sencillo modificar el criterio para admitir tareas en tiempo de holgura, siempre que se utilice el protocolo PCP o el protocolo CSP. Basta para ello incluir el factor de bloqueo B_i en el criterio de decisión, quedando éste como sigue.

$$\min_{\forall i \in lp(k)} (S_i^{max}(t) - B_i) > 0 \tag{2.14}$$

Es posible mejorar este criterio de decisión, sustituyendo el factor de bloqueo B_i por el factor de bloqueo actual b_i, el cual no está basado en el peor caso (como B_i), sino en el análisis de la situación en la que se encuentra actualmente el sistema. Este método consiste en detectar, cuando se calcula la holgura para una tarea τ_i, si es posible que dicha tarea pueda sufrir bloqueo. Este análisis se realiza en función del techo actual del sistema y de las secciones críticas que tareas menos prioritarias puedan estar ejecutando. Si se detecta que no puede haber bloqueo, b_i se hace igual a cero, mientras que si el bloqueo es posible, b_i se hace igual al tiempo restante de la sección crítica que puede bloquear a τ_i. El procedimiento detallado para calcular b_i para los protocolos PCP y CSP puede consultarse en [20] y [65] respectivamente.

Para permitir que las tareas que se ejecutan en tiempo de holgura compartan recursos con las tareas críticas, es necesario modificar el comportamiento de las operaciones de cierre y apertura de semáforos. Cuando una tarea aperiódica solicita cerrar un semáforo, es necesario comprobar no sólo que esté abierto, sino que exista suficiente tiempo para que dicho semáforo sea abierto antes de que la holgura se agote. De esta forma no se compromete el plazo de ninguna de las tareas críticas, ya que en el factor de bloqueo B_i no se consideran las secciones críticas de las tareas aperiódicas. Una tarea aperiódica puede cerrar un semáforo abierto sólo si

$$\min_{\forall i \in lp(j)} (S_i^{max}(t) - B_i) > c \tag{2.15}$$

siendo c la duración de la sección crítica y j el techo del semáforo que se desea cerrar.

Si se cumple esta condición, la holgura en todos los niveles iguales o inferiores a j debe ser decrementada en c, con el fin de garantizar que la tarea aperiódica finalizará su sección crítica a tiempo.

Recuperación del Tiempo de Cómputo no Utilizado

En el algoritmo exacto visto hasta el momento, se considera a efectos de cálculo que las tareas consumen siempre su tiempo de ejecución en el peor caso. No obstante, es muy

probable que todo este tiempo no sea utilizado.

Este tiempo no utilizado puede identificarse en el instante en que una tarea τ_i completa su ejecución en menor tiempo que su peor tiempo de ejecución. En este caso, no es necesario recalcular la holgura existente, sino que basta con aumentar la holgura en todos los niveles de prioridad inferior a la prioridad de la tarea τ_i.

Variación entre Instantes de Activación *(jitter)*

Si los instantes de llegada y activación de una tarea τ_i pueden diferir, se dice que dicha tarea exhibe variación entre instantes de activación *(jitter)*. Para que el cálculo de la holgura para una tarea τ_i sea correcto, es necesario tener en cuenta esta variación en el próximo instante de activación $x_j(t)$ de las tareas τ_j más prioritarias que τ_i. Para ello, el próximo instante de activación de una tarea debe calcularse según la siguiente ecuación:

$$x_j(t) = x'_j(t) + T_j - J_j \tag{2.16}$$

donde $x'_j(t)$ es el último instante de activación detectado por el sistema y J_j es la máxima variación entre instantes de activación.

Plazos Arbitrarios

Cuando las tareas tienen plazos arbitrarios es posible aplicar el algoritmo DSSA. En [20] se describe un algoritmo que permite calcular la holgura para la q-ésima activación de una tarea. Este algoritmo converge si la utilización del procesador es inferior al 100 %. El coste de este algoritmo lo hace inaplicable en la práctica y no puede ser utilizado conjuntamente con el cálculo aproximado de la holgura.

2.3. El Algoritmo de Asignación Dual de Prioridades

El algoritmo de Asignación Dual de Prioridades (DP), propuesto por Davis en [22], permite mantener la garantía de los tiempos de respuesta de las tareas críticas a la vez que mejora los tiempos de respuesta de las tareas aperiódicas. Este algoritmo está basado en los trabajos previos de Harbour, Klein y Lehoczky [28] y Burns y Wellings [15], en los que se investiga cómo la asignación de múltiples prioridades a tareas puede ser usada para aumentar el nivel de utilización del sistema. Este algoritmo destaca por proporcionar buenos tiempos de respuesta a tareas aperiódicas, por poder ser implementado de forma bastante eficiente, incluso en el nivel de aplicación, tal como se ilustra en [18], donde el algoritmo dual es implementado utilizando servicios POSIX y el lenguaje Ada y por ser

compatible con aspectos importantes tales como recursos compartidos entre tareas críticas y aperiódicas, tareas esporádicas, tareas con plazos arbitrarios, variación entre instantes de activación y tiempos de ejecución estocásticos.

En el modelo de asignación dual de prioridades se asume que existe un único rango de prioridades dividido en tres bandas: alta, media y baja. Cualquier nivel de prioridad de la banda superior se considera mayor que cualquier nivel de las bandas medias y baja. Las tareas críticas utilizan dos prioridades, una de la banda alta y otra de la baja. A las tareas aperiódicas se les asignan prioridades en la banda media. La única restricción para la asignación de prioridades es que se realice según una ordenación que garantice que el conjunto de las tareas críticas sea planificable teniendo en cuenta sus prioridades de la banda alta.

Cada vez que una tarea crítica τ_i se activa, lo hace con su prioridad del rango de la banda baja. Puede ser expulsada por cualquier tarea, crítica o aperiódica, más prioritaria de cualquier banda. Transcurrido un cierto tiempo, denominado tiempo de promoción, la prioridad de la tarea crítica τ_i se eleva a su prioridad de la banda alta. A partir de este momento sólo podrá ser expulsada por las tareas críticas que se hayan promocionado y tengan una prioridad superior en la banda alta.

El cálculo de los tiempos de promoción está basado en la aplicación directa del test de planificabilidad que calcula los tiempos de respuesta de la primera activación de cada tarea considerando un instante crítico inicial. Para ello, en [22] se demuestra que cuando se aplica el algoritmo DP, el peor tiempo de respuesta R_i para una tarea crítica τ_i es

$$R_i = w_i + Y_i \tag{2.17}$$

$$w_i^{m+1} = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i^m}{T_j} \right\rceil C_j \tag{2.18}$$

siendo Y_i el tiempo de promoción de τ_i. El instante crítico para τ_i sucede cuando ella y todas las tareas más prioritarias que τ_i son promocionadas de forma simultanea.

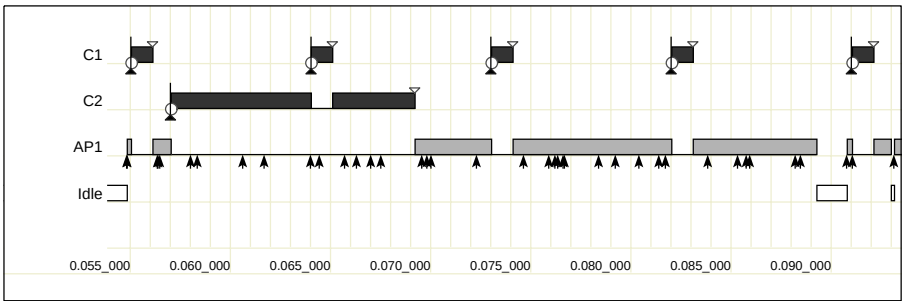
A partir de la ecuación anterior, puede determinarse cual es el máximo valor de Y_i que permite que el conjunto de tareas críticas sea planificable.

$$Y_i^{max} = D_i - R_i^{Y=0} \tag{2.19}$$

siendo $R_i^{Y=0}$ el peor tiempo de respuesta para τ_i cuando los tiempos de promoción son igual a cero para todas las tareas, es decir, el peor tiempo de respuesta de τ_i cuando no se aplica el algoritmo DP.

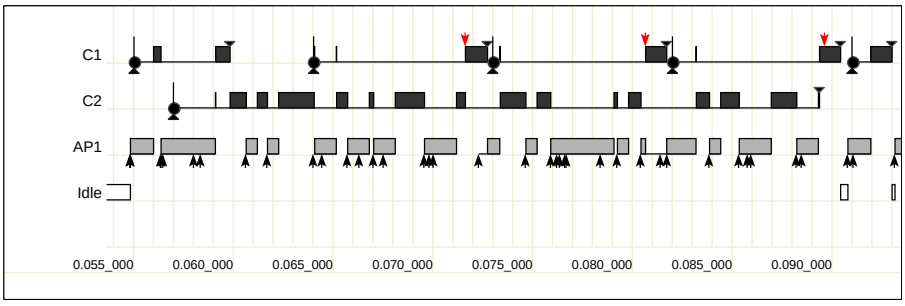
En el algoritmo DP se propone utilizar Y_i^{max} como tiempo de promoción de cada tarea, con el fin de minimizar los tiempos de respuesta de las tareas aperiódicas, aunque cualquier valor de Y_i inferior a Y_i^{max} es igualmente válido para que el conjunto de tareas críticas sea planificable.

El funcionamiento de este algoritmo se ilustra en las figuras 2.5 y 2.6. En ambas figuras se muestra la ejecución de dos tareas periódicas C1 y C2 y una tarea aperiódica AP1. En la primera figura no se utiliza el algoritmo DP, mientras que sí se utiliza en la segunda figura. En estas figuras el símbolo ↑ representa una activación de la tarea AP1 y el símbolo ↓ representa un instante de promoción.



$C1 : (T = D = 9ms, C = 1ms)\ C2 : (T = D = 47ms, C = 11ms)$

Figura 2.5: Servicio Aperiódico de Baja Prioridad



$C1 : (T = D = 9ms, C = 1ms)\ C2 : (T = D = 47ms, C = 11ms)$

Figura 2.6: Servicio Aperiódico con el Algoritmo DP

A continuación se analizan algunos aspectos relevantes de este algoritmo.

Recursos Compartidos entre Tareas Críticas y Aperiódicas

El algoritmo DP permite que las tareas críticas puedan utilizar semáforos para compartir recursos si se utiliza el protocolo de herencia de prioridades CSP. Esta propiedad se extiende incluso a semáforos compartidos por tareas críticas y aperiódicas.

El techo de cada semáforo se obtiene considerando las prioridades de las bandas media y alta. De esta forma, si una tarea crítica utiliza un semáforo, su techo pertenecerá a la banda alta de prioridades, tanto si es utilizado sólo por otras tareas críticas o por tareas críticas y aperiódicas.

Cuando una tarea crítica que se está ejecutando en la banda baja cierra un semáforo, cambia su prioridad a la banda alta. De forma similar, cuando una tarea aperiódica cierra un semáforo utilizado por alguna tarea crítica, su prioridad también cambia a la banda alta. De esta forma sólo puede existir una sección crítica en ejecución cuando no hay tareas críticas promocionadas, lo que garantiza que se mantengan todas las propiedades del protocolo CSP.

Dado que las propiedades del protocolo CSP se mantienen, basta con considerar tanto a las tareas críticas como a las aperiódicas en el cálculo de los factores de bloqueo B_i, para poder analizar si el conjunto de tareas críticas es planificable.

Tareas Esporádicas

Cuando una tarea esporádica no se activa a su máxima frecuencia se genera una cierta cantidad de tiempo sobrante en el sistema. Tal como opera el algoritmo DP, este tiempo sobrante se identifica de forma continua. Cuando una tarea esporádica se activa t unidades de tiempo más tarde de lo posible, su próximo instante de activación se retrasa también en t unidades. Este retraso en el instante de promoción redunda en beneficio de las tareas aperiódicas, las cuales pueden utilizar el tiempo sobrante generado por la tarea esporádica.

Utilización del Tiempo Recuperable

El tiempo recuperable puede ser fácilmente identificado en dos situaciones, cuando una tarea τ_i se ejecuta parcialmente antes de su instante de promoción y cuando una tarea τ_i consume menos tiempo del máximo posible.

En el primer caso, el instante de promoción puede retrasarse en función del tiempo de cómputo consumido por la tarea. Siendo $y_i(t)$ el próximo instante de promoción de τ_i, si τ_i se ejecuta en el intervalo $[t, t')$, su instante de promoción puede ser aplazado hasta $y_i(t')$, utilizando la siguiente ecuación:

$$y_i(t') = y_i(t) + (t' - t) \tag{2.20}$$

En el segundo caso, los instantes de promoción de las tareas menos prioritarias que τ_i pueden aplazarse en función de la cantidad de tiempo g_i no consumido por la tarea τ_i, utilizando para ello la ecuación:

$$\forall j \in lp(i) : y_j(t) = max(y_j(t), t + g_i) \tag{2.21}$$

Variación en los Instantes de Activación

Cuando los instantes de activación de una tarea pueden variar, el tiempo de respuesta en el peor caso se obtiene a partir de la siguiente ecuación, asumiendo una variación máxima entre activaciones igual a J_i.

$$R_i = w_i + Y_i + J_i \tag{2.22}$$

$$w_i^{m+1} = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i^m + J_j}{T_j} \right\rceil C_j \tag{2.23}$$

En este caso, la interferencia máxima ocasionada por una tarea τ_j sucede cuando dicha tarea se ha activado con un retardo de J_j unidades de tiempo, su instante de promoción coincide con el inicio del periodo ocupado a nivel de prioridad de la tarea τ_i y las activaciones sucesivas de τ_j no sufren variación.

Plazos Arbitrarios

Cuando las tareas tienen plazos arbitrarios también es posible aplicar el algoritmo DP. En este caso se analiza el tiempo de respuesta de q intervalos ocupados al nivel de prioridad de la tarea τ_i objeto de estudio. El tiempo de respuesta obtenido para cada una de las q invocaciones es el siguiente:

$$R_i(q) = w_i(q) - qT_i + Y_i + J_i \tag{2.24}$$

$$w_i^{m+1}(q) = (q+1)C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i^m(q) + J_j}{T_j} \right\rceil C_j \tag{2.25}$$

De esta forma, el peor tiempo de respuesta para una tarea τ_i es:

$$R_i = max_{q=0,1,2,3,\ldots}\Big(w_i(q) - qT_i\Big) + Y_i + J_i \tag{2.26}$$

2.4. Modificaciones al Algoritmo Dinámico Extractor de Holgura

En esta sección se proponen varias modificaciones al algoritmo dinámico extractor de holgura, corrigiendo algunas deficiencias del mismo relacionadas con la compartición de recursos entre tareas críticas y aperiódicas y la presencia de tareas esporádicas.

2.4.1. Compartición de Recursos entre Tareas Críticas y Aperiódicas

Uno de los principales inconvenientes del algoritmo dinámico de extracción de holgura estriba en que el método que permite la compartición de recursos entre tareas críticas y aperiódicas es muy poco satisfactorio.

Tal como se ha descrito en la sección 2.2, para permitir que una tarea aperiódica cierre un semáforo es necesario que se verifique la siguiente condición

$$\min_{\forall i \in lp(j)} (S_i^{max}(t) - B_i) > c \tag{2.27}$$

siendo c la duración de la sección crítica y j el techo del semáforo que se desea cerrar.

Esta solución dista mucho de ser satisfactoria por lo siguiente.

En primer lugar, el sistema necesita conocer, en tiempo de ejecución, la duración de cada sección crítica c, algo que no está contemplado en ninguna de las interfaces existentes de utilización de semáforos (o mutex POSIX u operaciones protegidas de Ada). Estas interfaces podrían modificarse, pero no parece aconsejable que el sistema deba tomar decisiones en función de tiempos de cómputo estimados por la aplicación, especialmente en algo tan sensible como la protección de las secciones críticas.

Segundo, cuando a una tarea aperiódica no se le permite cerrar un semáforo, el sistema debe suspenderla y recordar la cantidad de tiempo que requiere para completar su sección crítica. Con el fin de poder reactivarla, no basta con examinar el estado del semáforo, sino que es necesario examinar la lista de peticiones pendientes de cierre de semáforos cada vez que se incrementa la holgura disponible, lo que introduce complejidad y sobrecarga al sistema.

Por último, es necesario tener en cuenta que las operaciones de cierre y apertura de semáforos suelen implementarse de forma muy eficiente y la incorporación del criterio

propuesto para admitir tareas aperiódicas causaría una pérdida importante de eficiencia.

En el resto de esta sección se estudia la viabilidad de utilizar los protocolos de herencia de prioridad en sistemas en los cuales se utiliza el algoritmo extractor de holgura. Para ello se define en primer lugar una variante del algoritmo extractor de holgura y a continuación se estudia el comportamiento de los protocolos de herencia de prioridades cuando se utiliza esta variante del algoritmo.

Nueva Versión del Algoritmo Extractor de Holgura Dinámico

En este apartado se diseña una nueva versión del algoritmo extractor de holgura. Con este algoritmo se pretende poder aplicar los protocolos de herencia de prioridades tanto a tareas periódicas como aperiódicas en sistemas que utilizan el algoritmo de extracción de holgura.

Uno de los principales motivos por el que los protocolos de herencia de prioridad no pueden ser aplicados es debido a que el algoritmo extractor de holgura no define de forma precisa y adecuada la asignación de prioridades para las tareas aperiódicas. Por ello, introducimos en el algoritmo extractor de holgura un nuevo método de asignación de prioridades.

Con el fin de simplificar el criterio por el cual una tarea aperiódica puede ejecutarse, definamos la holgura del sistema S_S^{max} como

$$S_S^{max} = \min_{\forall i}(S_i(t) - B_i) \tag{2.28}$$

siendo $S_i(t) = S_i^{max}(t)$ si τ_i tiene menor o igual prioridad que la tarea crítica τ_k preparada más prioritaria o $S_i = \infty$ en caso contrario.

Definimos también las siguientes reglas de asignación de prioridades, similares a las del algoritmo de asignación dual de prioridades.

- El rango de prioridades del sistema está dividido en tres bandas, denominadas baja, media y alta.
- Las tareas críticas utilizan dos prioridades, una prioridad de la banda alta y otra de la baja. La asignación de prioridades para tareas críticas es tal que si una tarea i es más prioritaria que una tarea k en la banda alta, entonces i también es más prioritaria que k en la banda baja.
- La prioridad de una tarea crítica es su prioridad de la banda alta cuando $S_S^{max} = 0$, es decir, cuando no hay holgura en el sistema.

- La prioridad de una tarea crítica es su prioridad de la banda baja cuando $S_S^{max} > 0$, es decir, cuando hay holgura en el sistema.
- La prioridad de una tarea aperiódica siempre pertenece a la banda media.

Esta asignación de prioridades equivale a la original, excepto que una tarea aperiódica no necesariamente tendrá más prioridad que una crítica siempre que haya holgura. Si una tarea crítica hereda una prioridad como consecuencia de la aplicación de un protocolo de herencia de prioridades, esta prioridad heredada pertenecerá a la banda alta, tal como se expondrá a continuación, y la tarea aperiódica esperará a la tarea crítica aunque haya holgura en el sistema.

Tal como se demostrará en los apartados siguientes, este algoritmo es compatible con la aplicación de los protocolos de herencia de prioridad. Su principal inconveniente es el elevado número de cambios de prioridad que produce en las tareas críticas. No obstante, tal como se comenta al final de esta sección, es posible implementar de forma eficiente este algoritmo, evitando en la práctica estos cambios de prioridad.

Antes de abordar el estudio de los protocolos de herencia de prioridad, es importante tener en cuenta lo siguiente. Supongamos que τ_i es una tarea crítica que se activa en el instante t_0. La ejecución de τ_i puede ser interrumpida por ejecuciones de tareas aperiódicas mientras la holgura del sistema sea mayor que cero. Durante la ejecución de τ_i es posible también que su holgura se haga igual a cero en varias ocasiones pero se recupere antes de que τ_i termine. Esto es posible, por ejemplo, si tareas más prioritarias que ella no consumen todo su tiempo de cómputo posible. Finalmente puede suceder que en un instante t_1 la holgura de la tarea τ_i se haga cero y no se recupere hasta después del instante t_2 en el que dicha tarea termine su instancia actual. Dada la definición de holgura, en el intervalo (t_1, t_2) el sistema dispone de tiempo de procesamiento suficiente para que t_2 sea anterior al próximo vencimiento del plazo de la tarea τ_i. Este tiempo de procesamiento permite ejecutar lo que le resta a la instancia actual de τ_i, la interferencia que τ_i puede sufrir por parte de tareas más prioritarias y un máximo de B_i unidades de tiempo provenientes de otras tareas que pueden causar bloqueo a τ_i.

A raíz de lo expuesto, se observa que la ejecución de una instancia de una tarea crítica τ_i tiene dos fases. En la primera fase, desde t_0 a t_1, es irrelevante el número de ocasiones en las que la tarea puede ver retrasada su ejecución por causa de tareas menos prioritarias, incluyendo tareas aperiódicas que ejecutan secciones críticas, ya que existe holgura en el sistema suficiente para que τ_i termine a tiempo. Es en la segunda fase, desde t_1 a t_2, cuando el retraso producido por tareas menos prioritarias debe ser a lo sumo de B_i unidades de tiempo, puesto que de lo contrario τ_i podría perder su plazo. Por tanto, el análisis de las consecuencias de aplicar los protocolos de herencia de prioridad debe centrarse exclusivamente en el intervalo (t_1, t_2).

Adaptación del Protocolo CSP al Algoritmo DSSA

Si se utiliza el protocolo de herencia de prioridades CSP junto con la nueva versión del algoritmo extractor de holgura, se propone considerar a las tareas aperiódicas como si fuesen tareas críticas en todos los aspectos relacionados con el acceso a recursos compartidos. Esto es:

- Cuando un semáforo es utilizado por una tarea crítica, se utiliza a efectos del cómputo del techo de dicho semáforo la prioridad alta de la tarea crítica, tanto si dicho semáforo es compartido sólo por tareas críticas o por tareas críticas y aperiódicas.
- Las secciones críticas de todas las tareas, críticas o aperiódicas, son consideradas en el cálculo del factor de bloqueo de las tareas críticas.
- Las reglas de herencia de prioridades del protocolo CSP se aplican por igual a las tareas críticas y aperiódicas.

Por supuesto, a las tareas aperiódicas se les exige las mismas restricciones que a las tareas críticas en el uso de semáforos requeridas por el protocolo CSP, es decir

- Las secciones críticas deben estar anidadas.
- Las secciones críticas deben tener tiempos de cómputo limitados.
- Las tareas aperiódicas no se suspenden cuando están ejecutando una sección crítica.

Cuando se utiliza el protocolo CSP, una tarea crítica τ_i puede verse bloqueada por tareas críticas τ_k de menor prioridad o aperiódicas α_l que estén ejecutando una sección crítica cuyo techo de prioridad P sea mayor o igual que la prioridad de τ_i.

Sabemos por las propiedades del protocolo CSP que a lo sumo una tarea crítica puede causar bloqueo a otra tarea crítica de mayor prioridad, propiedades que se mantienen en el nuevo método de asignación de prioridades para el algoritmo extractor de holgura, puesto que las tareas críticas siempre mantienen la misma relación de prioridad, tanto si están ejecutándose con holgura en el sistema o sin ella. A lo sumo se puede manifestar un comportamiento distinto al habitual cuando una tarea crítica ejecuta una sección crítica cuando hay holgura en el sistema. En este caso, la tarea adquiere la prioridad P del techo del semáforo que guarda la sección crítica y pasa a la banda alta de prioridades, excluyendo la ejecución de cualquier otra tarea del sistema, incluso aquellas tareas cuya prioridad de la banda alta sea superior a P. Por supuesto, esta situación sólo se mantiene mientras exista holgura en el sistema. Analicemos a continuación qué sucede cuando es una tarea aperiódica la que ejecuta una sección crítica.

Una tarea aperiódica α_l puede iniciar una sección crítica sólo si no hay otra sección crítica en ejecución, puesto que de lo contrario la prioridad actual del sistema pertenecería a la banda alta de prioridades y α_l no podría haberse ejecutado. De esta forma, sabemos que sólo una de entre las tareas aperiódicas podrá estar ejecutando una sección crítica.

También podemos deducir que si una tarea aperiódica α_l está ejecutando una sección crítica cuyo techo es P, no podrá haber ninguna tarea crítica τ_i ejecutando una sección crítica con techo igual o inferior a P. Cuando α_l inició su sección crítica ninguna otra sección crítica podía estar en ejecución. Para que una tarea crítica inicie después de α_l su sección crítica, es necesario que la prioridad de τ_i sea superior a P. En este caso el techo de dicha sección crítica tendrá que ser también superior a P, puesto que por la definición del protocolo CSP, una tarea crítica sólo puede iniciar secciones críticas cuyo techo sea igual o superior a su prioridad.

Supongamos que vence la holgura mientras α_l está ejecutando una sección crítica con techo P. La sección crítica de α_l bloqueará a todas las tareas críticas cuya prioridad sea inferior a P. Como no puede haber ninguna otra tarea ejecutando una sección crítica con techo igual o inferior a P, estas tareas sólo podrán ser bloqueadas por α_l.

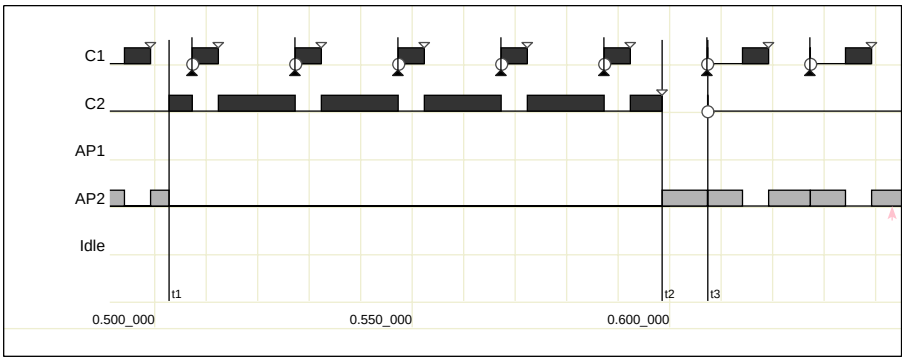
Ilustremos lo expuesto anteriormente con un ejemplo. Sea un sistema con cinco tareas, tres críticas, τ_0, τ_1 y τ_2, con prioridades 20, 21 y 22 en la banda alta y 0, 1 y 2 en la banda baja, y dos aperiódicas α_0 y α_1 con prioridades en la banda media de 10 y 11. (La prioridad 0 es la más baja del sistema). Supongamos que α_0 está en ejecución e inicia una sección crítica cuyo techo es 21. Esta sección crítica podrá causar bloqueo a las tareas τ_0 y τ_1, las cuales no podrán ejecutarse hasta que α_0 concluya su sección crítica. Tampoco podrá ejecutarse la tarea aperiódica α_1. Si ahora vence la holgura del sistema y no se recupera hasta que τ_0 termina su instancia actual, sólo α_0 causará bloqueo a τ_0 y τ_1. La única tarea que podrá iniciar otra sección crítica será τ_2, cuya prioridad es 22, y esta tarea no causa bloqueo a τ_0 y τ_1.

A raíz de este análisis concluimos que cuando una tarea crítica es bloqueada por una tarea aperiódica, no puede ser bloqueada simultáneamente por otra tarea crítica o aperiódica. De esta forma se mantiene la propiedad del protocolo CSP que garantiza que una tarea es bloqueada a lo sumo una vez por tareas de prioridad inferior. Como las secciones críticas de las tareas aperiódicas se tienen en cuenta para calcular el factor de bloqueo de las secciones críticas, y en la holgura del sistema se han descontado estos factores de bloqueo, puede afirmarse que la aplicación del protocolo CSP junto con la nueva versión del algoritmo extractor de holgura no provoca la pérdida de ningún plazo de las tareas críticas.

El funcionamiento de este método de compartición de recursos entre tareas críticas y aperiódicas puede observarse en las figuras 2.7 y 2.8.

En la figura 2.7 se observa la ejecución de dos tareas críticas (C1 y C2) y dos tareas aperiódicas (AP1 y AP2). La tareas C2 y AP1 comparten el mismo semáforo, y la duración de la sección crítica que guarda dicho semáforo es igual a 8ms. En t_1 finaliza la holgura de la tarea C2, por lo que AP2 es interrumpida y siguen en ejecución C1 y C2. C2 concluye su activación en el instante t_2, instante en el que la holgura del sistema se hace positiva. Desde t_2 hasta el plazo de C2 en t_3 se produce un intervalo de 8ms, que corresponde con el factor de bloqueo de la tarea C2, el cual había sido tenido en cuenta para calcular la holgura de C2.

La figura 2.8 representa la misma ejecución que la figura 2.7, con la excepción de que en este caso la tarea AP1 se activa en t_0, y antes de que se consuma la holgura adquiere un mutex cuyo techo de prioridad es igual a 10. En t_1 se agota la holgura, y la tarea AP1 bloquea a C2. La ejecución de la sección crítica es interrumpida por la tarea C1, la cual tiene mayor prioridad que el techo del mutex y por tanto no es bloqueada. Cuando finaliza la sección crítica, AP1 vuelve a su prioridad de la banda media, por lo que C2 pasa a ejecución hasta que termina su activación en t'_2. Puede observarse que aunque C2 concluye más tarde que en el caso anterior, lo hace antes de su plazo, dado que en el cálculo de la holgura se había previsto un factor de bloqueo al menos igual que la duración de la sección crítica de AP1.

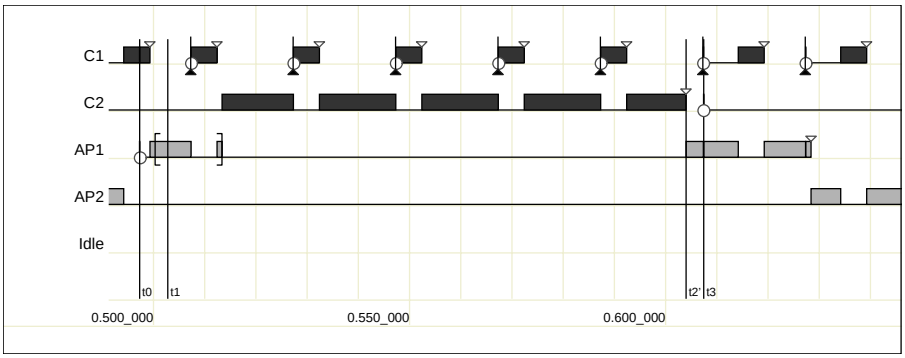


$C1 : (T = D = 20ms, C = 5ms, B = 0ms, Prio = 12)$
$C2 : (T = D = 300ms, C = 70ms, B = 8ms, Prio = 10)$

Figura 2.7: Recursos Compartidos con el Algoritmo Extractor de Holgura I

Adaptación del Protocolo PCP al Algoritmo DSSA

Para utilizar el protocolo PCP junto con la extracción de holgura partimos de los mismos supuestos que con el protocolo CSP descritos en el apartado anterior. Adicionalmente



$C1 : (T = D = 20ms, C = 5ms, B = 0ms, Prio = 12)$
$C2 : (T = D = 300ms, C = 70ms, B = 8ms, Prio = 10)$

Figura 2.8: Recursos Compartidos con el Algoritmo Extractor de Holgura II

es necesario que cuando una tarea τ_i, crítica o aperiódica, hereda la prioridad de otra tarea crítica τ_j, debe heredar la prioridad de la banda alta de τ_j, independientemente de si hay o no holgura en el sistema.

Esta modificación de las reglas del protocolo PCP es necesaria, puesto que de lo contrario puede producirse una inversión de prioridad diferente a la causada por el uso de recursos compartidos. Ilustremos esta posibilidad con un ejemplo.

Supongamos que se aplica la regla de herencia de prioridades teniendo en cuenta las prioridades actuales de las tareas. Sea un sistema formado por tres tareas críticas, τ_0, τ_1 y τ_2, con prioridades 20, 21 y 22 en la banda alta y 0, 1 y 2 en la banda baja (La prioridad 0 es la más baja del sistema). Supongamos que habiendo holgura τ_0 inicia una sección crítica cuyo techo es 20. τ_2 intenta iniciar una sección crítica guardada por el mismo semáforo y se bloquea (su prioridad es ahora 2 y el techo del sistema es 20). τ_0 hereda la prioridad 2. Supongamos que finaliza la holgura del sistema, con lo que las tres tareas recuperan su prioridad de la banda alta. Antes de que τ_0 finalice su sección crítica, se activa τ_1 y pasa a ejecución interrumpiendo a τ_0. La tarea τ_2 podrá ejecutarse cuando τ_1 termine y τ_0 finalice su sección crítica. De esta forma, se produce una inversión de prioridad no prevista (la ejecución de τ_1) y τ_2 puede perder su plazo.

Cuando se utiliza el protocolo PCP, una tarea crítica τ_i puede verse bloqueada por tareas críticas τ_k de menor prioridad o aperiódicas α_l que estén ejecutando una sección crítica cuyo techo de prioridad P sea mayor o igual que la prioridad de τ_i. Este bloqueo sucederá si τ_i está preparada para ejecución y alguna tarea de mayor o igual prioridad que τ_i (incluida τ_i) y prioridad igual o inferior a P intenta iniciar una sección crítica.

Sabemos por las propiedades del protocolo PCP que a lo sumo una tarea crítica puede causar bloqueo a otra tarea crítica de mayor prioridad, propiedades que se mantienen

en el nuevo método de asignación de prioridades para el algoritmo extractor de holgura puesto que las tareas críticas siempre mantienen la misma relación de prioridad, tanto si están ejecutándose con holgura en el sistema o sin ella. A lo sumo se puede manifestar un comportamiento distinto al habitual cuando una tarea crítica τ_i inicia una sección crítica habiendo holgura en el sistema. En este caso, si otra tarea τ_j con prioridad mayor que la de τ_i intenta iniciar una sección crítica se verá bloqueada, con independencia del valor actual del techo del sistema P^s, puesto que la prioridad de τ_j pertenece a la banda baja de prioridades. Por supuesto, esta situación sólo se mantiene mientras exista holgura en el sistema. Analicemos a continuación qué sucede cuando una tarea aperiódica ejecuta una sección crítica.

Una tarea aperiódica α_l puede iniciar una sección crítica sólo si no hay otra sección crítica en ejecución, puesto que de lo contrario el techo del sistema P^s impediría que α_l pueda iniciar su sección crítica. Recordemos que si hay alguna sección crítica en ejecución el techo del sistema P^s es siempre mayor que la prioridad de cualquier tarea aperiódica. Como consecuencia de esto, sabemos que sólo una de entre las tareas aperiódicas podrá estar ejecutando una sección crítica.

También podemos concluir que si una tarea aperiódica α_l está ejecutando una sección crítica con techo P, no podrá haber ninguna tarea crítica ejecutando una sección crítica con techo igual o inferior a P. Una vez α_l esté ejecutando su sección crítica sólo tareas con prioridad superior a P podrán iniciar secciones críticas cuyos techos sean superiores a P.

Supongamos que vence la holgura mientras α_l está ejecutando una sección crítica con techo P. En este momento todas las tareas críticas recuperan su prioridad de la banda alta. Si una tarea con prioridad j inferior o igual a P intenta cerrar un semáforo, la sección crítica de α_l bloqueará a todas las tareas críticas cuya prioridad sea igual o inferior a j. Como no puede haber ninguna tarea crítica ejecutando una sección crítica con techo igual o inferior a P, estas tareas sólo podrán ser bloqueadas por α_l.

Ilustremos lo expuesto anteriormente con un ejemplo. Sea un sistema con cinco tareas, tres críticas, τ_0, τ_1 y τ_2, con prioridades 20, 21 y 22 en la banda alta y 0, 1 y 2 en la banda baja, y dos aperiódicas α_0 y α_1 con prioridades en la banda media de 10 y 11. (La prioridad 0 es la más baja del sistema). Supongamos que α_0 inicia una sección crítica cuyo techo es 21. Esta sección crítica podrá causar bloqueo a las tareas τ_0 y τ_1, las cuales no podrán iniciar una sección crítica hasta que α_0 concluya su sección crítica. Supongamos que vence la holgura del sistema y no se recupera hasta que τ_0 termina su instancia actual. Si τ_1 intenta iniciar una sección crítica, sólo α_0 causará bloqueo a τ_0 y τ_1. La única tarea que podrá iniciar otra sección crítica será τ_2, cuya prioridad es 22, y esta tarea no causa bloqueo a τ_0 y τ_1.

A raíz de este análisis concluimos que cuando una tarea crítica es bloqueada por una

tarea aperiódica, no puede ser bloqueada simultáneamente por otra tarea crítica o aperiódica. De esta forma se mantiene la propiedad del protocolo PCP que garantiza que una tarea es bloqueada a lo sumo una vez por tareas de prioridad inferior. Como las secciones críticas de las tareas aperiódicas se tienen en cuenta para calcular el factor de bloqueo de las secciones críticas, y en la holgura del sistema se han descontado estos factores de bloqueo, puede afirmarse que la aplicación del protocolo PCP junto con la nueva versión del algoritmo extractor de holgura tampoco provoca la pérdida de ningún plazo de las tareas críticas.

Adaptación del Protocolo PIP al Algoritmo DSSA

Si se utiliza el protocolo básico de herencia de prioridades junto con la nueva versión del algoritmo extractor de holgura, se propone considerar a las tareas aperiódicas como si fuesen tareas críticas en todos los aspectos relacionados con el acceso a recursos compartidos. De esta forma, cuando se calcula el factor de bloqueo para una tarea crítica, las tareas aperiódicas se consideran como tareas de menor prioridad. Igualmente se tienen en cuenta las secciones que ejecutan las tareas aperiódicas que pueden causar bloqueo a tareas críticas.

Para utilizar el protocolo de herencia básico junto con la extracción de holgura es necesario que cuando una tarea τ_i hereda la prioridad de otra tarea τ_j, debe heredar la prioridad de la banda alta de τ_j, independientemente de si hay o no holgura en el sistema. Esta modificación de las reglas de este protocolo es necesaria, puesto que de lo contrario puede producirse una inversión de prioridad diferente a la causada por el uso de recursos compartidos. Los motivos de esta posible inversión de prioridad son equivalentes a los del protocolo PCP.

Tal como se ha comentado antes, el análisis del bloqueo que una tarea puede sufrir por parte de tareas menos prioritarias se limita al intervalo de tiempo que transcurre desde que la holgura de una tarea se hace cero hasta que dicha tarea finaliza. Durante este intervalo de tiempo, todas las tareas críticas se ejecutan con su prioridad en la banda alta, y las tareas aperiódicas utilizan una prioridad menor, perteneciente a la banda media. Esta asignación de prioridades permite que las siguientes propiedades de este protocolo se mantengan.

- Una tarea puede verse bloqueada a lo sumo una vez por cada tarea de menor prioridad.

  Cuando vence la holgura para una tarea τ_i, puede verse bloqueada una vez por cada tarea τ_k de menor prioridad, crítica o aperiódica, siempre que estas tareas hayan heredado una prioridad igual o superior a la de τ_i. Cuando τ_k finaliza su sección crítica, vuelve a su prioridad normal y ya no puede ejecutarse hasta que τ_i abandone el procesador, por lo que sólo puede bloquear una vez a τ_i.

- Si hay m semáforos que pueden bloquear a una tarea τ_i, τ_i puede verse bloqueada a lo sumo m veces.

  Durante el intervalo en el que una tarea τ_i se ejecuta sin holgura hasta que termina, una vez las tareas que tienen un semáforo cerrado y están bloqueando a τ_i abran dicho semáforo, no podrán volver a ejecutarse hasta que τ_i abandone el procesador, por lo que sólo podrán bloquear una vez a τ_i.

Implementación

El principal inconveniente del método de asignación de prioridades propuesto en esta sección es el elevado número de cambios de prioridad que provoca en el sistema, ya que todas las tareas críticas cambian de prioridad cuando la holgura pasa de cero a positiva y viceversa. En este apartado se discute una implementación que permite realizar estos cambios de forma eficiente.

Una de las estructuras de datos más adecuada para implementar las políticas de planificación definidas en Ada y POSIX es un conjunto de colas FIFO, donde cada cola está asociada a un nivel de prioridad. Esta es, por ejemplo, la estructura de datos utilizada por MaRTE OS, donde se recurre también a instrucciones específicas del procesador que permiten determinar de forma muy eficiente si una determinada cola está o no vacía, o cual es la primera cola no vacía. En esta implementación las operaciones de selección de la próxima tarea a ejecutar, y la inserción y extracción de tareas se ejecutan con un coste constante en la mayoría de los casos.

Para implementar el método de asignación de prioridades descrito en esta sección se propone utilizar un conjunto de colas FIFO adicional. En el primer conjunto de colas Δ se sitúan las tareas aperiódicas, mientras que en el segundo conjunto Γ situamos a las tareas críticas. Cuando hay holgura en el sistema se selecciona la tarea más prioritaria del conjunto Δ, salvo que esté vacío, en cuyo caso se seleccionaría la tarea más prioritaria del conjunto Γ. Por contra, cuando no hay holgura en el sistema, se selecciona la tarea más prioritaria de ambos conjuntos de colas, dando preferencia en caso de igualdad de prioridades al conjunto Δ.

En la figura 2.9 se muestra un ejemplo de esta disposición de tareas en dos conjuntos de colas. Las tareas críticas utilizan las prioridades 4 a 7 mientras que las tareas aperiódicas utilizan las prioridades 1 a 3. Cuando hay holgura en el sistema, las prioridades de las tareas críticas son inferiores a las de las tareas aperiódicas, ya que se utiliza en primer lugar el conjunto Δ. Por contra, cuando no hay holgura en el sistema las prioridades de las tareas críticas son superiores a las de las tareas aperiódicas, ya que en esta situación se selecciona la tarea más prioritaria de ambos conjuntos de colas.

Adicionalmente, cuando una tarea crítica hereda alguna prioridad por causa del proto-

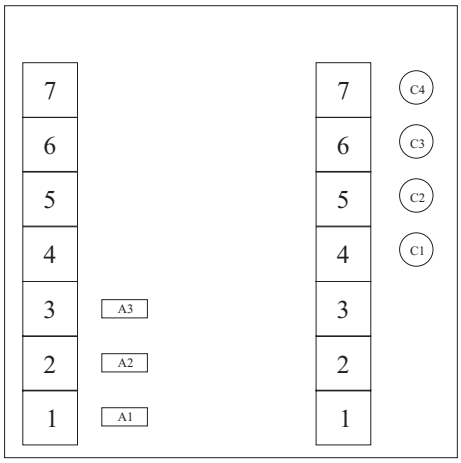


Figura 2.9: Implementación del Método de Asignación de Prioridades I

colo de herencia de prioridades que se utilice, en lugar de cambiar a otra cola del conjunto Γ, es movida a una cola del conjunto Δ. Cuando la tarea crítica deja de tener prioridades heredadas, vuelve al conjunto Γ.

La figura 2.10 muestra una situación en la que varias tareas tienen prioridades heredadas como consecuencia del uso de los protocolos de herencia de prioridad. En particular, la tarea aperiódica $A2$ tiene heredada la prioridad 5 mientras que la tarea crítica $C3$ ha heredado la prioridad 7. Esta situación puede suceder, por ejemplo, si se utiliza el protocolo CSP, la tarea $A2$ ha cerrado un semáforo con techo 5 y posteriormente, tras haberse agotado la holgura del sistema, la tarea $C3$ ha cerrado un semáforo cuyo techo es igual a 7.

Tal como puede observarse, esta implementación es equivalente a aplicar el método de asignación de prioridades descrito en esta sección en un único conjunto de colas compartido por tareas críticas y aperiódicas, y origina una sobrecarga significativamente menor.

2.4.2. Coexistencia con Tareas Esporádicas

Tal como se ha comentado en la sección 2.2, la holgura en un instante t' puede derivarse a partir de la holgura en un instante t. Este método es esencial para que sea factible utilizar en la práctica el algoritmo DSSA, pero tiene el inconveniente de que no permite determinar la holgura exacta si hay tareas esporádicas en el sistema. Analicemos a continuación el por qué de este inconveniente.

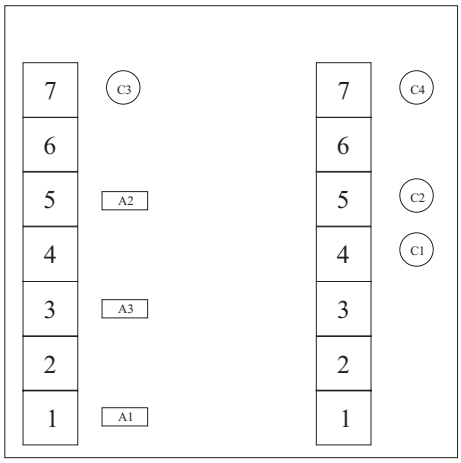


Figura 2.10: Implementación del Método de Asignación de Prioridades II

El método original para derivar la holgura consiste en:

- La holgura de cada tarea periódica o esporádica se calcula al final de cada activación.
- Cuando en un cambio de contexto la tarea que abandona el procesador es aperiódica, o el procesador estaba ocioso, la holgura es decrementada para todas las tareas críticas en una cantidad igual al tiempo transcurrido desde el cambio de contexto anterior.
- Cuando en un cambio de contexto la tarea τ_j que abandona el procesador es crítica, la holgura es decrementada para todas las tareas críticas con mayor prioridad que τ_i en una cantidad igual al tiempo transcurrido desde el cambio de contexto anterior.

Si aplicamos este método en presencia de tareas esporádicas, la holgura calculada podrá ser menor que la real cuando las tareas esporádicas se activen con menor frecuencia de lo posible. Esta estimación incorrecta de la holgura es producida por dos causas distintas.

La primera causa de la estimación incorrecta de la holgura es la siguiente. Supongamos que finaliza la activación de una tarea esporádica τ_i, y en ese momento se calcula la holgura para dicha tarea S_i. A partir de este instante, en los cambios de contexto oportunos el valor de $S_i(t)$ se irá decrementando, pudiendo incluso llegar a hacerse igual a cero. Si τ_i fuese una tarea periódica sabemos que su holgura se recalcula una vez en cada periodo. En cambio, ya que τ_i es una tarea esporádica, desconocemos cual será su próximo instante de activación, y no podemos determinar cuando $S_i(t)$ será recalculada. Como

consecuencia de lo anterior, si t_1 es el instante en el que τ_i hubiese terminado su próxima instancia si hubiera sido una tarea periódica y t_2 es el instante real en que dicha instancia termina, el método de derivación de holgura determinará unos valores de $S_i(t)$ en el intervalo (t_1, t_2) inferiores a los reales.

Si en dicho intervalo $S_i(t)$ se hace igual a cero, según la ecuación 2.14, si la prioridad de τ_i es menor que la tarea preparada más prioritaria, el sistema no podrá planificar ninguna tarea aperiódica hasta t_2, aunque en realidad la holgura real será diferente a cero. En un caso extremo, si τ_i nunca volviera a activarse, ninguna tarea aperiódica podría ser ya planificada cuando la prioridad del sistema fuese superior a la prioridad de τ_i.

La segunda causa de la estimación incorrecta de la holgura tiene también su origen cuando una tarea esporádica se activa con menor frecuencia de lo posible. Cuando se calcula la holgura para una tarea τ_i se tiene en cuenta la máxima interferencia posible que tareas más prioritarias pueden ocasionar a τ_i. Si entre estas tareas hay alguna tarea esporádica τ_j, y τ_j no se activa a la máxima frecuencia, la interferencia que τ_j causa en τ_i es menor que la prevista, y por tanto, la holgura de τ_i es mayor. El método de derivación de holgura no detecta esta circunstancia, por lo que la holgura calculada puede ser menor que la real.

Con el fin de subsanar estos dos problemas, los autores de este método proponen que, en el intervalo de tiempo que transcurre desde que una tarea esporádica τ_i puede activarse hasta que se activa realmente, la holgura de τ_i y de todas las tareas menos prioritarias que ella debe recalcularse cada tick de reloj, con el fin de disponer de la holgura correcta en todo momento, propuesta que como sus autores reconocen es inaplicable en la práctica. Los mismos autores proponen para calcular de forma aproximada la holgura en este caso recalcularla para todas las tareas de forma periódica, introduciendo una tarea encargada de realizar este cálculo. El inconveniente de esta propuesta estriba en que esta tarea debe ser considerada en el análisis de la planificabilidad de la aplicación y su sobrecarga es muy considerable.

Aunque el segundo problema expuesto tiene difícil solución y no hemos encontrado un método eficiente de abordarlo, se ha encontrado un método alternativo para derivar la holgura que al menos evita el primero de los problemas expuestos, método que se expone a continuación:

- La holgura de cada tarea periódica o esporádica se calcula cuando dicha tarea se activa. La holgura se calcula desde el instante en que la tarea se activa hasta su plazo.

- La holgura de cada tarea periódica o esporádica se hace igual a ∞ cuando su instancia actual termina.

- Cuando en un cambio de contexto la tarea que abandona el procesador es aperiódica, o el procesador estaba ocioso, la holgura es decrementada para todas las tareas críticas (con holgura diferente a ∞) en una cantidad igual al tiempo transcurrido desde el cambio de contexto anterior.

- Cuando en un cambio de contexto la tarea τ_i que abandona el procesador es crítica, la holgura es decrementada para todas las tareas críticas con mayor prioridad que τ_i (y con holgura diferente a ∞) en una cantidad igual al tiempo transcurrido desde el cambio de contexto anterior.

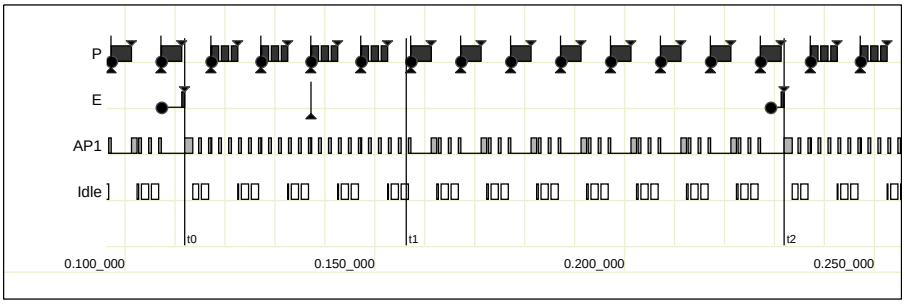
En este método se considera que cuando una tarea finaliza una activación puede no volver a activarse nunca, por lo que su holgura se considera a partir de este instante infinita. Aunque para las tareas periódicas conocemos su próximo instante de activación, el método es igualmente aplicable para este tipo de tareas, por lo que no se hace distinción entre tareas periódicas y esporádicas, lo que redunda en una mayor sencillez de implementación. Cuando una tarea se activa se calcula su holgura, instante en el que se "corrige" la holgura de la tarea que hasta ahora era infinita.

Evidentemente, este método no permite conocer la holgura real, pero tiene la propiedad de que permite conocer la holgura necesaria para poder admitir la ejecución de tareas aperiódicas en el sistema sin comprometer el plazo de las tareas críticas. Analicemos esta propiedad a continuación.

Supongamos que una tarea esporádica τ_i finaliza su activación en el instante t_1 y se activa de nuevo en el instante t_2, siendo t_2 igual o posterior al primer instante en que τ_i puede activarse. Según este nuevo método de derivación de holgura, sabemos que en cualquier instante $t \in (t_1, t_2)$ la holgura calculada $S_i(t)$ para τ_i será igual a ∞. De esta forma, en el intervalo (t_1, t_2), la holgura real $S_i^{max}(t)$ de τ_i no se utiliza para decidir si una tarea aperiódica puede ejecutarse en (t_1, t_2). Las tareas aperiódicas serán admitidas para ejecución utilizando la holgura de las tareas que estén preparadas en el sistema y no la holgura de las tareas que estén a la espera de su próxima activación. Si podemos demostrar que $\forall t \in (t_1, t_2)$: $S_i^{max}(t) > 0$, no importará que $S_i(t)$ sea igual a ∞, ya que $S_i^{max}(t)$ no va a condicionar la ejecución de ninguna tarea aperiódica.

Para demostrar esta propiedad, basta con examinar que valores toma la holgura real de una tarea. $S_j^{max}(t)$ es la suma de intervalos en los que el procesador está ocioso desde t hasta el próximo plazo de la tarea τ_i, asumiendo que no hay en el sistema tareas menos prioritarias que τ_i. Puesto que τ_i finaliza su activación en t_1 y la reanuda en t_2, en el intervalo (t_1, t_2) el procesador estará ocioso. De esta forma $S_j^{max}(t_1) > t_2 - t_1$, con lo que $\forall t \in [t_1, t_2)$: $S_j^{max}(t) > 0$.

De esta forma, es posible aplicar el algoritmo DSSA a tareas periódicas y esporádicas por igual sin que se produzcan situaciones en las la presencia de tareas esporádicas anula



$P : (T = D = 10ms, C = 4ms, Prio = 12)$ $E : (T = D = 30ms, C = 0{,}5ms, Prio = 10)$

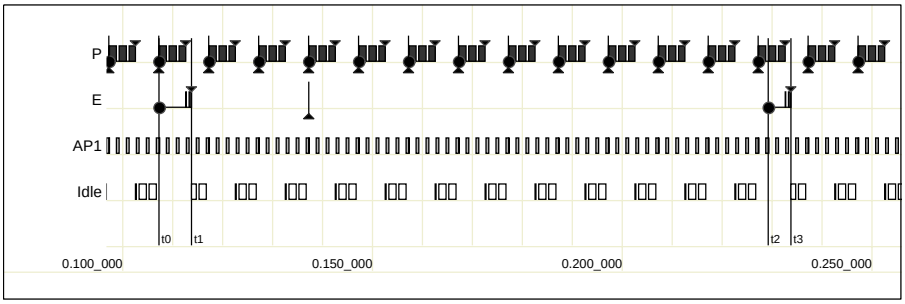
Figura 2.11: Derivación de la Holgura en Presencia de Tareas Esporádicas. Método Original

completamente la holgura del sistema. Una pequeña ventaja añadida es que el intervalo en el que se calcula la holgura es menor, por lo que la activación con menor frecuencia de las tareas esporádicas afecta menos a la cantidad de holgura que se detecta.

No obstante este método no está exento de inconvenientes. El cálculo de la holgura forma parte ahora de la activación de cada tarea, por lo que puede ser una causa de inversión de prioridad. Tampoco permite conocer la cantidad de holgura, aunque aproximada, en todo momento, ya que no se dispone de información relativa a las tareas cuya holgura es infinita. Finalmente, desde un punto de vista de implementación ocasiona más programaciones del temporizador que notifica que la holgura se hace igual a cero.

A continuación se ilustra lo expuesto en esta sección con un ejemplo. En la figura 2.11 se ejecutan dos tareas críticas, P y E, junto con una tarea aperiódica AP1. P es una tarea periódica y E es una tarea esporádica. Las activaciones de E no se producen a la máxima frecuencia posible. El método utilizado para derivar la holgura es el original. Cuando la tarea E finaliza su primera activación en t_0, se calcula su holgura. La holgura de E se agota en t_1. A partir de este instante se observa que la holgura del sistema se hace cero y AP1 deja de ejecutarse con preferencia sobre P, a pesar de existir realmente holgura. Sólo cuando en t_2 la holgura de E se calcula de nuevo, AP1 puede volver a hacer uso de la holgura y ejecutarse con preferencia sobre P.

La figura 2.12 representa la misma ejecución que la figura 2.11, pero en este caso se utiliza el nuevo método propuesto para derivar la holgura. Utilizando este método, la holgura se calcula cada vez que E comienza su ejecución, lo que sucede en los instantes t_0 y t_2. Cuando E finaliza, su holgura se hace infinita, lo que sucede en t_1 y t_3. Como puede observarse, la holgura del sistema se utiliza en todo momento, mejorando notablemente la respuesta de la tarea AP1, la cual, en este caso, nunca espera al procesador.



$P : (T = D = 10ms, C = 4ms, Prio = 12)\ E : (T = D = 30ms, C = 0{,}5ms, Prio = 10)$

Figura 2.12: Derivación de la Holgura en Presencia de Tareas Esporádicas. Método Propuesto

2.5. Resumen y Conclusiones

En este capítulo se han revisado los algoritmos de extracción de holgura dinámicos (DSSA) y de asignación dual de prioridades (DP). Se ha descrito su funcionamiento básico así como su comportamiento en presencia de recursos compartidos, variación entre activaciones, tareas esporádicas y plazos arbitrarios.

Los algoritmos DSSA y DP tienen en esencia el mismo objetivo, ofrecer un trato preferente a las tareas aperiódicas cuando hay holgura disponible en el sistema. Difieren en la forma en que detectan y aplican esta holgura. El algoritmo DSSA detecta la holgura de forma directa, analizando para una tarea dada a partir del estado actual del sistema qué holgura existe hasta que vence el plazo para dicha tarea. El algoritmo DP hace disponible la holgura a las tareas aperiódicas de forma indirecta, introduciendo el concepto de promoción. Una tarea ofrece su holgura a las tareas aperiódicas reduciendo su prioridad hasta que es promocionada, momento en que su prioridad vuelve a su valor normal.

El algoritmo DP supera al algoritmo DSSA en varios aspectos. Su definición es más sencilla y su implementación es también más eficiente. Su propia definición permite recuperar como holgura la interferencia no producida por tareas esporádicas. Es aplicable, de forma eficiente, cuando las tareas tienen plazos arbitrarios, mientras que en el algoritmo DSSA el coste de contemplar plazos arbitrarios es muy elevado.

El algoritmo DSSA es superior al algoritmo DP en la cantidad de holgura que puede detectar, tal como queda patente en los análisis realizados en [20] y [26]. También el coste de implementación para recuperar el tiempo de cómputo no utilizado es menor, aunque bastante significativo.

En este capítulo se han resuelto dos deficiencias del algoritmo DSSA.

En primer lugar, se ha propuesto un nuevo método de asignación de prioridades que

permite hacer totalmente compatible la utilización de recursos compartidos entre tareas periódicas y aperiódicas. Utilizando este método y, considerando a las tareas aperiódicas como críticas en lo relativo a la utilización de recursos, se ha demostrado que los protocolos de herencia de prioridad mantienen sus propiedades.

Esencialmente, la asignación de prioridades propuesta consiste en que cada tarea crítica utiliza dos prioridades, una alta y otra baja, mientras que las tareas aperiódicas utilizan una prioridad intermedia. Cuando hay holgura en el sistema, todas las tareas críticas se ejecutan con su prioridad baja, mientras que utilizan su prioridad alta cuando no hay holgura disponible. De esta forma, las tareas aperiódicas se ejecutan con preferencia cuando existe holgura en el sistema. En el momento en que una tarea crítica hereda otra prioridad, como consecuencia de la aplicación de un protocolo de herencia de prioridades, esta prioridad heredada se mantiene con independencia de la holgura del sistema, siendo ésta la clave que permite el correcto funcionamiento de los protocolos de herencia de prioridades en presencia de tareas aperiódicas. También se ha mostrado que este nuevo método de asignación de prioridades puede generar poca sobrecarga, ya que puede ser implementado de forma eficiente.

En segundo lugar, se ha propuesto un nuevo método de derivación de holgura que armoniza de forma sencilla la coexistencia de tareas periódicas y esporádicas. Este método consiste en calcular la holgura de una tarea cuando ésta se activa, y en asignar una holgura igual a infinito cuando la tarea finaliza. De esta forma, la holgura de una tarea esporádica inactiva es igual a infinito, y no condiciona la holgura global del sistema. En el método de derivación original, por contra, la holgura de una tarea esporádica inactiva puede hacerse cero de forma indefinida, condicionando así la holgura global del sistema.

En el capítulo 3 se aborda la posibilidad de incorporar estos algoritmos al lenguaje Ada y a los servicios POSIX, con el fin de evaluar si es posible adaptar estos algoritmos a estos estándares de una forma coherente y adecuada.

Capítulo 3

Nuevas Interfaces Ada y POSIX

En este capítulo se presentan nuevas interfaces para el lenguaje Ada y los servicios POSIX. El objetivo de estas interfaces es ofrecer al programador la posibilidad de utilizar las políticas de planificación basadas en la extracción de holgura y en la asignación dual de prioridades.

Los algoritmos en los que se basan estas políticas han sido estudiados en detalle en el capítulo 2. El objetivo principal de este capítulo consiste en evaluar si es posible diseñar interfaces adecuadas para que el programador pueda utilizar dichas políticas. Para considerar una interfaz como adecuada nos hemos basado en los siguientes criterios:

- Sencillez de uso.

  Los servicios definidos por estas nuevas interfaces deberían ser lo más sencillos posibles desde la perspectiva de su utilización por parte del programador. El número de nuevos servicios introducidos debería ser reducido. Es también importante que estos servicios resulten familiares al programador, por lo que tendrán que seguir los principios de diseño de las interfaces existentes.

- Compatible con la funcionalidad existente.

  Los servicios introducidos por estas interfaces deberían ser compatibles con los servicios actualmente existentes. Durante el diseño de estas nuevas interfaces cualquier servicio que ocasione incompatibilidades con otros servicios ya existentes debería ser reconsiderado.

- Posibilidad de implementar la interfaz Ada a partir de la interfaz POSIX.

  La definición de los servicios de tiempo real del lenguaje Ada están claramente orientados a poder ser implementados sobre una interfaz POSIX. De hecho, mu-

chas implementaciones de sistemas de compilación Ada realizan una conversión de servicios Ada en servicios POSIX. El diseño de las nuevas interfaces debe tener por tanto en cuenta este criterio.

- Implementación sencilla y eficiente.

  La definición de estas interfaces debe facilitar que su implementación debe ser sencilla y eficiente. Incorporar nuevos servicios a un sistema existente supone su modificación, y ésta debe ser por tanto lo más sencilla posible. Además, dado que el ámbito de aplicación de estas interfaces es la construcción de sistemas de tiempo real, la eficiencia de los servicios que se incorporen es un factor muy importante.

El diseño de estas interfaces ha sido realizado conjuntamente con su implementación, utilizando para ello el sistema operativo MaRTE OS, y en gran medida podríamos decir que las interfaces propuestas en este capítulo son fruto de su implementación. Durante este proceso se han valorado numerosas alternativas que han sido reconsideradas o incluso descartadas al introducir inconvenientes desde los puntos de vista de su uso e implementación.

La estructura de este capítulo es la siguiente:

Previamente al diseño de las nuevas interfaces, se revisan en las secciones 3.1 y 3.2 los aspectos más relevantes de los estándares Ada y POSIX relativos a la programación de sistemas de tiempo real. En la sección 3.3 se presenta perspectiva general de estas nuevas interfaces, las cuales se describen con detalle en las siguientes secciones. En la sección 3.5 se estudia cómo introducir en la interfaz la noción de activación periódica o esporádica, necesaria dado el funcionamiento de las políticas introducidas. En la sección 3.6 se describe la política de planificación basada en la extracción de holgura. En la sección 3.7 se describe la política basada en la asignación dual de prioridades. En la sección 3.8 se analiza la problemática de realizar cambios de forma dinámica en los parámetros de planificación. En la sección 3.9 se describen los servicios que permiten diseñar tareas con partes opcionales. Cada una de estas secciones trata una parte de las interfaces desarrolladas y siguen la misma estructura. En primer lugar se describen las interfaces para el lenguaje Ada y los servicios POSIX. A continuación, se discute cómo utilizar los servicios POSIX descritos para construir con ellos la interfaz Ada. Finalmente, se tratan diversos aspectos relacionados con esta parte de la interfaz, tales como ejemplos y recomendaciones de uso, discusión de alternativas, justificación de la interfaz o eficiencia de implementación.

3.1. El Lenguaje de Programación Ada

El estándar actual del lenguaje Ada, comúnmente conocido como Ada 95, es un descendiente directo del estándar Ada 83 [1], cuyo diseño fue promovido por el Departamento de Defensa de los Estados Unidos con el objetivo de disponer de un lenguaje apropiado para la construcción de sistemas empotrados.

Después de varios años de utilización de Ada 83, se tomó la decisión de revisar el lenguaje. La experiencia con el uso de Ada83 mostraba que este lenguaje se estaba utilizando en otros muchos contextos diferentes al original, lo que ocasionó nuevos requerimientos para el lenguaje, tales como incorporar aspectos de programación orientada a objetos. En el campo más específico de los sistemas empotrados, se habían producido importantes avances en el entendimiento de este tipo de sistemas, avances que mostraban que el planteamiento original de Ada 83 no era el adecuado para la construcción de sistemas de tiempo real. Tras un largo proceso de valoración de requerimientos y sus soluciones, finalmente se propuso la nueva versión del lenguaje, que fue denominada Ada 95 [2, 64].

Ada 95 está formado por un núcleo, fácilmente implementable sobre cualquier tipo de arquitectura, y un conjunto de anexos especializados. Todos los compiladores deben implementar el núcleo del lenguaje y cero o varios anexos especializados. Estos anexos son los siguientes:

- Programación de Sistemas, donde se tratan aspectos de acceso a código máquina, tratamiento de interrupciones, requerimientos adicionales de representación de entidades y facilidades para la identificación y asociación de atributos a las tareas.
- Sistemas de tiempo real. En este anexo se introducen la planificación basada en prioridades fijas con desalojo, el protocolo de acceso a los objetos protegidos, requerimientos relativos a la finalización de las tareas, la simplificación del soporte en tiempo de ejecución, la definición de servicios de tiempo monótono y el control directo de las tareas.
- Sistemas Distribuidos, donde se establece el concepto de partición, se clasifican las unidades de compilación en función de su aplicabilidad a sistemas distribuidos, y se define un mecanismo de comunicación entre particiones.
- Sistemas de Información, donde se incorporan a Ada características generalmente asociadas con los programas COBOL.
- Aspectos Numéricos. En este anexo se definen los tipos numéricos complejos y se establecen requerimientos estrictos para las operaciones aritméticas para tipos de coma fija y flotante.

- Seguridad. En este anexo se definen una serie de características que pueden ser utilizadas para ayudar al análisis de la corrección de programas.

Ada ofrece una funcionalidad muy amplia y bien definida para la construcción de sistemas de tiempo real. Esta funcionalidad aparece tanto en el núcleo del lenguaje como en los anexos especializados de programación de sistemas y sistemas de tiempo real.

En el núcleo del lenguaje se define un conjunto de construcciones sintácticas orientadas a la programación concurrente. La alternativa utilizada en Ada de incorporar a nivel sintáctico el soporte para la programación concurrente es relativamente inusual. En otros lenguajes de programación, este soporte es ofrecido bien por parte del sistema operativo o bien por parte de la biblioteca predefinida del lenguaje. Las construcciones que Ada ofrece en este campo son las siguientes:

- Tareas. Las tareas se definen en Ada como un tipo de datos. Su creación y activación se asocia con la declaración de variables del tipo de datos tarea. Pueden ser creadas de forma dinámica durante la elaboración de las declaraciones. La terminación de una tarea y la espera por parte de su tarea padre se establece en función de los ámbitos sintácticos del programa.

- Comunicación. La comunicación entre tareas está basada en el mecanismo de cita. Una tarea puede declarar varias operaciones visibles denominadas entradas. La cita entre dos tareas se establece cuando una tarea invoca la entrada de otra tarea y esta última acepta la cita. Durante la cita se realiza el intercambio de información.

- Selección múltiple. Permite a las tareas atender varias entradas simultáneamente y controlar qué entrada aceptar en función del estado de la aplicación.

- Objetos Protegidos. Los objetos protegidos se introducen en el lenguaje como tipos de datos cuyas variables pueden ser accedidas desde el programa a través de una interfaz bien definida. Las reglas del lenguaje garantizan que el acceso concurrente a estas variables se realiza en exclusión mutua, garantizando por tanto la consistencia del estado interno de estos objetos. Adicionalmente, las tareas pueden sincronizar su ejecución en función del estado del objeto protegido.

- Transferencia asíncrona de control. Un mecanismo que permite que una tarea abandone la ejecución de un bloque de sentencias en respuesta a un cambio en el estado del sistema o el vencimiento de un plazo temporal.

- Servicios de tiempo. Ada ofrece a la aplicación tipos de datos específicos para el tratamiento de información temporal, servicios de consulta del instante actual y sentencias de espera tanto relativas como absolutas.

El núcleo del lenguaje no exige ninguna política de planificación asociada a la ejecución de las tareas, permitiendo así al compilador escoger la política que pueda resultar más sencilla, haciendo uso, por ejemplo, de la política de planificación que utilice el sistema operativo sobre el que la aplicación se ejecuta. En sistemas de tiempo real estricto la política de planificación es un aspecto fundamental que afecta directamente a la corrección temporal del sistema. Por ello, en el anexo de sistemas de tiempo real se define de forma muy rigurosa la política de planificación utilizada por Ada para aplicaciones de tiempo real, política que coincide con el paradigma de prioridades fijas ampliamente utilizado en la actualidad. En particular, esta política está basada en los siguientes elementos:

- La definición de la política estándar denominada `FIFO_Within_Priorities`, que corresponde con una política de planificación basada en prioridades fijas con desalojo en la que, ante igualdad de prioridad, tiene preferencia la tarea que está más tiempo preparada para ejecutarse.

- La asignación de prioridades fijas a tareas y a manejadores de interrupción mediante directivas del lenguaje y mediante llamadas explícitas de cambio de prioridad.

- El concepto de herencia de prioridad y la distinción entre prioridad base y activa de cada tarea. La prioridad base es la prioridad asignada mediante directivas o cambios explícitos de prioridad. La prioridad activa de una tarea es la mayor de las que tenga heredadas. La prioridad base, la del techo de un objeto protegido y la de otra tarea durante una cita son las posibles causas de herencia de prioridad. La prioridad activa que se determina en cada momento es la que se utiliza para planificar la tarea.

- La definición del protocolo de herencia de prioridad en el acceso a los objetos protegidos. Ada adopta sólo un protocolo de cambio de prioridad que denomina `Ceiling_Locking` y corresponde con el protocolo CSP [57].

- La definición del orden en las distintas colas de espera definidas por el lenguaje, en función de la prioridad activa de cada tarea.

En el anexo de tiempo real también se establecen nuevos tipos de datos para la representación del tiempo, diferentes a los contemplados en el núcleo del lenguaje. Esta nueva representación ofrece valores temporales monótonos no decrecientes, una granularidad más fina y posibilita una implementación eficiente a partir de los recursos que se encuentran habitualmente en el hardware actual.

También en los anexos se contemplan otros servicios orientados a permitir controlar desde programación el comportamiento de las tareas. Estos servicios son los siguientes:

- Identificación de las tareas. Aunque las tareas en Ada son accesibles desde un punto de vista sintáctico como variables, la construcción de planificadores en el nivel de aplicación requiere de un mecanismo uniforme para referenciarlas.
- Atributos de las tareas. Mediante instanciación del paquete genérico de atributos, pueden definirse atributos asociados al conjunto de tareas de una partición.
- Cambio de prioridades. Ada define operaciones que permiten realizar cambios en la prioridad base de las tareas. El efecto de estas operaciones se describe en función de la política de planificación estándar, definiendo de forma detallada su relación con los cambios de prioridad automáticos realizados por el mecanismo de herencia de prioridades.
- Control síncrono de las tareas. En este apartado se introduce el concepto de semáforo binario privado, un mecanismo elemental de sincronización que puede ser utilizado para la construcción de mecanismos más generales o complejos.
- Control asíncrono de las tareas. Se contempla la posibilidad de controlar de forma asíncrona la ejecución de las tareas. El mecanismo adoptado consiste en suspender y reactivar tareas utilizando la prioridad conceptual de suspensión.

En la actualidad el estándar Ada se encuentra en fase de revisión [40], con el objetivo de disponer de una nueva versión en el año 2005. Las mejoras previstas relativas a sistemas de tiempo real incluyen la inclusión del perfil de tiempo real `Ravenscar` para sistemas de alta integridad [19] y los relojes de consumo de procesador [30].

3.2. La Interfaz de Aplicación POSIX

La familia de estándares POSIX surgió para intentar unificar las diferentes versiones de sistemas operativos Unix existentes. POSIX es un estándar IEEE cuya referencia es IEEE 1003, y también es un estándar internacional, en este caso con referencia ISO/IEC9945.

POSIX es una familia de estándares en constante evolución. En 1988 se aprobó el primer estándar denominado 1003.1, estándar que fue posteriormente modificado en 1990. Su principal objetivo fue estandarizar los servicios básicos ofrecidos por los sistemas operativos Unix.

Es en 1993 cuando se abordó por primera vez, con la enmienda 1003.1b (Extensiones de tiempo real) [47], la estandarización de servicios orientados a la programación de sistemas de tiempo real. En esta enmienda se definieron interfaces para utilizar semáforos, memoria compartida entre procesos, colas de mensajes, operaciones de entrada-salida

síncrona y asíncrona, políticas de planificación basadas en prioridades fijas con desalojo, señales de tiempo real, relojes y temporizadores.

La posibilidad de programar hilos de ejecución se introduce en POSIX con la enmienda 1003.1c (Hilos) [48] publicada en 1995. Hasta el momento, la única entidad contemplada era el proceso, y existía un convencimiento generalizado de que los hilos de ejecución resultaban ser más convenientes para muchos tipos de aplicaciones. El diseño de una aplicación concurrente suele ser más sencillo cuando se utilizan hilos, ya que comparten el espacio de memoria del proceso. Otro aspecto destacable es la eficiencia en su gestión. En esta enmienda, se definen nuevos servicios de sincronización, tales como los mutex y las variables condición. Las políticas de planificación basadas en prioridades fijas orientadas a procesos fueron trasladadas al ámbito de los hilos. La introducción de los hilos supuso también una revisión profunda del resto del estándar, ya que muchos de los servicios existentes no estaban preparados para asumir la existencia de múltiples hilos en el contexto de un proceso.

Estas enmiendas fueron incluidas en el estándar de base, publicándose una nueva versión del estándar 1003.1 en 1996 [49]. La presencia de hilos y de operaciones de sincronización, la política de planificación basada en prioridades fijas y los servicios temporales supusieron la aceptación de los servicios POSIX como un marco adecuado para la programación de sistemas de tiempo real.

POSIX ha seguido evolucionando, introduciendo nuevos servicios en sucesivas enmiendas. La enmienda 1003.1d (Extensiones de tiempo real adicionales) [52] contempla la activación eficiente de procesos, temporización de la espera en operaciones bloqueantes, la medida y limitación de los tiempos de ejecución y la política servidor esporádico. La enmienda 1003.1j (Extensiones de tiempo real avanzadas) [53] contempla operaciones de sincronización en multiprocesadores, gestión de memoria de diversos tipos y relojes monótono y sincronizado. El acceso a los protocolos de red es tratado por la enmienda 1003.1g, y los servicios de trazas son incorporados por la enmienda 1003.1q [54]. Todas estas enmiendas fueron fusionadas en el estándar de base, formando la última versión del estándar 1003.1 [55], la cual converge con el estándar X/Open [46], el cual constituye otro estándar relevante en el ámbito de los sistemas operativos POSIX.

Los servicios POSIX son muy extensos, y cubren muchos ámbitos de aplicación diferentes. Muchos sistemas de tiempo real no requieren muchos de estos servicios, y, con el fin de clarificar que características son las requeridas por los sistemas de tiempo real, se publicó en 1998 el estándar 1003.13 [50]. En este estándar se definen cuatro perfiles de sistemas de tiempo real, denominados “Sistema de Tiempo Real Mínimo”, “Controlador de Tiempo Real”, “Sistema de Tiempo Real Dedicado” y “Sistema de Tiempo Real Multipropósito”. Para cada uno de estos perfiles se definen los servicios POSIX que deben estar presentes. En la actualidad se encuentra en fase de borrador la revisión de estos perfiles

[56], en la que se pretenden incluir muchos de los servicios incluidos en la última versión de 1003.1.

Tal como puede observarse, la evolución de POSIX hacia los sistemas operativos de tiempo real ha sido muy intensa, lo que muestra la relevancia que los sistemas operativos de tiempo real tienen en la actualidad. Al igual que en el caso del lenguaje Ada, es muy destacable la enorme influencia que han tenido en este proceso los resultados obtenidos por la investigación en el campo de los sistemas de tiempo real, y especialmente, en lo relacionado con la planificación basada en prioridades fijas.

Por último, cabe destacar que la relación entre POSIX y el lenguaje Ada es muy importante. Por una parte, se hace patente que los servicios POSIX pueden servir como soporte en ejecución para los servicios ofrecidos por el lenguaje Ada. De hecho, muchos compiladores transforman las construcciones de alto nivel de Ada en llamadas a servicios POSIX ofrecidos por el sistema operativo subyacente. Por otro lado, existe una interfaz Ada para los servicios POSIX, definida en el estándar 1003.5c [51]. Esta interfaz cubre los servicios definidos en la versión de POSIX 1003.1:1996 junto con la enmienda 1003.1g.

El conjunto de servicios ofrecidos por POSIX y Ada para la programación de sistemas de tiempo real es muy similar. Por ejemplo, la política de planificación es virtualmente idéntica. El acceso a recursos compartidos está basado en mecanismos de exclusión mutua, tales como los objetos protegidos y los mutex, y se utiliza en ambos casos protocolos de herencia de prioridad. Existen servicios de tiempo monótono y operaciones para sincronizar la ejecución de las tareas con instantes de tiempo.

Por otro lado, difieren en varios aspectos. Ada ofrece muchos de sus servicios mediante construcciones sintácticas, mientras que en POSIX los servicios son siempre funciones de biblioteca, ya que POSIX no depende en principio de ningún lenguaje de programación concreto, aunque normalmente sus servicios se definen utilizando el lenguaje C. POSIX ofrece un abanico de servicios más amplio que Ada, en parte porque su evolución ha sido mucho más dinámica. Entre estos servicios cabe destacar la coexistencia de múltiples políticas de planificación, los relojes de tiempo de cómputo, la política basada en el servidor esporádico y servicios de sincronización eficientes en multiprocesadores.

3.3. Perspectiva General

En esta sección se describe de forma general las nuevas interfaces Ada y POSIX diseñadas para las políticas de extracción de holgura y asignación dual de prioridades.

Las políticas basadas en la extracción de holgura y en la asignación dual de prioridades distinguen entre dos tipos de tareas, críticas y aperiódicas. Las políticas presentadas en este capítulo para Ada y POSIX pueden ser utilizadas si el programador asigna una de

estas políticas a las tareas críticas de su aplicación y cualquier otra política (prioridades fijas con desalojo, turno rotatorio, etc.) a las tareas aperiódicas. Las políticas basadas en la extracción de holgura y en la asignación dual de prioridades no han sido diseñadas para coexistir en la misma aplicación, por lo que su utilización conjunta debería evitarse.

Para que estas políticas ofrezcan los efectos deseados, es importante que el programador diseñe su aplicación teniendo en cuenta ciertos requisitos:

- Estas políticas deben aplicarse a tareas que exhiban un comportamiento periódico (o esporádico) y predecible, tal como exigen las teorías desarrolladas en el contexto de la planificación por prioridades fijas con desalojo.
- El conjunto de tareas críticas de la aplicación debe tener garantizados sus requerimientos temporales. Esta garantía es consecuencia de la prioridad asignada a cada tarea y de sus características temporales.
- El uso de recursos compartidos está contemplado por estas políticas, siempre que esté basado en la utilización de los protocolos de herencia de prioridad disponibles.
- Las dos políticas estudiadas no están diseñadas para su uso en sistemas multiprocesadores, por lo que su utilización en este tipo de sistemas debe evitarse.

La coexistencia de múltiples políticas no está contemplada por el lenguaje Ada, por lo que se define una nueva política de planificación general que establece un marco basado en prioridades fijas con desalojo, donde cada tarea puede utilizar una política de planificación individual. Esta política es la siguiente:

```
pragma Task_Dispatching_Policy (Fixed_Priorities);
```

Se define también una nueva directiva que permite asignar políticas individuales a tareas.

```
pragma Individual_Task_Dispatching_Policy (policy);
```

Por último, se definen dos nuevas políticas individuales de planificación, una basada en el algoritmo extractor de holgura y la otra en el algoritmo de asignación dual de prioridades.

```
pragma Individual_Task_Dispatching_Policy (Slack_Policy);
pragma Individual_Task_Dispatching_Policy (Dual_Policy);
```

El estándar POSIX, a diferencia de Ada, sí contempla la coexistencia de múltiples políticas. En este caso, simplemente se definen dos nuevas políticas denominadas `SCHED_SLACK` y `SCHED_DUAL`. Estas políticas se han definido a nivel de hilos POSIX, pero no a nivel de procesos. Esto se ha debido en gran parte a que su implementación se ha realizado en MaRTE OS, sistema que permite la existencia de un sólo proceso, ya que está basado en el perfil POSIX de sistema de tiempo real mínimo [50]. No obstante, la definición de estas políticas a nivel de procesos sería posible y análoga a la definición realizada a nivel de hilos.

Cuando se asigna a una tarea una de estas políticas es necesario especificar un conjunto de parámetros de planificación. Para Ada se definen dos nuevas directivas para establecer estos parámetros:

```
pragma Slack_Policy_Parameters
         (Period, Deadline, Worst_Case_Execution_Time,
          Blocking, Jitter, With_Low_Priority);

pragma Dual_Policy_Parameters
         (Promotion, Low_Priority);
```

En el caso de la extracción de holgura es necesario especificar para cada tarea sus características temporales, ya que el cálculo de la holgura es función de estos valores (sección 2.2). Para la asignación dual de prioridades el parámetro necesario es el tiempo de promoción, el cual es función del tiempo de respuesta de la tarea (sección 2.3).

En POSIX estos parámetros se definen como nuevos componentes de la estructura `sched_param`:

```
// SCHED_SLACK
struct timespec sched_period;
struct timespec sched_deadline;
struct timespec sched_wcet;
struct timespec sched_blocking;
int             sched_withlowprio;

// SCHED_DUAL
struct timespec sched_promotion;
int             sched_lowprio;
```

La correcta asignación de prioridades es uno de los aspectos más importantes cuando se utilizan estas políticas. Una tarea crítica utiliza dos prioridades, denominadas alta y baja. La prioridad alta corresponde con la prioridad que se asigna a una tarea utilizando los servicios ya existentes, tales como la directiva Ada `Priority` o la función POSIX `pthread_setschedparam`.

Las tareas aperiódicas deben utilizar un rango de prioridades inferior al utilizado por las prioridades altas de las tareas críticas. Esta prioridad es asignada igualmente utilizando los servicios ya existentes.

La prioridad baja de cada tarea crítica se establece de forma diferente en cada una de estas dos políticas. Cuando se utiliza la asignación dual de prioridades, es la aplicación quien establece la prioridad baja, mientras que es el sistema quien la establece cuando se utiliza la extracción de holgura.

La decisión adoptada para la asignación dual de prioridades es la que ofrece mayor flexibilidad. La única restricción consiste en que la prioridad baja debe ser inferior a la prioridad alta de cualquier tarea crítica. En cambio, es correcto asignar una prioridad baja que sea igual o superior a la de una tarea aperiódica.

En el caso de la extracción de holgura, la prioridad baja la determina el sistema, garantizando que será inferior a la de cualquier tarea aperiódica de la aplicación. Esto limita las posibilidades de asignar prioridades, pero se optó por este mecanismo para permitir una implementación más eficiente de esta política. En la sección 3.6 se analiza en detalle esta cuestión. Para ofrecer algo más de flexibilidad se permite que la aplicación indique si la tarea crítica dispondrá de prioridad baja o, por el contrario, tan sólo utilizará su prioridad alta. Este último caso puede ser implementado también de forma eficiente y puede ser útil en ciertas ocasiones, tales como el uso de las secciones de cómputo opcional que se describen más adelante.

Cuando se utilizan estas políticas, las tareas críticas alternan entre sus prioridades baja y alta para ofrecer mejores tiempos de respuesta a las tareas aperiódicas. Cuando se utiliza la extracción de holgura, las tareas críticas se ejecutan con su prioridad baja mientras haya holgura en el sistema. Cuando ésta se agota, las tareas críticas cambian a su prioridad alta. En el caso de la asignación dual de prioridades, cuando una tarea crítica se activa utiliza su prioridad baja, y cambia a su prioridad alta cuando se promociona. Las reglas que rigen estos cambios de prioridad se definen en las secciones 3.6 y 3.7.

Ambas políticas requieren que el sistema realice ciertas acciones cuando se activa y cuando finaliza la instancia de una tarea. Por ejemplo, cuando una instancia se activa se calcula su holgura o se determina su instante de promoción, según la política que se esté aplicando. Actualmente no existe ni en Ada ni en POSIX la noción de activación o terminación de una instancia de una tarea, por lo que es necesario introducir un nuevo servicio que cumpla con este requisito. Este servicio, descrito en detalle en la sección 3.5, consiste en notificar al sistema que, cuando la tarea que lo llame se suspenda, se considere que termina su instancia actual, y en consecuencia, cuando vuelva a activarse, que comienza la ejecución de una nueva instancia. Este servicio se define en Ada y POSIX de la forma siguiente:

```
package Ada.Scheduling is
  procedure To_Wait_Activation;
end Ada.Scheduling;


int sched_towait_activation (void);
```

El paquete `Ada.Scheduling` es un nuevo paquete del lenguaje Ada que ofrece nuevos servicios relacionados con la planificación de tareas. El siguiente fragmento de código ilustra la utilización del servicio de notificación de instancia terminada:

```
task body Periodic_Task is
begin
   loop
      Do_Something;
      Next_Activation := Next_Activation + Period;
      Ada.Scheduling.To_Wait_Activation;
      delay until Next_Activation;
   end loop;
end Periodic_Task;
```

Otro de los aspectos abordados en el diseño de estas políticas es el cambio en los parámetros de planificación. Estos parámetros se asignan normalmente en la creación de una tarea pero, bajo ciertas circunstancias, puede ser necesario cambiarlos en tiempo de ejecución, por ejemplo si se produce un cambio de modo. En Ada se definen dos nuevos paquetes, `Ada.Scheduling.Slack_Policy` y `Ada.Scheduling.Dual_Policy` que ofrecen subprogramas para consultar y modificar los parámetros de planificación. En POSIX puede utilizarse la función `pthread_setschedparam` ya definida. Este servicio de cambio de parámetros se describe en la sección 3.8. También se ofrece en esta parte de la interfaz la posibilidad de restablecer la política de planificación, que esencialmente consiste en asignar a todas las tareas críticas su prioridad alta. De esta forma, puede interrumpirse temporalmente el trato preferente a las tareas aperiódicas.

El último servicio diseñado en esta interfaz consiste en ofrecer a las tareas críticas la posibilidad de ejecutar cómputo opcional y se describe en detalle en la sección 3.9. Una tarea crítica debe tener limitado el tiempo que cada una de sus instancias se ejecuta. No obstante, se ha propuesto en diversos trabajos un modelo de tareas en el que cada tarea puede estar formada por varias partes: inicial, opcional y final. Las partes inicial y final se ejecutan siempre y su tiempo de cómputo es conocido y limitado. En cambio, no es necesario que la parte opcional se ejecute siempre, e incluso, tenga un tiempo de ejecución limitado. Esta parte opcional se ejecuta hasta que puede comprometer el plazo de las tareas críticas de la aplicación.

Este servicio se ofrece en Ada mediante el siguiente objeto protegido, definido en el paquete `Ada.Scheduling`:

```
protected Scheduler is
   entry Wait_Slack_Exhausted;
private
   -- definido por la implementación
end Scheduler;
```

Este servicio es común para ambas políticas, aunque su definición es función de la política de la tarea que lo utiliza. La entrada `Wait_Slack_Exhausted` concluye cuando el sistema detecta que no hay holgura disponible o que la tarea que espera en esta entrada se promociona, en función de la política que se utilice. Este servicio, en conjunción con la transferencia asíncrona de control, permite a una tarea ejecutar código opcional que puede ser abortado. Este código puede formar parte de la tarea o ejecutarse en una tarea distinta.

En POSIX este servicio se define de forma muy distinta, ya que no se dispone de la transferencia asíncrona de control. En este caso, se definen las siguientes funciones:

```
int sched_request_optional_execution (void);
int sched_optional_execution_terminated (void);
```

El código ejecutado entre estas dos funciones se considera como opcional por parte del sistema y, cuando éste detecta que no es posible seguir ejecutando cómputo opcional, envía una nueva señal denominada `SIGSLACK` a la tarea.

El resto de este capítulo define con mayor nivel de detalle la interfaz descrita en esta sección.

3.4. Nuevas Políticas de Planificación

3.4.1. Interfaz Ada

Se define un nuevo valor para la directiva `Task_Dispatching_Policy` como sigue:

```
pragma Task_Dispatching_Policy (Fixed_Priorities);
```

Esta política es idéntica a la política `FIFO_Within_Priorities` excepto que se permite asignar políticas individuales para las tareas con la siguiente directiva:

```
pragma Individual_Task_Dispatching_Policy (policy);
```

La política individual por defecto es `FIFO_Within_Priorities`.

Se define una nueva política individual `Slack_Policy` que corresponde con la política basada en la extracción de holgura. Esta política se describe en la sección 3.6.

Se define una nueva política individual `Dual_Policy` que corresponde con la política basada en la asignación dual de prioridades. Esta política se describe en la sección 3.7.

3.4.2. Interfaz POSIX

Se define una nueva política `SCHED_SLACK` que corresponde con la política basada en la extracción de holgura. Esta política se describe en la sección 3.6.

Se define una nueva política individual `SCHED_DUAL` que corresponde con la política basada en la asignación dual de prioridades. Esta política se describe en la sección 3.7.

3.4.3. Comentarios

En Ada sólo existe la posibilidad de disponer de una política de planificación global para todas las tareas de una misma partición. Por contra, las políticas basadas en la extracción de holgura o en la asignación dual de prioridades requieren al menos la existencia de dos políticas, una para las tareas críticas y otra para las tareas aperiódicas. Además, las tareas aperiódicas pueden utilizar cualquier política de planificación (turno rotatorio, tiempo compartido, etc.) siempre y cuando utilicen prioridades inferiores a la de las tareas críticas. Dadas estas consideraciones, en esta interfaz se ha adoptado la propuesta hecha en [4] en la cual se permite la existencia de tareas con políticas de prioridad individuales, dentro de un marco común basado en prioridades fijas con desalojo.

El estándar POSIX, a diferencia de Ada, sí contempla la coexistencia de múltiples políticas. En este caso, simplemente se definen dos nuevas políticas denominadas `SCHED_SLACK` y `SCHED_DUAL`.

3.5. Notificación de Instancia Terminada

La ejecución de una tarea crítica está formada por una sucesión de instancias. Estas instancias pueden activarse de forma periódica o esporádica. La notificación de instancia terminada es un mecanismo que permite a la tarea en ejecución notificar al sistema que su instancia actual concluye.

3.5.1. Interfaz Ada

La interfaz Ada para este nuevo servicio es la siguiente:

```
package Ada.Scheduling is
  procedure To_Wait_Activation;
end Ada.Scheduling;
```

El paquete `Ada.Scheduling` es un nuevo paquete del lenguaje Ada que ofrece nuevos servicios relacionados con la planificación de tareas.

La definición del procedimiento `To_Wait_Activation` es la siguiente:

- Se definen dos nuevos estados para las tareas, *activa* e *inactiva*. Una tarea está *inactiva* cuando espera su próxima activación. Una tarea está *activa* en cualquier otro caso. El cambio entre estos estados está controlado por el procedimiento `To_Wait_Activation`.
- Cuando una tarea *activa* llame al procedimiento `To_Wait_Activation`, pasará a ser una tarea *inactiva* cuando sea eliminada de la cola de tareas preparadas.
- Cuando una tarea *inactiva* sea insertada en la cola de tareas preparadas, se convertirá en una tarea *activa*.

3.5.2. Interfaz POSIX

La interfaz POSIX para este nuevo servicio es la siguiente:

```
int sched_towait_activation (void);
```

Su definición es equivalente a la de la interfaz Ada. La construcción de la interfaz Ada sobre la interfaz POSIX es en este caso directa.

3.5.3. Comentarios

La planificación basada en la extracción de holgura y en la asignación dual de prioridades requieren que el sistema conozca los instantes en los que una tarea crítica, tanto periódica como esporádica, comienza y finaliza una de sus instancias. En el caso de la extracción de holgura, estos instantes son necesarios para calcular la holgura de cada tarea, mientras que en el caso de la asignación dual de prioridades, estos instantes se utilizan para modificar la prioridad de la tarea y así alterne entre la banda alta y baja de prioridades.

Con el fin de que el sistema pueda aplicar las operaciones necesarias cuando una tarea finaliza o reanuda una de sus instancias, se han considerado las siguientes alternativas:

- Definir nuevos servicios para que una tarea espere su próxima activación.
- Considerar que una tarea finaliza su instancia actual siempre que se suspende.
- Modificar los servicios existentes que permiten que una tarea se suspenda para que incorporen una opción que notifique al sistema que la tarea concluye su instancia actual.
- Incluir un servicio que notifique al sistema que la tarea concluirá su instancia actual la próxima vez que se suspenda.

La primera alternativa consitiría en definir servicios que permitieran a una tarea periódica solicitar al sistema detenerse hasta el vencimiento de su próximo periodo o a una tarea esporádica sincronizarse con el evento que la activa. De esta forma, el sistema tendría constancia de los instantes de activación y reanudación de cada tarea. Esta alternativa presenta el inconveniente de tener que diseñar servicios para los cuales ya existe en realidad una solución. Una tarea periódica se construye fácilmente utilizando la sentencia `delay until` en Ada, o temporizadores periódicos, funciones temporizadas o espera temporizada en el caso de POSIX. Las tareas esporádicas pueden ser activadas de forma muy diversa, y tanto en Ada como en POSIX existen servicios apropiados para resolver esta clase de activaciones, tales como llamadas a entradas de objetos protegidos o tareas, señales, variables condición, etc. No parece por tanto oportuno sobrecargar las interfaces Ada y POSIX con nuevos servicios que replicarían la funcionalidad existente.

La segunda alternativa asume que cuando una tarea crítica se suspende, siempre lo hace porque ha finalizado su instancia actual. De esta forma, pasar de preparado a suspendido correspondería para el sistema al instante de finalización y pasar de suspendido a preparado al instante de activación. Esta alternativa es simple y efectiva, pero impone una restricción excesiva al programador. Aunque en principio una tarea crítica no debe suspenderse a sí misma salvo que concluya su activación, posiblemente haya situaciones en la práctica donde esta suspensión sea necesaria. Por ejemplo, en la sección 3.9 se estudia cómo una tarea crítica puede sincronizar su ejecución con una tarea aperiódica y, para ello, la tarea crítica requiere suspender su ejecución antes de finalizar su instancia actual.

La tercera alternativa es similar a la anterior, pero en este caso es el programador quien decide que el sistema considere la acción de suspender a una tarea equivalente a que la tarea haya finalizado su activación. En POSIX, podrían definirse nuevas versiones de las funciones que causan que una tarea se suspenda, por ejemplo, `pthread_cond_wait_finalizing`, `sigwait_finalizing`, etc. En Ada se deberían añadir nuevas sentencias,

tales como `delay until T finalizing`, etc. La modificación del lenguaje Ada o la definición de nuevas versiones de funciones ya existentes en POSIX no son, sin embargo, las soluciones más adecuadas.

Dados los inconvenientes que plantean las alternativas anteriores, se considera la cuarta alternativa como la más adecuada. Con ella se incorpora un único servicio, no se modifica el lenguaje Ada ni ninguna función POSIX existente, y es compatible con todos los servicios por los cuales una tarea puede esperar su próxima activación, sea esta periódica o esporádica.

El siguiente fragmento de código ilustra la utilización del procedimiento `To_Wait_Activation`.

```
task body Periodic_Task is
begin
   loop
      Do_Something;
      Next_Activation := Next_Activation + Period;
      To_Wait_Activation;
      delay until Next_Activation;
   end loop;
end Periodic_Task;
```

De esta forma, cuando la tarea se suspenda en la sentencia `delay until`, el sistema entenderá que la tarea concluye su instancia actual.

Un aspecto controvertido de la definición del procedimiento `To_Wait_Activation` consiste en que si una tarea no se suspende entre una instancia y la siguiente, el sistema considera ambas instancias como una sola. Esto puede suceder, por ejemplo, si las tareas tienen plazos superiores a su periodo o si una tarea pierde su plazo.

Cuando se utiliza la extracción de holgura los plazos deben ser inferiores al periodo, puesto que es inviable calcular la holgura asumiendo lo contrario. Si una tarea pierde su plazo, su holgura será igual a cero, y por tanto no podrán ejecutarse tareas aperiódicas. De esta forma, la pérdida de un plazo sólo afecta a la tarea que lo pierde y a tareas menos prioritarias, tal como sucedería si no se aplicase la extracción de holgura.

La asignación dual de prioridades admite tareas con plazos superiores al periodo, pero esta característica no se utiliza tal como se ha definido esta interfaz. En cualquier caso, no se genera ningún problema, puesto que si una tarea no termina antes de que venza su periodo estará promocionada, y no podrá ejecutarse ninguna tarea aperiódica.

Para permitir que una instancia se active mientras la anterior todavía está en ejecución, la definición de este procedimiento debería incluir la siguiente regla:

- Cuando una tarea *activa* llame al procedimiento `To_Wait_Activation`, la próxima

vez que la tarea llame a una operación potencialmente bloqueante y no se bloquee, pasará a ser una tarea *inactiva* e, inmediatamente, a ser de nuevo una tarea *activa*.

El inconveniente de esta definición alternativa es que su implementación es más compleja. En particular, haría necesario modificar el código de cualquier operación bloqueante para detectar si la tarea ha notificado que concluye su activación. En la definición propuesta basta con modificar el código del planificador en los puntos en los que una tarea se suspende y se reactiva.

Finalmente se optó por no incluir esta regla, dado que su implementación es compleja y no se ocasiona ningún problema severo si no se aplica.

3.6. Planificación Basada en la Extracción de Holgura

3.6.1. Interfaz Ada

La política de planificación basada en la extracción de holgura permite al programador realizar aplicaciones con dos clases de tareas, críticas y aperiódicas. La ejecución de las tareas críticas está sujeta a un conjunto de restricciones temporales, mientras que las tareas aperiódicas no están sujetas a este tipo de restricciones. El objetivo de esta política consiste en ofrecer un buen tiempo de respuesta para las tareas aperiódicas sin comprometer las restricciones temporales de las tareas críticas. Esta política se introduce en el lenguaje Ada mediante un nuevo valor de la directiva `Individual_Task_Dispatching_Policy`.

```
pragma Individual_Task_Dispatching_Policy (Slack_Policy);
```

Las tareas cuya política es `Slack_Policy` deben definir un conjunto de parámetros de planificación utilizando la siguiente directiva.

```
pragma Slack_Policy_Parameters
  (Period, Deadline, Worst_Case_Execution_Time,
   Blocking, Jitter, With_Low_Priority);
```

De esta forma, a una tarea se le asocian cinco parámetros temporales que corresponden con su periodo, plazo, tiempo de ejecución en el peor caso, factor de bloqueo y variación entre activaciones. El tipo de estos parámetros debe ser `Ada.Real_Time.Time_Span`. El sistema define para cada tarea que utiliza esta política un parámetro denominado `Prioridad_Alta`, que corresponde con la prioridad especificada por la directiva `Priority` o, con el valor de la prioridad asignada por la última llamada realizada al procedimiento `Ada.Dynamic_Priorities.Set_Priority`

El parámetro `With_Low_Priority`, cuyo tipo debe ser `Boolean`, permite distinguir entre dos clases de tareas, aquellas que cambian su prioridad como consecuencia de la política basada en la extracción de holgura y las que no cambian de prioridad. Adicionalmente, para las tareas que pueden cambiar de prioridad, el sistema define para cada una de ellas otro parámetro denominado `Prioridad_Baja`. Este parámetro corresponde a la prioridad que utiliza la tarea cuando hay holgura disponible en el sistema. El sistema determina este valor de prioridad de forma automática, garantizando que sea menor que la prioridad de cualquier otra tarea que no utilice la política `Slack_Policy`, y que la relación entre las prioridades `Prioridad_Alta` y `Prioridad_Baja` de las tareas sea la misma.

Las reglas de planificación que se aplican a las tareas cuya política es `Slack_Policy` son análogas a las de la política estándar `Fifo_Within_Priorities`, junto con un conjunto de reglas adicionales que se describen a continuación.

El sistema define para cada tarea cuya política es `Slack_Policy` un parámetro denominado `Holgura`. Este parámetro representa la cantidad de tiempo que una tarea crítica puede retrasar su ejecución sin perder su plazo. Las reglas para calcular el atributo `Holgura` de cada tarea son las siguientes:

- Cuando una tarea es creada, se asigna al atributo `Holgura` de dicha tarea el valor 0.

- En el instante en que una tarea cuya política es `Slack_Policy` se hace activa, se asigna al parámetro `Holgura` de dicha tarea el valor de $S_i(t)$, calculado según la siguiente ecuación.

$$S_i(t) = \left(e - t - \sum_{\forall j \in hep(i)} I_j(t,e) - B_i \right)_0 \tag{3.1}$$

$$I_j(t,e) = c_j(t) + f_j(t,e)C_j + min(C_j, (e - x_j(t) - f_j(t,e)T_j)_0) \tag{3.2}$$

$$f_j(t,e) = \left\lfloor \frac{(e - x_j(t))_0}{T_j} \right\rfloor \tag{3.3}$$

i	La tarea para la cual se calcula la holgura
t	Instante en que la tarea i se hace activa
e	El instante t incrementado con el valor del parámetro `Deadline` de la tarea
$hep(i)$	Conjunto de tareas con prioridad igual o superior a la tarea i (incluye a la tarea i)
B_i	Valor del parámetro `Blocking` de la tarea i
C_j	Valor del parámetro `Worst_Case_Execution_Time` de la tarea j
T_j	Valor del parámetro `Period` de la tarea j
$c_j(t)$	Si la tarea j está inactiva, este término es igual a cero. Si la tarea j está activa, es igual al parámetro `Worst_Case_Execution_Time` menos el tiempo de ejecución consumido por la tarea j desde que se hizo activa por última vez respecto al instante actual t
x_j	Instante en que la tarea j se hizo activa por última vez respecto al instante actual t, incrementado con el valor del parámetro `Period` de la tarea j, y decrementado con el valor `Jitter` de la tarea j.

- En el instante en que una tarea cuya política es `Slack_Policy` se hace inactiva, se asigna al parámetro `Holgura` de dicha tarea el valor ∞.

- Cuando una tarea cuya política es `Slack_Policy` deja de ser la tarea en ejecución se realiza una operación de actualización de la holgura para todas las tareas cuya política es `Slack_Policy` y su parámetro `Prioridad_Alta` es igual o superior, exceptuando a la tarea que ha dejado de ser la tarea en ejecución.

- Cuando una tarea cuya política es `Slack_Policy` se convierte en la tarea en ejecución y la política de la tarea en ejecución anterior es diferente a `Slack_Policy` o el procesador estaba ocioso, se realiza una operación de actualización de la holgura para todas las tareas cuya política es `Slack_Policy` del sistema, incluida la tarea que se ha convertido en la tarea en ejecución.

- Una operación de actualización de la holgura consiste en determinar el tiempo transcurrido desde que se realizó la última operación de actualización de la holgura y el instante actual. Este tiempo transcurrido se utiliza para decrementar el parámetro `Holgura` de la tarea. Si como consecuencia de esta acción, el parámetro `Holgura` se hiciese negativo, se le asignaría el valor cero.

Las tareas que pueden cambiar de prioridad (parámetro `With_Low_Priority` igual a `True`) pueden ver disminuida su prioridad con el fin de dar preferencia a tareas menos prioritarias (esta disminución no afecta a las tareas cuyo parámetro `With_Low_Priority` es igual a `False`). Las reglas que controlan los cambios de prioridad para estas tareas son las siguientes:

En primer lugar, se define el parámetro del sistema `Holgura_Disponible`, cuyo valor es igual al menor valor de los parámetros `Holgura` de todas las tareas cuya política es `Slack_Policy`.

- Si como consecuencia de la aplicación de una de las reglas anteriores, el parámetro `Holgura_Disponible` cambia de cero a positivo, la prioridad base de cada tarea, cuya política sea `Slack_Policy` y su parámetro `With_Low_Priority` sea igual a `True`, se hace igual a su prioridad `Prioridad_Baja`.

- Si como consecuencia de la aplicación de una de las reglas anteriores, el parámetro `Holgura_Disponible` cambia de positivo a cero, la prioridad base de cada tarea, cuya política sea `Slack_Policy` y su parámetro `With_Low_Priority` sea igual a `True`, se hace igual a su prioridad `Prioridad_Alta`.

- En todo momento el tiempo de ejecución de cualquier tarea, exceptuando las tareas cuya política sea `Slack_Policy` y su parámetro `With_Low_Priority` sea igual a `True`, está limitado al valor del parámetro `Holgura_Disponible`. Cuando se alcanza este límite, la prioridad base de cada tarea con política `Slack_Policy`, cuyo parámetro `With_Low_Priority` es igual a `True`, se hace igual a su prioridad `Prioridad_Alta`. El parámetro `Holgura` se hace cero para la tarea o tareas cuyo parámetro `Holgura` sea el menor de los parámetros `Holgura` de todas las tareas cuya política es `Slack_Policy`, quedando por tanto el parámetro `Holgura_Disponible` igual a cero.

3.6.2. Interfaz POSIX

En POSIX se define una nueva opción denominada `POSIX_THREAD_SLACK_SCHEDULING`. Si esta opción está implementada, existe una nueva política de planificación denominada `SCHED_SLACK` y nuevos miembros de la estructura `sched_param`.

```
struct timespec sched_period;
struct timespec sched_deadline;
struct timespec sched_wcet;
struct timespec sched_blocking;
struct timespec sched_jitter;
int             sched_lowprio;
```

Estos miembros corresponden a los parámetros que la interfaz Ada define para las tareas críticas. La descripción de está política es análoga a la descrita en la interfaz Ada. Los efectos de utilizar está política con la función `pthread_setschedparam` están descritos en la sección 3.8, en la cual se define cómo se realizan cambios en los parámetros

de planificación de forma dinámica.

3.6.3. Comentarios

Utilización de la Política Basada en la Extracción de Holgura

La utilización de esta política supone la siguiente metodología por parte del programador de aplicaciones de tiempo real.

Las tareas críticas, periódicas o esporádicas, deberían utilizar la política basada en la extracción de holgura, asignando a estas tareas sus parámetros temporales de periodo, plazo, tiempo de cómputo en el peor caso, factor de bloqueo y variación entre activaciones. Estos parámetros son necesarios para garantizar la planificabilidad de la aplicación, por lo que es razonable suponer que el programador disponga de estos valores. Dada la definición de esta política, las tareas críticas deberían utilizar el servicio de notificación descrito en la sección 3.5. Aquellas tareas consideradas como aperiódicas utilizarían una política diferente a la extracción de holgura. Estas tareas no necesitan utilizar ninguno de los servicios introducidos en esta interfaz. Para que el sistema pueda garantizar que las tareas críticas no pierden sus plazos aplicando esta política, la prioridad de cualquier tarea crítica debería ser mayor que la de cualquier tarea aperiódica.

El siguiente fragmento de programa muestra la utilización de esta política en el lenguaje Ada.

```
pragma Task_Dispatching_Policy (Fixed_Priorities);

task Periodic_Task is
   pragma Individual_Task_Dispatching_Policy (Slack_Policy);
   pragma Priority (High);
   pragma Slack_Policy_Parameters
      (Period, Deadline, Worst_Case_Execution_Time,
       Blocking, Jitter, True);
end Periodic_Task;

task body Periodic_Task is
begin
   loop
      Do_Something;
      Next_Activation := Next_Activation + Period;
      To_Wait_Activation;
      delay until Next_Activation;
   end loop;
end Periodic_Task;

task Aperiodic_Task is
```

```
      pragma Priority (Low);
   end Aperiodic_Task;

   task body Aperiodic_Task is
   begin
      loop
         Wait_Something;
         Do_Something;
      end loop;
   end Aperiodic_Task;
```

Asignación de Prioridades

La política basada en la extracción de holgura requiere que cada tarea crítica utilice dos prioridades, una prioridad alta cuando no hay holgura en el sistema y otra baja cuando si la hay. La prioridad baja de estas tareas debe ser inferior a la de las tareas aperiódicas y además, la relación entre las prioridades alta y baja del conjunto de tareas críticas debe ser la misma, es decir, si una tarea es más prioritaria que otra respecto a la prioridad alta, también lo debe ser respecto a la prioridad baja.

Para garantizar esta restricción, hay dos alternativas posibles. El programador determina la prioridad baja de la tarea y el sistema verifica que la prioridad asignada es correcta, o bien el sistema es el encargado de determinar la prioridad baja de forma automática. Ya que ambas alternativas introducen cierta complejidad en el sistema, se optó por la segunda, ya que es más sencilla desde el punto de vista del usuario de esta interfaz.

Respecto a la prioridad de las tareas aperiódicas, también se evaluó la necesidad de restringir que esta prioridad fuese menor que la prioridad de cualquier tarea crítica. Se descartó restringir este caso porque hay situaciones en las que una tarea no crítica debe tener una prioridad mayor. Por ejemplo, el soporte en tiempo de ejecución de GNAT utiliza varias tareas internas que se ejecutan a la máxima prioridad, las cuales son parte del sistema y no pueden considerarse como tareas críticas en realidad.

También se planteó la posibilidad de utilizar una asignación de prioridades más flexible, igual a la de la política de asignación dual de prioridades, en la que el programador decide la prioridad baja de cada tarea, con la única restricción de que esta prioridad sea inferior a las prioridades altas de las tareas críticas. Finalmente, esta alternativa fue descartada, a pesar de ser técnicamente posible, por motivos de eficiencia de implementación. En este caso hubiese sido necesario limitar, en función de la holgura del sistema, el tiempo de ejecución de todas las tareas críticas de la aplicación, y no sería posible implementar de forma eficiente la operación de cambio de prioridad de las tareas críticas descrito en la sección 2.4.1.

Operación de Cambio de Prioridad

El cambio entre la prioridad baja y alta de las tareas críticas puede suponer una sobrecarga muy significativa para el sistema, ya que este cambio se realiza para todas las tareas críticas del sistema. No obstante, hay que tener en cuenta que la descripción de una política de planificación no implica que su implementación deba realizarse de la misma forma.

Tal como se describió en la sección 2.4.1 es posible implementar de forma eficiente estos cambios de prioridad. Adicionalmente, la implementación descrita en dicha sección también proporciona un método sencillo para resolver la asignación automática de prioridades.

Tareas que no Cambian de Prioridad

En la sección 3.9 se describen varios servicios que permiten a una tarea solicitar tiempo de ejecución opcional. Con el fin de maximizar el tiempo que el sistema puede ofrecer a la tarea, es conveniente que la tarea comience su ejecución tan pronto como sea posible. Este tipo de tareas pueden declararse como tareas que no cambian de prioridad en la interfaz descrita en esta sección, ya que no disminuyen su prioridad cuando hay holgura en el sistema.

Es importante destacar que cuando una tarea crítica no cambia de prioridad cuando hay holgura en el sistema, es posible que otra tarea crítica más prioritaria sí lo haga. Debido a esto, la ejecución de este tipo de tareas debe estar limitada a la holgura disponible.

Desde el punto de vista de la implementación, si se utiliza la técnica del doble conjunto de colas, estas tareas deben pertenecer al conjunto utilizado por las tareas aperiódicas, dado que las tareas de este conjunto no varían su prioridad en función de la holgura del sistema.

Selección de Funcionalidad

En la sección 2.2 se ha descrito la funcionalidad que la planificación basada en la extracción de holgura puede ofrecer, pero no toda esta funcionalidad ha sido incorporada en la política de planificación descrita en esta sección. En este apartado se describen los criterios utilizados para decidir la parte de esta funcionalidad que se ha considerado adecuada con el fin de incluirla en la política descrita en esta sección.

Es posible utilizar dos algoritmos para calcular la holgura, uno exacto y otro aproximado. El coste temporal del algoritmo exacto es muy elevado, e incluso, difícil de evaluar, puesto que depende en gran medida de la forma en la que las tareas críticas intercalan sus

ejecuciones. El coste temporal del algoritmo aproximado es lineal respecto al número de tareas críticas, por lo que la sobrecarga que introduce en el sistema, aunque significativa, es aceptable. Dado que el algoritmo aproximado ofrece una buena estimación de la holgura a un coste menor y conocido, se ha optado por utilizarlo.

Para aplicar la holgura es posible utilizar dos métodos, calcular la holgura cuando se necesite o calcularla una vez por activación de cada tarea crítica y derivar su valor en cada cambio de contexto. Ambos métodos tienen un coste significativo, pero el coste del primero es realmente excesivo, ya que cada vez que una tarea aperiódica pasa a estar preparada y cuando una tarea crítica finaliza su ejecución, es necesario calcular la holgura para todas las tareas críticas del sistema. Evidentemente, sólo el método de derivación de la holgura puede ser considerado para su aplicación práctica.

El cálculo del plazo efectivo (ecuación 2.11) como paso previo al cálculo de la holgura ha sido descartado. Esto ha sido debido a que el coste de este algoritmo es lineal con el número de tareas, similar por tanto al del cálculo aproximado de la holgura, y es posible que en muchas ocasiones no sirva para mejorar significativamente la aproximación de la holgura real.

El nuevo método de derivación de la holgura expuesto en la sección 2.4.2, que consiste en calcular la holgura cuando una tarea se activa, ha sido adoptado en esta interfaz, ya que permite tratar por igual tanto a tareas periódicas como esporádicas. Otra alternativa hubiera consistido en distinguir entre tareas periódicas y esporádicas o hacer que el sistema detectase cuándo una tarea exhibe un comportamiento esporádico, comparando la diferencia entre activaciones y el periodo. En este caso, la holgura de las tareas periódicas podría calcularse al finalizar su activación, mientras que la de las tareas esporádicas debería calcularse tanto al principio como al final. Esta alternativa fue descartada porque complicaba en exceso la definición de esta interfaz.

Los métodos originales descritos en la sección 2.2 que permiten detectar la interferencia no producida por tareas esporádicas son excesivamente complejos. Esto dificulta tanto su implementación en el sistema como su propia definición en las interfaces Ada y POSIX. Adicionalmente, es difícil modelar la sobrecarga que introducen en el análisis de planificabilidad del sistema. Por todo ello, se ha optado por no incorporar ninguno de estos métodos.

La recuperación del tiempo de cómputo no utilizado es una característica importante de la extracción de holgura. Es bastante probable que una activación consuma menos tiempo del previsto, debido al pesimismo que suele acompañar a la estimación de los tiempos máximos de cómputo. No obstante, esta funcionalidad puede incorporar una sobrecarga muy significativa en el sistema, ya que su coste es lineal respecto al número de tareas menos prioritarias. Por ejemplo, el cálculo de la holgura para la tarea más prioritaria del sistema es trivial, ya que su holgura no depende de otras tareas, mientras que

actualizar el tiempo de cómputo que no utiliza implica actualizar la holgura de todo el resto de tareas del sistema. Por otro lado, si no se recupera este tiempo de cómputo se simplifica notablemente el comportamiento del sistema. En este caso, cuando la holgura de una tarea se hace igual a cero, sólo puede recuperarse cuando finaliza la instancia actual de dicha tarea. Esta propiedad hace que sea más sencillo comprender el funcionamiento de la extracción de holgura y simplifica el análisis de tiempos de respuesta. Debido a estas consideraciones, finalmente se ha optado por no incorporar esta funcionalidad.

En la sección 2.4.1 se han descrito los inconvenientes que plantea la estrategia adoptada para permitir recursos compartidos entre tareas periódicas y aperiódicas, y se ha propuesto una variante del algoritmo extractor de holgura que permite de una forma mucho más sencilla y natural la compartición de recursos. Esta variante ha sido la seleccionada para esta política.

Las técnicas descritas en la sección 2.2 que permiten determinar el bloqueo efectivo han sido también consideradas. Algunas de estas técnicas son muy eficientes, mientras que otras introducen sobrecargas significativas porque requieren monitorizar el tiempo de ejecución de las secciones críticas. En cualquiera de los casos, al permitir que las tareas aperiódicas puedan bloquear la ejecución de las tareas críticas se invalidan los supuestos en que se basan estas técnicas, y por tanto, dejan de ser aplicables. Por ejemplo, estas técnicas asumen que cuando una tarea crítica se activa, si su prioridad es mayor que el techo actual del sistema, se sabe que no podrá ser bloqueada por tareas críticas de menor prioridad. En este caso el factor de bloqueo efectivo para la tarea se haría igual a cero. En cambio, al permitir que las tareas aperiódicas puedan bloquear a las tareas críticas, el bloqueo puede todavía producirse.

3.7. Planificación Basada en la Asignación Dual de Prioridades

3.7.1. Interfaz Ada

La política de planificación basada en la asignación dual de prioridades permite al programador realizar aplicaciones con dos clases de tareas, críticas y aperiódicas. La ejecución de las tareas críticas está sujeta a un conjunto de restricciones temporales, mientras que las tareas aperiódicas no están sujetas a este tipo de restricciones. El objetivo de esta política consiste en ofrecer un buen tiempo de respuesta para las tareas aperiódicas sin comprometer las restricciones temporales de las tareas críticas. Esta política se introduce en el lenguaje Ada mediante un nuevo valor de la directiva `Individual_Task_Dispatching_Policy`.

```
pragma Individual_Task_Dispatching_Policy (Dual_Policy);
```

Las tareas cuya política es `Dual_Policy` deben definir un conjunto de parámetros de planificación utilizando la siguiente directiva.

```
pragma Dual_Policy_Parameters (Promotion, Low_Priority);
```

El parámetro `Promotion`, cuyo tipo debe ser `Ada.Real_Time.Time_Span`, es el tiempo de promoción de la tarea. El parámetro `Low_Priority`, cuyo tipo debe ser `Integer`, es la prioridad de la tarea cuando no está promocionada. Adicionalemnte el sistema el sistema define para cada una de las tareas que utilizan esta política el parámetro `Prioridad_Alta`, que corresponde con la prioridad especificada por la directiva `Priority`, o, con el valor de la prioridad asignada por la última llamada realizada al procedimiento `Ada.Dynamic_Priorities.Set_Priority`, y es la prioridad que se asigna a una tarea cuando está promocionada.

Las reglas de planificación que se aplican a estas tareas son análogas a las de la política estándar `Fifo_Within_Priorities`, junto con un conjunto de reglas adicionales que se describen a continuación.

- En el instante en que una tarea cuya política es `Dual_Policy` se hace inactiva, la prioridad base de esta tarea se hace igual al valor de su parámetro `Low_Priority`. A partir de este instante, a esta tarea se le considera no promocionada. Si la tarea tuviese una operación de promoción pendiente, ésta es cancelada.
- En el instante en que una tarea cuya política es `Dual_Policy` se hace activa, se planifica una operación de promoción para que suceda al cabo de `Promotion` unidades de tiempo. Esta operación consiste en asignar a la prioridad base de la tarea su parámetro `Prioridad_Alta`. A partir del momento en que se realice la operación de promoción, a esta tarea se le considerará promocionada.

3.7.2. Interfaz POSIX

En POSIX se define una nueva opción denominada `POSIX_THREAD_DUAL_SCHEDULING`. Si esta opción está implementada, existe una nueva política de planificación denominada `SCHED_DUAL` y nuevos miembros de la estructura `sched_param`.

```
struct timespec sched_promotion;
int             sched_lowprio;
```

Estos miembros corresponden a los parámetros que la interfaz Ada define para las tareas que utilizan la política `Dual_Policy`. La descripción de está política es análoga a la definida en la interfaz Ada. Los efectos de utilizar está política con la función `pthread_setschedparam` están descritos en la sección 3.8, en la cual se describe cómo se realizan cambios en los parámetros de planificación de forma dinámica.

3.7.3. Comentarios

Utilización de la Política Basada en la Asignación Dual de Prioridades

La utilización de esta política supone una metodología similar descrita para la política basada en la extracción de holgura. Las tareas críticas pueden utilizar la política basada en la asignación dual de prioridades, y en este caso pueden ceder su holgura a tareas aperiódicas. Es igualmente correcto que utilicen la política estándar basada en prioridades fijas ya que entonces equivalen a una tarea cuyo tiempo de promoción es igual a cero. En este caso, estas tareas no ceden su holgura. Aquellas tareas consideradas como aperiódicas utilizarían una política diferente a la asignación dual de prioridades. El programador debe determinar los valores adecuados para los parámetros `Promotion` y `Low_Priority` para que el sistema pueda garantizar que las tareas críticas cumplen sus plazos.

Asignación de Prioridades

La política basada en la asignación dual de prioridades requiere que cada tarea crítica utilice dos prioridades, una prioridad alta cuando está promocionada y otra baja cuando no lo está. Esta política proporciona bastante libertad en cuanto a la asignación de la prioridad baja. El único requisito importante consiste en que la prioridad baja de las tareas críticas debe ser inferior a la prioridad alta de cualquier tarea crítica de la aplicación. No es necesario que la prioridad baja guarde relación alguna con la prioridad de las tareas aperiódicas, ni que la relación existente entre las prioridades altas y bajas sea la misma.

En un principio se pensó en establecer una asignación de prioridades más rígida como la que ofrece la distinción entre tareas que cambian o no de prioridad utilizada por la planificación basada en la extracción de holgura. Esta asignación fue descartada porque con la asignación dual de prioridades es sencillo conseguir el mismo efecto. Una tarea que cambia de prioridad correspondería a una tarea cuya prioridad baja es inferior a la de cualquier tarea aperiódica y una tarea que no cambia de prioridad correspondería a una tarea cuya prioridad baja es superior a la de cualquier tarea aperiódica.

Selección de Funcionalidad

La asignación dual de prioridades es una política sencilla, eficiente y totalmente compatible con aspectos importantes como la compartición de recursos o la presencia de tareas esporádicas. Tan sólo un aspecto de esta política se ha considerado inadecuado para su incorporación en esta interfaz, la utilización del tiempo recuperable.

Tal como se ha descrito en la sección 2.3, cuando una tarea se ejecuta parcialmente antes de su instante de promoción, es posible aplazar dicho instante, aumentando así la posibilidad de atender de forma preferente la ejecución de tareas aperiódicas. Para poder implementar esta técnica, es necesario volver a planificar la operación de promoción cada vez que la tarea crítica abandona el procesador, lo que puede hacer aumentar de forma muy significativa el número total de operaciones de promoción realizadas por el sistema. También se vería comprometida la excelente sencillez de esta política, tanto en su definición como implementación. Otro aspecto a tener en cuenta es la utilidad de esta técnica, ya que si una tarea se ha ejecutado parcialmente, el retardo que puede causar en una tarea aperiódica, una vez se ha promocionado, se ve reducido, puesto que el tiempo de cómputo que resta es inferior al máximo posible. Dadas estas consideraciones, se decidió primar la sencillez y la eficiencia frente a la mejora de sus prestaciones, no incorporando esta técnica en la política descrita en esta sección. En la sección 3.9, no obstante, está técnica se utiliza para aumentar el tiempo disponible para la ejecución de partes opcionales. En este caso, cuando una tarea solicita ejecutar cómputo opcional, se vuelve a planificar la operación de promoción pendiente.

El método que permite la utilización del tiempo de cómputo no producido que puede detectarse cuando finaliza la activación de una tarea ha sido también descartado, ya que su coste de implementación es muy elevado, dado que para poder implementar esta técnica es necesario volver a planificar todas las operaciones de promoción pendientes para tareas menos prioritarias cada vez que una tarea finaliza una activación.

3.8. Modificación de Parámetros de Planificación

Durante la ejecución de la aplicación de tiempo real pueden darse circunstancias que hagan necesario modificar los parámetros de planificación de las tareas críticas. Generalmente a estas situaciones se les conoce como un cambio de modo. En esta sección se definen operaciones que permiten al programador modificar los parámetros de planificación.

3.8.1. Interfaz Ada

Para cada política se define un nuevo paquete que contiene subprogramas para modificar y consultar parámetros de planificación, así como para poder restablecer la política de planificación. Estos paquetes son los siguientes:

Para la política `Slack_Policy`

```
package Ada.Scheduling.Slack_Policy is

   procedure Set_Period
     (New_Period : Ada.Real_Time.Time_Span;
      T          : Ada.Task_Identification.Task_Id
                   := Ada.Task_Identification.Current_Task);
   function Get_Period
     (T : Ada.Task_Identification.Task_Id
          := Ada.Task_Identification.Current_Task)
     return Ada.Real_Time.Time_Span;

   procedure Set_Deadline
     (New_Deadline : Ada.Real_Time.Time_Span;
      T            : Ada.Task_Identification.Task_Id
                     := Ada.Task_Identification.Current_Task);
   function Get_Deadline
     (T : Ada.Task_Identification.Task_Id
          := Ada.Task_Identification.Current_Task)
     return Ada.Real_Time.Time_Span;

   procedure Set_Worst_Case_Execution_Time
     (New_WCET : Ada.Real_Time.Time_Span;
      T        : Ada.Task_Identification.Task_Id
                 := Ada.Task_Identification.Current_Task);
   function Get_Worst_Case_Execution_Time
     (T : Ada.Task_Identification.Task_Id
          := Ada.Task_Identification.Current_Task)
     return Ada.Real_Time.Time_Span;

   procedure Set_Blocking
     (New_Blocking : Ada.Real_Time.Time_Span;
      T            : Ada.Task_Identification.Task_Id
                     := Ada.Task_Identification.Current_Task);
   function Get_Blocking
     (T : Ada.Task_Identification.Task_Id
          := Ada.Task_Identification.Current_Task)
     return Ada.Real_Time.Time_Span;

   procedure Set_With_Low_Priority
     (New_Value : Boolean;
      T         : Ada.Task_Identification.Task_Id
```

```
          := Ada.Task_Identification.Current_Task);

   function Get_With_Low_Priority
     (T : Ada.Task_Identification.Task_Id
          := Ada.Task_Identification.Current_Task)
     return Boolean;

   procedure Slack_Policy_Reset;

end Ada.Scheduling.Slack_Policy;
```

Para la política `Dual_Policy`

```
package Ada.Scheduling.Dual_Policy is

   procedure Set_Promotion
     (New_Promotion : Ada.Real_Time.Time_Span;
      T             : Ada.Task_Identification.Task_Id
                        := Ada.Task_Identification.Current_Task);
   function Get_Promotion
     (T : Ada.Task_Identification.Task_Id
          := Ada.Task_Identification.Current_Task)
     return Ada.Real_Time.Time_Span;

   procedure Set_Low_Priority
     (Priority : System.Any_Priority;
      T        : Ada.Task_Identification.Task_Id
                   := Ada.Task_Identification.Current_Task);
   function Get_Low_Priority
     (T : Ada.Task_Identification.Task_Id
          := Ada.Task_Identification.Current_Task)
     return System.Any_Priority;

   procedure Dual_Policy_Reset;

end Ada.Scheduling.Dual_Policy;
```

La aplicación de estos subprogramas está sujeta a las siguientes reglas:

- Los procedimientos `Set_Period`, `Set_Deadline`, `Set_Worst_Case_Execution_Time`, `Set_Blocking`, `Set_Promotion`, `Set_Low_Priority` y `Set_With_Low_Priority` modifican el valor del parámetro de planificación respectivo para la tarea especificada. El nuevo valor tiene validez de forma inmediata y se utilizará en cuanto sea requerido por las reglas de planificación descritas en las secciones 3.6 y 3.7.

- Las funciones `Get_Period Get_Deadline`, `Get_Worst_Case_Execution_Time`, `Get_Blocking`, `Get_Promotion`, `Get_Low_Priority`, `Get_With_Low_Priority` retornan el valor del parámetro de planificación respectivo de la tarea especificada. Este valor corresponde con el valor asignado por la directivas `Slack_Policy_Parameters` o `Dual_Policy_Parameters`, o, en caso de haber sido modificado, con el último valor asignado por alguno de los procedimientos citados en el párrafo anterior.
- Al ejecutar el procedimiento `Slack_Policy_Reset`, al parámetro `Holgura` de todas las tareas que utilizan la política `Slack_Policy` se le asigna el valor cero, y en consecuencia, el parámetro `Holgura_Disponible` se hace también igual a cero.
- Al ejecutar el procedimiento `Dual_Policy_Reset`, todas las operaciones de promoción pendientes para tareas que utilizan la política `Dual_Policy` se efectúan de inmediato.

3.8.2. Interfaz POSIX

En POSIX los atributos de planificación pueden modificarse utilizando la función `pthread_setschedparam` y su valor puede consultarse utilizando la función `pthread_getschedparam`. Las políticas `SCHED_SLACK` o `SCHED_DUAL` pueden ser utilizadas por ambas funciones.

En lo referente a los nuevos miembros introducidos por estas políticas en la estructura `sched_param`, se definen las siguientes reglas:

- La función `pthread_getschedparam` debe retornar el valor asignado a estos miembros por la última llamada a la función `pthread_setschedparam`.
- La función `pthread_setschedparam` establece nuevos parámetros de planificación para la tarea especificada. Estos nuevos valores tienen validez de forma inmediata y se utilizarán en cuanto sean requeridos por las reglas de planificación descritas en las secciones 3.6 y 3.7.
- Cuando a un hilo cuya política sea distinta de `SCHED_SLACK` se le asigne esta política debido a la ejecución de la función `pthread_setschedparam`, al parámetro `Holgura` de dicho hilo se le asignará el valor de cero. En consecuencia, el parámetro `Holgura_Del_Sistema` se hará también igual a cero.
- Cuando a un hilo cuya política sea `SCHED_SLACK` se le asigne otra política debido a la ejecución de la función `pthread_setschedparam`, se volverá a calcular el parámetro `Holgura_Del_Sistema`. La prioridad de dicho hilo será la establecida por la nueva política.

- Cuando a un hilo cuya política sea `SCHED_DUAL` se le asigne otra política debido a la ejecución de la función `pthread_setschedparam`, la prioridad de dicho hilo será la establecida por la nueva política. Si este hilo tiene una operación de promoción pendiente, esta operación será cancelada.

Adicionalmente se definen las siguientes funciones que permiten restablecer las políticas de planificación.

```
int sched_slack_policy_reset (void);
int sched_dual_policy_reset (void);
```

Al ejecutar la función `sched_slack_policy_reset`, al parámetro `Holgura` de todos los hilos que utilizan la política `SCHED_SLACK` se le asigna el valor cero, y en consecuencia, el parámetro `Holgura_Disponible` se hace también igual a cero.

Al ejecutar la función `sched_dual_policy_reset`, todas las operaciones de promoción pendientes para hilos que utilizan la política `SCHED_DUAL` se efectúan de inmediato.

La implementación de la interfaz Ada sobre la interfaz POSIX es directa. Los subprogramas Ada que consultan y modifican los parámetros de planificación pueden construirse utilizando las llamadas POSIX `pthread_getschedparam` y `pthread_setschedparam`. El restablecimiento de la política de planificación es equivalente en ambas interfaces.

3.8.3. Comentarios

En el desarrollo de esta interfaz se ha prestado especial atención a la posibilidad de que la aplicación realice cambios de modo. Un cambio de modo es una acción compleja, en la cual pueden eliminarse tareas e incorporarse otras, así como modificarse las características y requerimientos temporales de las tareas que continúan en el nuevo modo. Existen diversos protocolos de cambios de modo, que permiten analizar las propiedades temporales del sistema mientras se produce el cambio de modo. En [58] puede consultarse una excelente revisión de estos protocolos y su problemática.

Cuando se produce un cambio de modo, tanto la planificación basada en la extracción de holgura como la basada en la asignación dual de prioridades pueden causar serios inconvenientes. Supongamos, por ejemplo, que en una operación de cambio de modo una tarea aumenta su frecuencia de activación, lo que provoca un aumento de la interferencia en las tareas de menor prioridad. Como consecuencia de este aumento de la interferencia, la holgura o el instante de promoción de estas tareas menos prioritarias será menor. El sistema debería recalcular la holgura o los instantes de promoción para dichas tareas con el fin de reflejar correctamente el cambio producido. Dado que este tipo de acciones son muy costosas y difíciles de implementar, se ha preferido optar por ofrecer una interfaz

que facilite al programador prever de forma sencilla el efecto que un cambio en algún parámetro de planificación puede causar en el comportamiento de las tareas, junto con un mecanismo que permite restablecer la política de planificación y garantizar un comportamiento predecible ante un conjunto de cambios en los parámetros de planificación.

Cuando se utiliza la extracción de holgura, aplicando las reglas descritas en la sección 3.6, cualquier cambio en algún parámetro de planificación tendrá lugar en la próxima activación de la tarea. De forma análoga, al usar la asignación dual de prioridades y aplicando las reglas de la sección 3.7, un cambio en el instante de promoción se aplicará también en la próxima activación de la tarea. Si se modifica la prioridad baja de la tarea, la próxima vez que la tarea se haga inactiva se utilizará el nuevo valor.

Dado que normalmente es deseable que una operación de cambio de modo se realice con la mayor inmediatez posible, parece razonable que el programador disponga de un mecanismo que permita cancelar el trato preferente que las políticas de extracción de holgura y asignación dual de prioridades dan a las tareas no críticas. Para ello se ha introducido la posibilidad de restablecer la política de planificación. Otro objetivo de restablecer la política de planificación es permitir que de forma determinista pueda aplicarse un conjunto de cambios. La política de extracción de holgura se restablece anulando la holgura para todas las tareas críticas del sistema. Como consecuencia de esto, el trato preferente a tareas no críticas se reanudará cuando todas las tareas críticas pasen a estar inactivas. Algo similar sucede con la política de asignación dual de prioridades, ya que al promocionarse todas las tareas cuando se restablece esta política, sólo cuando todas las tareas críticas pasen a estar inactivas los mecanismos de la asignación dual de prioridades volverán a aplicarse con normalidad.

Por supuesto, si el programador desea realizar de forma simultánea un conjunto de modificaciones en los parámetros de planificación que afecte a varias tareas, es necesario que la tarea que los realice lo haga con una prioridad lo bastante elevada para no ser interrumpida mientras efectúa estas modificaciones, lo cual es práctica habitual cuando se programa un cambio de modo.

3.9. Secciones de Cómputo Opcional

En esta sección se definen mecanismos que permiten a una tarea ejecutar cómputo opcional. El tiempo de este cómputo opcional estará limitado a la cantidad de holgura disponible en el sistema.

3.9.1. Interfaz Ada

El siguiente objeto protegido, definido en el paquete `Ada.Scheduling`, permite a una tarea sincronizar su ejecución con la holgura existente en el sistema.

```
protected Scheduler is
   entry Wait_Slack_Exhausted;
private
   -- definido por la implementación
end Scheduler;
```

Si el objeto protegido `Scheduler` es utilizado por una tarea cuya política es `Slack_Policy`, el comportamiento de este objeto protegido es el siguiente.

- Una llamada a la entrada `Wait_Slack_Exhausted` es aceptada cuando el parámetro `Holgura` es igual a cero para la tarea que llama a esta entrada o para cualquier otra tarea cuya política sea `Slack_Policy` y su prioridad `Prioridad_Alta` sea igual o inferior.

- Si la llamada a la entrada `Wait_Slack_Exhausted` es aceptada inmediatamente no se produce ningún efecto.

- Si la llamada a la entrada `Wait_Slack_Exhausted` no es aceptada inmediatamente, se realiza una operación de actualización de holgura (ver sección 3.6) para todas las tareas cuya política sea `Slack_Policy` y su prioridad `Prioridad_Alta` sea mayor o igual que la de la tarea que ha realizado la llamada, exceptuando esta tarea. En este momento comienza una sección de cómputo opcional para la tarea que ha realizado la llamada.

- Mientras una tarea esté ejecutando una sección de cómputo opcional, cuando dicha tarea abandone el procesador se actualizará la holgura a todas las tareas cuya política sea `Slack_Policy`, en lugar de sólo a las tareas que tengan una prioridad `Prioridad_Alta` igual o superior.

- Si la llamada a la entrada `Wait_Slack_Exhausted` es cancelada, se realiza una operación de actualización de holgura para todas las tareas cuya política sea `Slack_Policy`. En este momento la sección de cómputo opcional para la tarea que ha hecho la llamada finaliza.

- Cuando la llamada a la entrada `Wait_Slack_Exhausted` es aceptada y no había sido aceptada inmediatamente, la sección de cómputo opcional para la tarea que ha hecho la llamada finaliza.

Si el objeto protegido `Scheduler` es utilizado por una tarea cuya política es `Dual_Policy`, el comportamiento de este objeto protegido es el siguiente.

- Una llamada a la entrada `Wait_Slack_Exhausted` es aceptada cuando la tarea que realiza la llamada está promocionada.
- Si la llamada a la entrada `Wait_Slack_Exhausted` es aceptada inmediatamente no se produce ningún efecto.
- Si la entrada `Wait_Slack_Exhausted` no es aceptada inmediatamente el sistema procede como sigue. La operación de promoción pendiente se anula. Se planifica una nueva operación de promoción utilizando un nuevo instante de promoción. Este nuevo instante de promoción se determina incrementando el instante de promoción que se calculó cuando la tarea se hizo activa con el tiempo de ejecución consumido por dicha tarea desde que pasó a ser activa. En este momento comienza una sección de cómputo opcional para la tarea que ha realizado la llamada.
- Si la llamada a la entrada `Wait_Slack_Exhausted` es cancelada, la sección de cómputo opcional para la tarea que ha hecho la llamada finaliza.
- Cuando la llamada a la entrada `Wait_Slack_Exhausted` es aceptada y no había sido aceptada inmediatamente, la sección de cómputo opcional para la tarea que ha hecho la llamada finaliza.

3.9.2. Interfaz POSIX

Las dos siguientes funciones, junto con una nueva señal denominada SIGSLACK, permiten a un hilo sincronizar su ejecución con su holgura efectiva.

```
int sched_request_optional_execution (void);
int sched_optional_execution_terminated (void);
```

Un hilo puede llamar a la función `sched_request_optional_execution` para notificar al sistema que comienza la ejecución de una sección de cómputo opcional. Un hilo puede llamar a la función `sched_optional_execution_terminated` para notificar al sistema que ha concluido la ejecución de su cómputo opcional.

Cuando estas funciones son llamadas por un hilo que utiliza la política `SCHED_SLACK` su comportamiento es el siguiente.

- La función `sched_request_optional_execution` retorna cero si el hilo que ha efectuado la llamada no está ejecutando una sección de cómputo opcional y el parámetro `Holgura_Disponible` es mayor que cero.

- La función `sched_request_optional_execution` retorna `EAGAIN` si el hilo que ha efectuado la llamada no está ejecutando una sección de cómputo opcional y el parámetro `Holgura_Disponible` es igual a cero.

- Si la función `sched_request_optional_execution` retorna cero, comienza una sección de cómputo opcional para el hilo que ha realizado la llamada a esta función. En este momento se realiza una operación de actualización de holgura (ver sección 3.6) para todos los hilos que utilizan la política `SCHED_SLACK` con prioridad `Prioridad_Alta` mayor o igual que la del hilo que ha efectuado la llamada, exceptuando a este hilo.

- La función `sched_request_optional_execution` no produce efecto alguno si es llamada por un hilo que se encuentre ejecutando una sección de cómputo opcional. En este caso esta función retorna el error `EINVAL`.

- Cuando un hilo que está ejecutando una sección de cómputo opcional llama a la función `sched_optional_execution_terminated` se realiza una operación de actualización de holgura para todos los hilos que utilizan la política `SCHED_SLACK`. En este momento, la sección de cómputo opcional del hilo que ha efectuado la llamada finaliza. En este caso esta función retorna cero.

- Si la función `sched_optional_execution_terminated` es llamada por un hilo que no está ejecutando una sección de cómputo opcional, esta función no produce ningún efecto. En este caso esta función retorna el error `EINVAL`.

- Mientras un hilo está ejecutando una sección de cómputo opcional, cuando dicho hilo abandone el procesador se actualizará la holgura de todos los hilos que utilizan la política `SCHED_SLACK`, en lugar de sólo a los hilos que tienen una prioridad `Prioridad_Alta` igual o superior.

- Si un hilo está ejecutando una sección de cómputo opcional y el parámetro `Holgura` se hace cero para otro hilo con prioridad `sched_priority` igual o inferior, el sistema envía la señal `SIGSLACK` al hilo que está ejecutando la sección de cómputo opcional. En este momento la sección de cómputo opcional de este hilo finaliza.

Cuando estas funciones son llamadas por un hilo que utiliza la política `SCHED_DUAL` su comportamiento es el siguiente.

- La función `sched_request_optional_execution` retorna cero si el hilo que hace la llamada no está promocionado y no está ejecutando una sección de cómputo opcional.

- La función `sched_request_optional_execution` retorna `EAGAIN` si el hilo que hace la llamada ya está promocionado y no está ejecutando una sección de cómputo opcional.
- Si la función `sched_request_optional_execution` retorna cero, el sistema procede como sigue. La operación de promoción pendiente se anula. Se planifica una nueva operación de promoción utilizando un nuevo instante de promoción. Este nuevo instante de promoción se determina incrementando el instante de promoción que se calculó cuando el hilo se hizo activo con el tiempo de ejecución consumido por dicho hilo desde que pasó a ser activo. En este momento comienza una sección de cómputo opcional para el hilo que ha realizado la llamada.
- La función `sched_request_optional_execution` no produce efecto alguno si es llamada por un hilo que se encuentra ejecutando una sección de cómputo opcional. En este caso esta función retorna el error `EINVAL`.
- Cuando un hilo que está ejecutando una sección de cómputo opcional llama a la función `sched_optional_execution_terminated` finaliza la sección de cómputo opcional para dicho hilo. En este caso esta función retorna cero.
- Si un hilo está ejecutando una sección de cómputo opcional y es promocionado, el sistema envía la señal `SIGSLACK` al dicho hilo En este momento la sección de cómputo opcional de este hilo finaliza.
- La función `sched_optional_execution_terminated` no produce efecto alguno si es llamada por un hilo que no se encuentra ejecutando una sección de cómputo opcional. En este caso esta función retorna el error `EINVAL`.

A continuación se describe, en líneas generales, cómo se puede construir la interfaz Ada sobre la interfaz POSIX. La implementación del objeto protegido `Scheduler` sería la siguiente:

```
protected body Scheduler is
   entry Wait_Slack_Exhausted  when True
   is
   begin
      Res := sched_request_optional_execution ();
      if Res = EAGAIN then
         return;
      end if;
      I := Current_Task_Index;
      Exhausted (I) := False;
      requeue Wait_Slack_Exhausted_Private (I) with abort;
   end Wait_Slack_Exhausted;
```

```
      procedure Slack_Exhausted_For (I : Task_Index)
      is
      begin
         Exhausted(I) := True;
      end Slack_Exhausted_For ;

      entry Wait_Slack_Exhausted_Private (for I in Task_Index)
        when Exhausted(I)
      is
      begin
         null;
      end Wait_Slack_Exhausted_Private;
   end Scheduler;
```

Cuando una tarea ejecuta la entrada `Wait_Slack_Exhausted`, se notifica al sistema el inicio de una sección de cómputo opcional. Si en ese instante no se cumplen las condiciones para que esta sección de cómputo opcional pueda comenzar, la entrada finaliza. En caso contrario, la llamada se encola en la entrada privada `Wait_Slack_Exhausted_Private` hasta que el sistema notifique que la sección de cómputo opcional debe terminar, enviando para ello la señal SIGSLACK a la tarea. Esta señal está capturada por la tarea, y cuando se activa su manejador, se ejecuta la entrada `Slack_Exhausted_For`, lo que causa que se acepte la entrada `Wait_Slack_Exhausted_Private`. En este momento se notifica al sistema que la sección de código opcional ha concluido.

La señal `SIGSLACK` está ignorada durante la ejecución normal de la tarea, salvo cuando ésta se suspende al ser encolada en `Wait_Slack_Exhausted_Private`. Una vez la tarea es reactivada, la señal `SIGSLACK` vuelve a quedar ignorada.

Cuando la entrada `Wait_Slack_Exhausted` se utiliza en la sentencia de disparo de una construcción de transferencia asíncrona de control, es posible que la tarea concluya el código abortable de esta construcción antes de que el sistema envíe la señal `SIGSLACK`. Cuando esto sucede, la tarea es eliminada de la cola de espera de la entrada `Wait_Slack_Exhausted_Private` y, por tanto, el código de esta entrada no se ejecuta. Con el fin de notificar al sistema que la sección de cómputo opcional ha concluido, al final del código abortable se incluye una llamada a `sched_optional_execution_terminated`.

3.9.3. Comentarios

Utilización

El soporte para cómputo opcional permite al programador diseñar tareas críticas que ejecutan una parte opcional cuando el sistema dispone de holgura. El tiempo de cómputo

de una parte opcional no debe formar parte del tiempo de cómputo en el peor caso de la tarea crítica, puesto que la parte opcional sólo se ejecuta si hay holgura disponible.

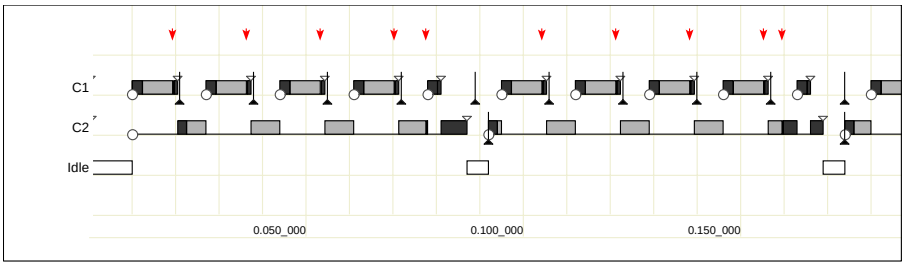
Cuando se utiliza la extracción de holgura, una tarea puede consumir más tiempo de ejecución mientras su holgura y la holgura de tareas menos prioritarias sea positiva. El consumo de tiempo de ejecución de una sección de cómputo opcional debe decrementar la holgura para todas las tareas del sistema, ya que no forma parte del tiempo de cómputo en el peor caso.

De forma análoga, cuando se utiliza la asignación dual de prioridades, una tarea puede consumir más tiempo de ejecución mientras no se promocione. En el momento en que una tarea solicita consumir más tiempo de ejecución, se revisa el instante de promoción para dicha tarea con el fin de maximizar el tiempo disponible para ejecutar la sección de cómputo opcional.

El siguiente fragmento de código ilustra una tarea crítica con tres partes: inicial, opcional y final. El cómputo que realiza la parte opcional no se incluye como tiempo de cómputo en el peor caso de la tarea crítica. Las partes inicial y final tienen garantizado su tiempo de respuesta, mientras que la parte opcional puede ser abortada para que la parte final concluya en su plazo.

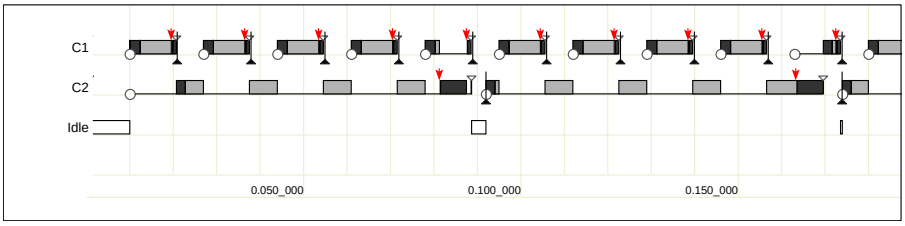
```
task body Critical is
begin
   loop
      Initial_Part;
      select
         Ada.Scheduling.Scheduler.Wait_Slack_Exhausted;
      then abort
         Optional_Part;
      end select;
      Final_Part;

      Next_Activation := Next_Activation + Period;
      To_Wait_Activation;
      delay until Next_Activation;
   end loop;
end Critical;
```

Las figuras 3.1 y 3.2 muestran la ejecución de dos tareas críticas con partes inicial, opcional y final. Las partes inicial y final están representadas con gris oscuro y la parte opcional con gris claro. En la primera figura se utiliza la extracción de holgura y en la segunda la asignación dual de prioridades. En este ejemplo el plazo de la tarea `C1` se ha hecho inferior al periodo con el fin de que la parte opcional de esta tarea no haga uso de toda la holgura disponible, permitiendo así la ejecución de la parte opcional de la tarea



$C1 : (T = 17ms, D = 11ms, C = 3ms)\ C2 : (T = D = 82ms, C = 8ms)$

Figura 3.1: Partes Opcionales (Extracción de Holgura)



$C1 : (T = 17ms, D = 11ms, C = 3ms)\ C2 : (T = D = 82ms, C = 8ms)$

Figura 3.2: Partes Opcionales (Asignación Dual de Prioridades)

`C2`. En la figura 3.1 se utiliza la política de extracción de holgura, y las tareas `C1` y `C2` no utilizan su prioridad baja. En la figura 3.2 se utiliza la asignación dual de prioridades. La prioridad baja de las tareas `C1` y `C2` mantiene la misma relación que la prioridad normal.

Una forma alternativa de diseñar partes opcionales consiste en que éstas se ejecuten en tareas no críticas sincronizadas con las tareas críticas. La ventaja de este método estriba en que las partes opcionales pueden utilizar políticas de planficación diferentes a la de las tareas críticas que solicitan su ejecución. El siguiente fragmento de código es un ejemplo de este método.

```
task body Critical is
begin
 loop
   Initial_Part;
   select
      Ada.Scheduling.Scheduler.Wait_Slack_Exhausted;
      Sync.Stop;
   then abort
      Sync.Start;
      Sync.Wait_Terminated;
   end select;
   Final_Part;
```

```
   Next_Activation := Next_Activation + Period;
   To_Wait_Activation;
   delay until Next_Activation;
 end loop;
end Critical;


task body Optional is
begin
 loop
   Sync.Wait_Started;
   select
      Sync.Wait_Stopped;
   then abort
      Optional_Part;
      Sync.Terminated;
   end select;
 end loop;
end Optional;


protected type Sync is
   procedure Start;
   entry Wait_Terminated;
   procedure Stop;

   entry Wait_Started;
   entry Wait_Stopped;
   procedure Terminated;
private
   Active   : Boolean := False;
   Finished : Boolean := False;
end Sync;


protected body Sync is
   procedure Start is
   begin
      Active   := True;
      Finished := False;
   end Start;

   procedure Stop is
   begin
      Active := False;
   end Stop;

   procedure Terminated is
   begin
```

```
      Finished := True;
   end Terminated;

   entry Wait_Started when Active is
   begin
      null;
   end Wait_Started;

   entry Wait_Terminated when Finished is
   begin
      null;
   end Wait_Terminated;

   entry Wait_Stopped when  not Active is
   begin
      null;
   end Wait_Stopped;
end Sync;
```

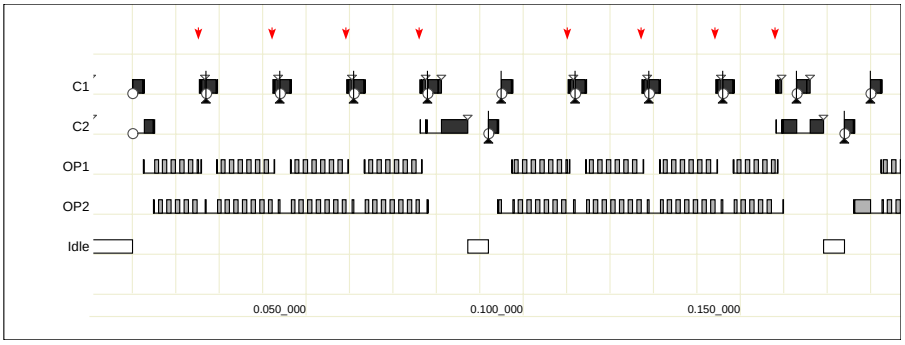
En este ejemplo la tarea crítica y la tarea opcional sincronizan su ejecución utilizando el objeto protegido `Sync`. La tarea crítica activa a la tarea opcional y espera su terminación. Esta espera está limitada a que el sistema determine que esta parte opcional debe terminar. Si se produce esta circunstancia, la tarea crítica aborta la ejecución de la tarea opcional.

Las figuras 3.3 y 3.4 muestran la ejecución de dos tareas críticas `C1` y `C2` que utilizan este método. Las partes opcionales de estas tareas críticas se ejecutan respectivamente en las tareas `OP1` y `OP2`. Estas dos tareas se ejecutan a la misma prioridad con turno rotatorio. En la figura 3.3 se utiliza la política de extracción de holgura, y las tareas `C1` y `C2` no cambian de prioridad. En la figura 3.4 se utiliza la asignación dual de prioridades. La prioridad baja de las tareas `C1` y `C2` mantiene la misma relación que la prioridad normal y es superior a la prioridad de `OP1` y `OP2`.

3.10. Implementación en MaRTE OS

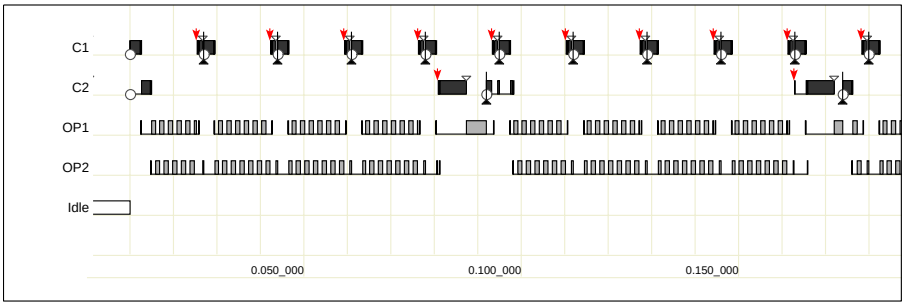
MaRTE OS [3] es un sistema operativo de tiempo real recientemente desarrollado en la Universidad de Cantabria [1]. Ofrece los servicios requeridos por el perfil POSIX de sistema de tiempo real mínimo [50], así como otros servicios tales como la política basada en el servidor esporádico. También incluye servicios propuestos por sus creadores entre los que destacan la planificación y la gestión de interrupciones a nivel de usuario. Puede ser utilizado bajo la licencia GPL.

[1] MaRTE OS está disponible en http://marte.unican.es



$C1 : (T = D = 17ms, C = 3ms)\ C2 : (T = D = 82ms, C = 8ms)$

Figura 3.3: Partes Opcionales en Tareas (Extracción de Holgura)



$C1 : (T = D = 17ms, C = 3ms)\ C2 : (T = D = 82ms, C = 8ms)$

Figura 3.4: Partes Opcionales en Tareas (Asignación Dual de Prioridades)

MaRTE OS está casi totalmente codificado en el lenguaje Ada. Para acceder a los recursos hardware del ordenador se utilizan en ciertas ocasiones los lenguaje C y ensamblador. Las interfaces de acceso a los servicios de MaRTE OS son POSIX C y POSIX Ada. Sobre la interfaz POSIX C se ha adaptado el soporte de ejecución del compilador GNAT [59], con lo cual también se puede acceder a los servicios de MaRTE OS de forma indirecta utilizando el lenguaje Ada.

De MaRTE OS cabría destacar dos aspectos, su diseño y su eficiencia. El código de este sistema está estructurado en un gran número de paquetes Ada cuya funcionalidad está muy bien delimitada, lo que hace que su código sea fácilmente comprensible y modificable. En lo referente a la eficiencia se ha hecho un importante esfuerzo en la selección e implementación de las estructuras de datos más adecuadas, y se ha utilizado un estilo de programación orientado a la eficiencia, aunque sin sacrificar, en momento alguno, la legibilidad del código.

La interfaz descrita en este capítulo ha sido implementada en este sistema operativo, y podríamos decir que esta interfaz es en gran medida consecuencia de su implementación. Cada nueva idea relacionada con la interfaz ha sido programada, y a partir de la experiencia de su implementación, revisada y modificada (o rechazada en más de una ocasión). De esta forma se ha intentado conseguir que la interfaz fuese no sólo funcionalmente adecuada, sino también viable de implementar en la práctica.

Esta implementación se ha realizado en el núcleo del sistema, aunque también se consideró utilizar la interfaz para implementar políticas de planificación a nivel de aplicación que ofrece MaRTE OS, descrita en [5] y [6]. Finalmente se optó por no utilizar esta interfaz, puesto que uno de los objetivos de esta tesis consistía en valorar la eficiencia de la implementación de las políticas basadas en la extracción de holgura y en la asignación dual de prioridades. En gran medida, esta valoración debía basarse en comparar estas políticas con la política estándar definida por Ada y POSIX e implementada en el núcleo de MaRTE OS. La definición de políticas a nivel de implementación introduce una cierta sobrecarga que hubiese distorsionado esta comparación. En cualquier caso, esta interfaz ha sido estudiada y todo parece indicar que técnicamente sería posible implementar con ella estas políticas.

Las principales modificaciones realizadas a MaRTE OS podrían resumirse en los siguientes puntos:

- Modificación del descriptor de tarea para incorporar los atributos de planificación requeridos por las políticas de extracción de holgura y asignación dual de prioridades.

- Modificación de las funciones de asignación y consulta de políticas y parámetros de

planificación, para incorporar las políticas descritas en este capítulo y sus parámetros.

- Dos nuevos paquetes, `Kernel.Slack` y `Kernel.Dual`, encargados respectivamente de implementar las políticas de extracción de holgura y asignación dual de prioridades. En el paquete `Kernel.Slack` se ha implementado también una versión optimizada de la política de extracción de holgura. Esta versión está descrita en la sección 3.6.

- Un nuevo paquete `Kernel.Invocations` encargado de la gestión de las activaciones y terminaciones de las instancias de las tareas. Este paquete atiende el servicio de notificación descrito en la sección 3.5 y notifica las activaciones y terminaciones de las instancias de las tareas a los paquetes `Kernel.Slack` y `Kernel.Dual`.

- Modificación del planificador de MaRTE OS con el fin de notificar al paquete `Kernel.Invocations` que una tarea se hace activa o inactiva. También se ha implementado en el planificador de MaRTE OS el método del doble conjunto de colas descrito en la sección 3.6.

- Se ha introducido un método alternativo de tratamiento de interrupciones del temporizador hardware con el fin de evaluar el impacto de este método en la planificabilidad de las tareas y en la eficiencia del sistema. Este método, que consiste en aplazar el envío de la interrupción del temporizador hardware, se describe con detalle en la sección 5.4.

A nivel del soporte de GNAT se ha implementado la capa que actúa como interfaz entre los subprogramas Ada descritos y los servicios ofrecidos por MaRTE OS. No se ha abordado, en cambio, la modificación del compilador para incorporar las nuevas directivas introducidas por esta interfaz. La asignación de políticas de planificación y sus parámetros se ha resuelto en los programas de prueba desarrollados utilizando llamadas a la interfaz en tiempo de ejecución.

En lo referente a los servicios de cómputo opcional, han surgido diversas complicaciones que han condicionado el realizar una implementación totalmente fiel a la interfaz propuesta.

En primer lugar, el compilador GNAT permite tratar señales enviadas a todo el proceso. Estas señales pueden ser tratadas por entradas de objetos protegidos asociadas a interrupciones. En cambio, este compilador no contempla el envío de señales a tareas individuales. La interfaz que permite programar secciones de cómputo opcional requiere que se envíe la señal `SIGSLACK` a una tarea en particular cuando vence la holgura o se promociona, según la política que se utilice. De esta forma, es necesario modificar el

subsistema del soporte de GNAT encargado del tratamiento de las señales, pero la complejidad de este subsistema es muy elevada. Finalmente se optó por utilizar la funcionalidad existente en GNAT, y enviar la señal `SIGSLACK` a nivel de proceso, señal que es tratada por una entrada privada del objeto `Scheduler`.

Otra dificultad encontrada está relacionada con el uso de la versión POSIX de esta interfaz. El uso de secciones opcionales requiere abortar una sección de código. Esta acción puede programarse utilizando las funciones POSIX `siglongjmp` y `sigsetjmp`, pero se requiere utilizar la función `siglongjmp` desde un manejador de señales. Esto no es posible en MaRTE OS, puesto que en este sistema los manejadores de señales son ejecutados por una tarea interna del sistema, lo que impide realizar un salto desde un manejador al código de la tarea que ha recibido la señal. De esta forma, la interfaz POSIX no ha sido probada desde el nivel de aplicación, y no se ha podido incluir en el análisis de la sobrecarga del sistema realizado en la sección 5.8 el estudio de la interfaz POSIX.

3.11. Resumen y Conclusiones

En este capítulo se han definido nuevas interfaces para el lenguaje Ada y los servicios POSIX. Estas interfaces permiten al programador utilizar las políticas basadas en la extracción de holgura y en la asignación dual de prioridades.

Ambas políticas requieren conocer los instantes en que una tarea inicia y termina una instancia, por lo que se ha introducido en la sección 3.5 el servicio de notificación periódica.

La política basada en la extracción de holgura descrita en la sección 3.6 constituye una notable modificación respecto a su formulación original. El método de asignación de prioridades descrito en la sección 2.4.1 ha sido adoptado con el fin de hacer posible la utilización de esta política junto con los protocolos de herencia de prioridad. De forma análoga, el nuevo método de derivación de holgura descrito en la sección 2.4.2 ha sido utilizado para facilitar la coexistencia de tareas periódicas y esporádicas. Ciertos aspectos han sido descartados al considerar que su funcionalidad no justificaba su complejidad para definirlos o implementarlos. Estos aspectos han sido el cálculo del plazo efectivo, la recuperación del tiempo de cómputo no producido y los factores de bloqueo efectivos.

La política basada en la asignación dual de prioridades descrita en la sección 3.7 responde fielmente a su formulación original. De ella tan sólo se ha descartado la recuperación del tiempo de cómputo no producido por considerar que este aspecto introduce una sobrecarga importante en la implementación. Ha sido utilizado, no obstante, como método para maximizar el tiempo de ejecución de las secciones de cómputo opcionales, ya que en este caso la sobrecarga introducida en el sistema es mucho menor.

En la sección 3.8 se han definido servicios para modificar los parámetros de planificación en tiempo de ejecución. Estos servicios, junto con las definiciones de las políticas descritas en las secciones 3.6 y 3.7, permiten al programador conocer de forma sencilla y determinista el efecto que un cambio en los parámetros de planificación puede ocasionar. También se ha definido un servicio para restablecer la política de planificación. Al restablecer la política, las tareas críticas recuperan su prioridad normal, lo que permite que las tareas aperiódicas no interfieran en una posible operación de cambio de modo. Este restablecimiento permite también abordar de forma sencilla el problema que ocasiona el que los parámetros de planificación de una tarea sean dependientes de los de otras tareas.

En la sección 3.9 se definen servicios de soporte para cómputo opcional, lo que permite diseñar tareas con partes opcionales que pueden ejecutarse cuando el sistema dispone de holgura suficiente. Utilizando estos servicios, el sistema puede notificar a la tarea que está ejecutando una parte opcional que debe concluirla para que no pierda su plazo. Las partes opcionales pueden ser ejecutadas por la propia tarea o por tareas sincronizadas. Utilizar tareas sincronizadas aporta la ventaja de que éstas pueden utilizar cualquier política de planificación.

En la sección 3.10 se ha descrito, en líneas generales, aspectos relativos a la implementación realizada en MaRTE OS de esta interfaz. Esta implementación de la interfaz ha servido para conocer mejor las características y limitaciones de las políticas de extracción de holgura y asignación dual de prioridades, así como para garantizar que fuese viable implementar los servicios descritos.

Valorando las interfaces descritas, pensamos que se ha conseguido que sean sencillas de utilizar por parte del programador. La funcionalidad que ofrecen es elevada y el conjunto de servicios introducidos es relativamente reducido. También se ha intentado que su definición sea sencilla y ante todo precisa. En este aspecto, la definición de la política basada en la extracción de holgura ha resultado algo compleja, dadas las características de esta política.

El desarrollo de la interfaz POSIX ha sido realizado de forma conjunta con el de la interfaz Ada, buscando en todo momento que la interfaz POSIX sirviese tanto para ser usada directamente, como para que sirviera como base para implementar sobre ella la interfaz Ada. En general ambas interfaces son muy similares, excepto en el caso del soporte para partes opcionales. En POSIX este soporte está basado en el envío de señales, mientras que en Ada se utiliza un objeto protegido, pensado para ser utilizado conjuntamente con la sentencia de transferencia asíncrona de control.

Respecto a la compatibilidad con los servicios existentes, entendemos que ésta ha sido conseguida. El diseño de ambas interfaces ha estado condicionado en todo momento por este requerimiento.

En el capítulo 5 se evaluará la eficiencia de las políticas basadas en la extracción de holgura y la asignación dual de prioridades tal como han sido definidas en esta interfaz. En concreto, se analizará el impacto que la sobrecarga del sistema para cada política tiene en la planificabilidad del conjunto de tareas de una aplicación de tiempo real. La medición de la sobrecarga del sistema ha sido realizada utilizando los servicios de trazas POSIX que han sido incorporados a MaRTE OS. Los diversos trabajos que se han realizado en torno a estos servicios se describen en el capítulo 4.

Capítulo 4

Servicios de Trazas POSIX

En este capítulo se describen diversos trabajos realizados en torno a los servicios de trazas POSIX.

El trabajo realizado en esta tesis en torno a las políticas de planificación basadas en la extracción de holgura y en la asignación dual de prioridades se ha centrado fundamentalmente en el diseño de nuevas políticas para el lenguaje Ada y los servicios POSIX, y en el análisis de eficiencia de la implementación en el sistema MaRTE OS de estas políticas.

La implementación de estas políticas se consideró como una parte importante del diseño, con el fin de garantizar que la definición de las políticas facilitase una implementación eficiente y coherente con el resto de servicios ofrecidos por POSIX y Ada. Implementar nuevas políticas de planificación requiere disponer de herramientas adecuadas que sirvan tanto para analizar y comprender su comportamiento, así como para poder depurar posibles errores en su programación.

Por otro lado, el análisis de la eficiencia requiere disponer de un instrumento de medida, capaz de medir el tiempo que consumen las actividades del sistema relacionadas con la gestión de las políticas objeto de estudio.

Una herramienta adecuada a las necesidades expuestas es la obtención de trazas de la actividad del sistema. En una traza se registran los eventos significativos generados por el sistema junto con el instante en que se producen. Esta información se recoge durante la actividad del sistema y puede ser analizada posteriormente, obteniendo a partir de ella cronogramas, que permiten observar el comportamiento de las actividades que se ejecutan en el sistema, así como medir el tiempo de ejecución de estas actividades.

En el año 2000 fue aprobada la enmienda 1003.1q al estándar POSIX 1003.1, en la que se define una interfaz para acceder a los servicios de trazas ofrecidos por el sistema

operativo, aspecto todavía no contemplado por POSIX 1003.1. Dada la existencia de esta nueva interfaz POSIX, se consideró interesante implementar esta interfaz y no otra de carácter propietario. De esta forma, se aportaría al sistema MaRTE OS nueva funcionalidad y se podría valorar la adecuación de estos nuevos servicios de trazas en sistemas de tiempo real. Además de la implementación de estos servicios en MaRTE OS, se optó también por definir una propuesta de interfaz POSIX/Ada a estos servicios y otra interfaz para obtener métricas temporales a partir de las trazas definidas por POSIX.

La estructura de este capítulo es la siguiente:

Los servicios de trazas POSIX son descritos en la sección 4.1. En la sección 4.2.1 se detallan las directrices utilizadas para definir la interfaz POSIX/Ada a estos servicios, interfaz que se presenta en detalle en la sección 4.2.2. La adecuación de los servicios de trazas POSIX para su uso en sistemas de tiempo real se discute en la sección 4.3. En la sección 4.4 se resumen los aspectos más relevantes de la implementación de estos servicios en el sistema MaRTE OS. Por último, en las secciones 4.5.1 y 4.5.2 se aborda el problema de obtener métricas temporales a partir de los eventos almacenados en las trazas definidas por POSIX.

4.1. Introducción a los Servicios de Trazas POSIX

Los servicios de trazas se incorporan a los servicios POSIX con la enmienda 1003.1q [54] y forman parte en la actualidad del estándar POSIX 1003.1:2001 [55]. En esta enmienda se introduce un conjunto de servicios que permiten al programador trazar las acciones que se generan en el sistema causadas por la ejecución de aplicaciones.

Estos servicios están diseñados en torno a los siguientes conceptos:

Trazas *(trace streams)*. Una traza es un objeto no persistente que es capaz de registrar eventos junto con la información necesaria para interpretar dichos eventos.

Eventos *(trace events)*. Un evento es una acción realizada por el sistema que puede ser registrada en una traza. Cada evento es de un cierto tipo definido por el sistema o por la aplicación.

Registros *(trace logs)*. Un registro es un objeto que permite almacenar (o transferir) una traza a un medio persistente, tal como un fichero en disco.

Roles de procesos. Los servicios de trazas POSIX están diseñados en función de un modelo en el cual un proceso puede asumir uno o varios de los siguientes roles: controlador, analizador o trazado.

En líneas generales, los servicios de trazas funcionan de la forma siguiente:

- Cuando un proceso está en ejecución, ciertas acciones que realiza pueden generar eventos. Por ejemplo, el proceso puede realizar una llamada al sistema que tenga asociada la generación de un evento. También es posible que el proceso defina sus propios tipos de eventos, y de forma explícita solicite al sistema de trazas la generación de dichos eventos.

  Hay que resaltar que mientras un proceso no tenga trazas asociadas, los eventos que genera son despreciados por el sistema de trazas y no producen, por tanto, ningún efecto.

- En cualquier momento, un proceso puede crear una traza para otro proceso. A estas trazas se les denomina *activas*. Al proceso que crea la traza se le denomina *controlador*, mientras que al proceso para el cual se crea la traza se le denomina *trazado*.

  En cuanto el proceso controlador active la traza que ha creado, los eventos generados por el proceso trazado irán siendo almacenados en la traza.

- Una vez la acción de trazado se está produciendo, el proceso controlador puede desactivar la traza, volver a activarla, borrar sus eventos, extraer eventos, etc.

- En principio, el proceso controlador es el único que puede acceder a los eventos que se van guardando en la traza, salvo que dicha traza haya sido creada con un registro asociado. Cuando una traza tiene registro asociado, los eventos de la traza se transfieren a un fichero, bien de forma automática o manual. Una vez concluye la acción de trazado, otro proceso al que se le denomina *analizador* puede crear una traza a partir de dicho fichero, y recuperar los eventos previamente almacenados en ella. A este tipo de trazas se les denomina *pre-grabadas*.

A raíz de lo expuesto, se observa que 1003.1q define un modelo claro, sencillo y coherente mediante el cual las aplicaciones pueden hacer uso de los servicios de trazas del sistema. No obstante, a pesar de la relativa sencillez de este modelo, 1003.1q es un estándar extenso y en ocasiones, complejo. Esta extensión y complejidad son fruto de la necesidad de ofrecer un conjunto de servicios que puedan ser utilizados en un amplio abanico de situaciones. Con el fin de comprender mejor el funcionamiento y el alcance de estos servicios, a continuación se desarrollan con más profundidad algunos aspectos de 1003.1q.

Opciones De Implementación

Dado que el conjunto de servicios es extenso, el estándar ofrece varias opciones de implementación, con el fin de que el fabricante del sistema pueda optar por incorporar parte

de estos servicios, en función del uso al que está destinado su sistema. Estas opciones son las siguientes:

_POSIX_TRACE. Esta es la opción que indica que en el sistema se han incorporado los servicios de trazas descritos en 1003.1q. El resto de opciones dependen de ésta. En esencia, el conjunto de servicios que implica esta opción permite al programador utilizar el sistema de trazas tal como ha sido descrito al principio de esta sección, exceptuando la posibilidad de asociar registros a trazas.

_POSIX_TRACE_LOG. Esta opción ofrece la posibilidad de asociar registros a trazas, permitiendo así guardar las trazas en ficheros y analizarlas con posterioridad a la acción de trazado. La implementación de esta opción es relativamente compleja, por lo que puede suponer una carga excesiva, por ejemplo, para sistemas de tiempo real pequeños. En particular, el perfil de sistema de tiempo real mínimo no requiere soporte para sistemas de ficheros de propósito general, lo que permite implementar sólo parte de las características de esta opción.

_POSIX_TRACE_EVENT_FILTER. Con esta opción es posible filtrar qué eventos serán registrados en una traza. La consideración de este servicio como opción es algo discutible, ya que en realidad su implementación no es en absoluto costosa y la utilidad del filtrado de eventos es bastante elevada.

_POSIX_TRACE_INHERIT. Si el sistema implementa esta opción, es posible trazar simultáneamente varios procesos. En particular, permite trazar un proceso junto con toda su descendencia. Es imposible implementar esta opción en los perfiles de sistema de tiempo real mínimo y controlador, dado que en estos perfiles tan sólo se requiere la existencia de un proceso en el sistema.

Gestión de los registros de trazas

Cuando se dispone de la posibilidad de asociar registros a trazas es necesario plantear en qué forma los eventos de la traza serán volcados en su registro. 1003.1q ofrece dos mecanismos no excluyentes de volcado, uno automático y otro manual. El mecanismo automático consiste en eliminar los eventos de la traza cuando ésta se llena y volcarlos en el registro. Este es el funcionamiento por defecto que se establece cuando se crea una traza con registro. Adicionalmente, el proceso controlador de la traza puede volcar su contenido en el registro de forma explícita.

El volcado de eventos en la traza introduce diversas complicaciones que obligan al estándar a no requerir ciertos servicios que podrían ser de difícil implementación. En primer lugar, se plantea la dificultad de permitir al proceso controlador extraer eventos

de la traza para su análisis y garantizar a su vez que estos eventos también se vuelcan en el registro. Dada esta dificultad, el estándar no requiere esta posibilidad. De esta forma, las trazas con registro deberían ser utilizadas tan sólo para crear registros, analizando el contenido de estas trazas con posterioridad. Otro aspecto importante es el tratamiento de los eventos que se producen durante la propia acción de volcado. En este caso, el estándar opta por no requerir que estos eventos sean guardados en la traza y, por tanto, también volcados en el registro.

Espacio de nombres de eventos

Con el fin de facilitar un espacio de nombres lo suficientemente amplio para acomodar los distintos tipos de eventos definidos por el sistema, por bibliotecas y por la propia aplicación, en 1003.1q los tipos de eventos se identifican por cadenas de caracteres. Desde programa, estos tipos de eventos se asocian con identificadores de eventos, existiendo una relación uno a uno entre tipos de eventos y sus identificadores.

Cada proceso posee su propio espacio de nombres de eventos, formado por los eventos del sistema (comunes, por tanto, a todos los procesos) y por los eventos de usuario, definidos por la propia aplicación. Todas las trazas que se asocian a un proceso trazado comparten el mismo espacio de nombres. En el caso de las trazas pre-grabadas, el espacio de nombres de eventos es guardado junto con los eventos en el fichero.

Gestión del desbordamiento

Tanto las trazas como los registros tienen una capacidad de almacenamiento limitada, la cual puede ser definida por el programador. 1003.1q define varias opciones y acciones relativas a la situación de desbordamiento de ambas entidades. El programador puede optar por trazas y registros que eliminan eventos antiguos para almacenar nuevos en caso de desbordamiento, o bien trazas y registros que detienen la acción de trazado. En el caso de los registros, también se permite optar por registros de capacidad ilimitada, lo que normalmente representa que el registro no se almacena en un fichero de datos, sino que es transmitido a algún tipo de dispositivo de comunicaciones.

Análisis concurrente

1003.1q contempla la posibilidad de analizar una traza concurrentemente con la acción de trazado. Para estos casos el estándar contempla funciones que sincronizan la ejecución del proceso controlador con el almacenamiento de eventos en la traza. En particular, un proceso puede suspender su ejecución hasta que se almacene algún evento en la traza y, opcionalmente, temporizar esta espera.

4.2. Interfaz POSIX/Ada para los Servicios de Trazas POSIX

4.2.1. Directrices de Diseño

En el diseño de la interfaz Ada para los servicios de trazas POSIX se han seguido las directrices que fueron aplicadas en el diseño del estándar POSIX.5, con el fin de facilitar el que esta interfaz pueda pasar a formar parte de dicho estándar.

En líneas generales, el diseño de POSIX.5 está basado en identificar las abstracciones de POSIX.1 (tipos, valores y operaciones) que están representadas en el lenguaje C y expresar dichas abstracciones de la mejor forma posible en Ada. Esta idea general se plasma en una serie de directrices, cada una de las cuales analiza un tópico concreto de la interfaz POSIX.1 y determina como representar dicho tópico en Ada.

A continuación se detallan las principales directrices utilizadas en el diseño de la interfaz POSIX/Ada para los servicios de trazas POSIX.

Tipos de datos

1003.1q define nuevos tipos de datos, tales como identificadores de trazas, atributos de trazas, etc. También hace uso de tipos de datos definidos en otras secciones de 1003.1. Para diseñar la interfaz Ada a los servicios de trazas es necesario decidir la correspondencia de estos tipos entre los lenguajes C y Ada.

En 1003.1q se observa que los nuevos tipos introducidos pertenecen a una de estas tres categorías: tipos opacos, tipos estructura y tipos enteros que representan valores enumerados.

Por tipo opaco se entiende aquel que es accesible sólo mediante funciones definidas en 1003.1q. La correspondencia en Ada es, evidentemente, definir estos tipos como privados. De esta forma, un tipo opaco como el identificador de traza:

```
typedef implementation-defined trace_id_t;
```

se convierte en un tipo privado

```
type Trace_Identifier is private;
```

La segunda categoría de tipos de datos a considerar son estructuras, las cuales se utilizan tan sólo para consultar su información. Por tanto, no es necesario considerar que sus componentes puedan ser modificados por la aplicación. Para estos casos, la mejor

correspondencia en Ada es la de un tipo privado acompañado de funciones que retornen cada uno de los componentes de la estructura. Por ejemplo, la estructura que representa la información asociada a un evento:

```
struct posix_trace_status_info {
  int posix_stream_status;
  int posix_stream_full_status;
  int posix_stream_flush_status;
  ...
}
```

se convierte en el siguiente tipo privado junto con las funciones de consulta necesarias:

```
type Status is private;

function Is_Full          (Trace_Status : Status) return Boolean;
function Is_Flushing      (Trace_Status : Status) return Boolean;
...
```

Una representación alternativa consistiría en utilizar tipos estructura Ada. El inconveniente de esta solución estriba en que el programador podría hacer uso de notación agregada para expresar objetos de estas estructuras, lo que podría invalidar el programa en el caso de que el fabricante del sistema, o una posterior revisión del estándar, incorporase extensiones a dicha estructura.

Finalmente, en muchas ocasiones el tipo entero se utiliza para representar valores enumerados: opciones, valores booleanos, etc. En el diseño de POSIX.5 se decidió mantener la decisión de utilizar tipos enteros restringiendo el conjunto de posibles valores. Como excepción, cuando el tipo entero representa claramente un valor booleano, en Ada se utiliza el tipo `Boolean`. Veamos el siguiente ejemplo que ilustra esta conversión de tipos.

```
#define POSIX_TRACE_LOOP             implementation-defined
#define POSIX_TRACE_UNTIL_FULL       implementation-defined
#define POSIX_TRACE_FLUSH            implementation-defined
```

Estas constantes representan las tres posibles políticas de traza llena definidas en 1003.1q. Estas constantes se utilizan como posibles valores para argumentos de tipo entero. En Ada se definen las mismas constantes para un nuevo tipo entero cuyo rango debe permitir tan sólo estos valores.

```
type Stream_Full_Policy is range implementation-defined;

Loop_Stream       : constant Stream_Full_Policy
```

```
                    := implementation-defined;
Until_Full_Stream : constant Stream_Full_Policy
                    := implementation-defined;
Flush_Stream      : constant Stream_Full_Policy
                    := implementation-defined;
```

La alternativa de utilizar tipos enumerados del lenguaje Ada fue considerada pero desestimada. El principal problema del tipo enumerado estriba en la posible dificultad de establecer una asociación entre cada valor del tipo enumerado y su valor correspondiente utilizado por el sistema operativo, por ejemplo, en el caso de que el sistema operativo utilice números enteros no contiguos para representar una serie de valores de una determinada opción.

El resto de tipos utilizados por 1003.1q están definidos en el estándar POSIX.1. Estos tipos tienen ya su equivalencia en POSIX.5, por lo que han sido directamente utilizados en el diseño de la interfaz Ada para 1003.1q.

Espacio de nombres

El espacio de nombres utilizado en POSIX.1 es muy grande. Una de las metas del diseño de POSIX.5 fue normalizar este espacio de nombres, definiendo un conjunto de reglas consistentes para el nombrado de tipos, constantes y operaciones. Es cierto que las interfaces POSIX más antiguas carecen de reglas consistentes de nombrado, pero las interfaces más recientes, tales como 1003.1q, han seguido un diseño más cuidadoso en este aspecto. No obstante, el estilo de nombrado que se sigue en estos estándares más recientes difiere notablemente del utilizado en Ada, especialmente porque Ada ofrece mejores características que C para el nombrado de entidades, tales como paquetes, sobrecarga de nombres, etc. Por ello, en el diseño de la interfaz Ada para 1003.1q se ha decidido aplicar, en la medida de lo posible, las siguientes reglas de transformación de nombres.

- El prefijo `posix_trace_` está presente en todas las operaciones de 1003.1q. También lo está en algunos de los nuevos tipos introducidos por este estándar y en todas las constantes. Este prefijo ha sido eliminado en Ada, al no ser necesario dado que dicho prefijo lo ofrece el paquete `POSIX.Trace`, en el cual se encuentran las definiciones de la interfaz Ada para 1003.1q.

- Cuando en 1003.1q una operación se aplica a un objeto de un cierto tipo, el tipo de dicho objeto se incluye en el prefijo de dicha operación, tal como puede observarse en la función `posix_trace_attr_get_attributes`, la cual se aplica a objetos del tipo atributos de traza (`attr`). En el caso de Ada, este componente del prefijo también es eliminado, ya que no es práctica habitual este estilo de nombrado.

- Una vez eliminados los prefijos, la longitud de los nombres se reduce notablemente, lo que permite mejorar la expresividad de cada nombre. De esta forma, en Ada las palabras compuestas se separan utilizando el carácter subrayado y las abreviaturas se sustituyen por sus palabras completas respectivas.
- En algunos casos, la aplicación de las reglas anteriores conduce a subprogramas Ada de igual nombre. Dado que los perfiles de estos subprogramas difieren, puede utilizarse la sobrecarga de nombres del lenguaje Ada.

A continuación se muestran algunos ejemplos de aplicación de estas reglas:

```
// Nombres de la Interfaz C

POSIX_TRACE_CLOSE_FOR_CHILD
posix_trace_attr_getmaxsystemeventsize
posix_trace_trid_eventid_open
posix_trace_eventid_open

-- Equivalencia en la Interfaz Ada

Close_For_Child
Get_Maximum_System_Event_Size
Open
Open
```

Funciones C multi-propósito

En 1003.1q, al igual que en otras partes de 1003.1, se han definido funciones C multi-propósito. En estos casos un argumento determina las acciones concretas a realizar por la función. Por ejemplo, la función que establece el filtro para una traza actualiza dicho filtro en función de un argumento (`how`), el cual determina si la actualización consiste en asignar, incluir o excluir eventos del filtro.

```
// Valores para el argumento how
#define  POSIX_TRACE_SET_EVENTSET    implementation-defined
#define  POSIX_TRACE_ADD_EVENTSET    implementation-defined
#define  POSIX_TRACE_SUB_EVENTSET    implementation-defined

int posix_trace_set_filter (trace_id_t trid,
                            const trace_event_set_t *set,
                            int how);
```

En POSIX.5 se prefiere descomponer estas funciones en varias, encargándose cada

una de ellas de una sola acción bien definida. En estos casos el nombre de la función se modifica para representar con claridad la acción a realizar.

La función anterior sería transformada en Ada en las tres funciones siguientes:

```
procedure Set_Filter
procedure Add_To_Filter
procedure Subtract_From_Filter
```

Tratamiento de errores

En POSIX.1 ciertas funciones devuelven un valor `-1` cuando se detecta alguna situación errónea. En estos casos, la variable `errno` es asignada por el sistema con el motivo concreto de dicho error. En otros casos, es la función la que retorna el código de error. Esto hace que sea característico en un programa verificar este código de error tras cada llamada a una función POSIX.

En Ada existe un mecanismo mucho más adecuado para tratar situaciones de error, las excepciones. En POSIX.5, los subprogramas no devuelven códigos de error, sino que se eleva la excepción `POSIX.Error` cuando se detecta una situación errónea. El motivo concreto del error puede consultarse utilizando la función `POSIX.Get_Error_Code`. De esta forma, en Ada se elimina la necesidad de incluir código de detección de errores tras cada llamada POSIX y asegura que cualquier error que se produzca será notificado a la aplicación, ya que no se requiere programar de forma explícita el tratamiento de errores.

Esta decisión tiene una consecuencia muy importante el en diseño de la interfaz Ada, ya que el subprograma Ada correspondiente a una función C no va a devolver un código de error. Debido a esto, es necesario decidir si una función C se convierte en una función o subprograma Ada, y en el caso de ser una función, qué valor devolverá.

En el diseño de la interfaz Ada para 1003.1q se han seguido las siguientes reglas para tomar esta decisión:

- Si la función C devuelve un único resultado en uno de sus argumentos, el subprograma Ada correspondiente es una función que devuelve dicho resultado.
- En cualquier otro caso, es decir, si la función C devuelve varios resultados en sus argumentos, o si no devuelve ninguno, el subprograma Ada correspondiente es un procedimiento.

A continuación se muestran dos ejemplos de conversión de funciones C en sus correspondientes subprogramas Ada.

```
int posix_trace_start (trace_id_t trid);

int posix_trace_trid_eventid_open (trace_id_t trid,
                                   const char *event_name,
                                   trace_event_id_t *event);
procedure Start
  (Trid           : in Trace_Identifier;
   Masked_Signals : in POSIX.Signal_Masking := POSIX.RTS_Signals);

function Open     (Trid : Trace_Identifier;
                   Name : POSIX_String) return Event_Identifier;
```

Iteradores

Un iterador es un operador que permite visitar todas las partes de un objeto estructurado. En el diseño de POSIX.5 se debatió ampliamente cual sería la forma más adecuada de los iteradores, y se tuvieron en cuenta dos alternativas. En la primera alternativa se propuso que un iterador estaría constituido por un conjunto de operaciones, las cuales permitirían iniciar la iteración, obtener el siguiente elemento sobre el cual iterar, consultar el fin de la iteración, reiniciarla y detenerla. Como segunda alternativa se propuso la utilización de procedimientos genéricos, siendo ésta la alternativa elegida, ya que con ella se consigue un código más compacto y elegante, además de ser más segura respecto a su uso concurrente.

En 1003.1q existe tan sólo un caso en el que se define un iterador, el cual permite visitar los tipos de eventos que están definidos en una traza.

La interfaz C consta de dos funciones, una que permite recuperar el siguiente tipo de evento definido y otra que permite reiniciar la iteración.

```
int posix_trace_eventtypelist_getnext_id (trace_id_t trid,
                                          trace_event_id_t *event,
                                          int *unavailable);

int posix_trace_eventtypelist_rewind (trace_id_t trid);
```

En la interfaz Ada el programador debe instanciar `For_Every_Event_Type` con un procedimiento que será ejecutado para cada uno de los tipos de eventos definidos en la traza.

```
generic
with procedure Action (Event : in Event_Identifier;
                       Quit  : in out Boolean);
```

```
procedure For_Every_Event_Type (Trid : in Trace_Identifier);
```

Operaciones interrumpibles

Algunas funciones de 1003.1 pueden finalizar debido a la entrega de una señal. En estos casos, la función termina con un código de error igual al valor `EINTR`.En el caso de la interfaz Ada, tal como se ha expuesto anteriormente, se generaría la excepción `POSIX_ERROR`. En el diseño de POSIX.5 se consideró que sería conveniente, para facilitar la programación, incorporar un argumento a cada operación interrumpible que estableciese la máscara de señales para el proceso mientras durase la ejecución de dicha operación. Este argumento, por defecto, indica que se enmascaran todas las señales utilizadas por el soporte en tiempo de ejecución. Los otros posibles valores son enmascarar todas las señales o ninguna.

Consideraciones adicionales

La mayor parte del diseño de la interfaz Ada para los servicios de trazas POSIX puede basarse en las directrices descritas con anterioridad. No obstante, hay un aspecto de 1003.1q que requiere una consideración aparte, el tipo de datos que representa la información que puede asociarse a un evento.

Cuando se genera un evento, bien por parte del sistema operativo o bien desde la aplicación, y se guarda en una traza, 1003.1q permite asociar información a dicho evento, generalmente dependiente del tipo del evento. Esta información queda definida por el sistema o por la aplicación, no por 1003.1q, por lo que las funciones que permiten generar y recuperar eventos disponen de un argumento de tipo puntero a memoria, para permitir representar con total libertad la estructura o tipo de dicha información. Por ejemplo, las dos siguientes funciones [1] hacen uso de esta característica, en los argumentos `data_ptr` y `data`.

```
void posix_trace_event (trace_event_id_t event_id,
                        const void *data_ptr,
                        size_t data_len);

int posix_trace_getnext_event (trace_id_t trid,
                               struct posix_trace_event_info *event,
                               void *data,
                               size_t  num_bytes,
                               size_t *data_len,
                               int *unavailable);
```

[1] estas funciones se utilizan respectivamente para generar un evento y recuperar un evento de una traza

En Ada el uso de este tipo de datos (puntero a memoria) está claramente desaconsejado, por lo que se intentó utilizar alguna solución más segura desde el punto de vista de la programación.

Una alternativa adecuada es la de ofrecer procedimientos genéricos que puedan ser instanciados con el tipo de datos que representa la información que se asocia al evento que se genera. De esta forma, los procedimientos Ada correspondientes a las funciones C anteriores serían los siguientes:

```
generic
  type T is private;
procedure Trace_Event (Event : in Event_Identifier;
                       Data  : in T);

generic
  type T is private;
procedure Get_Next_Event
  (Trid           : in  Trace_Identifier;
   Info           : out Event_Info;
   Data           : out T;
   Unavailable    : out Boolean;
   Masked_Signals : in  POSIX.Signal_Masking := POSIX.RTS_Signals);
```

Esta solución, aunque en principio parece satisfactoria, presenta un serio inconveniente, ya que no puede determinarse el tipo de la información que el evento tiene asociada (`Data`) hasta que se conoce el tipo del evento que se recupera (`Info`). Con esta solución, se obliga al programador a determinar el tipo de la información antes de conocer el tipo del evento, lo cual es inviable.

Debido a esto, no queda otra alternativa que recurrir a una solución poco satisfactoria que consiste en diseñar una interfaz de bajo nivel similar a la de C. Para ello, se ha seguido la misma solución que se aplica en otras partes de POSIX.5, en la cual el tipo puntero a memoria se asimila al tipo predefinido `Ada.Streams.Stream_Element_Array` (que corresponde, en esencia, a un vector de bytes), requiriendo que el programador realice conversiones libres entre el tipo de datos de su información y este tipo de datos. Los procedimientos anteriores quedarían como sigue:

```
procedure Trace_Event (Event : in Event_Identifier;
                       Data  : in Ada.Streams.Stream_Element_Array);
procedure Get_Next_Event
  (Trid           : in  Trace_Identifier;
   Info           : out Event_Info;
   Data           : out Ada.Streams.Stream_Element_Array;
```

```
   Last           : out Ada.Streams.Stream_Element_Offset;
   Unavailable    : out Boolean;
   Masked_Signals : in  POSIX.Signal_Masking := POSIX.RTS_Signals);
```

La versión genérica de `Trace_Event` puede mantenerse junto con su versión no genérica, pero no en el caso de `Get_Next_Event`.

Una forma alternativa de abordar este problema consistiría en utilizar las características de programación orientada a objetos del lenguaje Ada, y en concreto, los tipos extensibles, para representar la información asociada a un evento. Desafortunadamente, esta solución no sería compatible con la interfaz C, lo que imposibilitaría disponer de un sistema que pudiese ofrecer las interfaces Ada y C de los servicios de trazas, ni poder construir la interfaz Ada sobre la interfaz C, lo que es una situación bastante común.

4.2.2. Definición

En esta sección se muestra la especificación completa del paquete `POSIX.Trace`, el cual constituye la interfaz POSIX Ada para los servicios de trazas POSIX definidos en el estándar 1003.1q. Este paquete está dividido en secciones, las cuales corresponden con las secciones del estándar 1003.1q. En cada sección se presenta la interfaz Ada propuesta junto con la interfaz original en C, seguidas de una breve descripción de los servicios que ofrece dicha sección y los aspectos más destacables de la transformación de la interfaz C en Ada.

Dependencias del paquete POSIX.Trace

```
with System;
with Ada.Streams;
with Ada.Task_Identification;
with POSIX.Process_Identification;
with POSIX.IO;

package POSIX.Trace is
```

Tipos de datos primitivos del sistema. *(Sección 1003.1q 2.5)*

Interfaz C

```
typedef implementation-defined trace_id_t;
```

```
typedef implementation-defined trace_attr_t;
typedef implementation-defined trace_event_id_t;
typedef implementation-defined trace_event_set_t;
```

Interfaz Ada

```
type Trace_Identifier is private;
type Attributes       is private;
type Event_Identifier is private;
type Event_Set        is private;
```

Estos cuatro tipos privados corresponden con los cuatro tipos opacos definidos en 1003.1q. Representan respectivamente los identificadores de trazas, los atributos que definen las características de una traza, los identificadores de tipos de eventos y los conjuntos de eventos. Como puede observarse, la transformación entre interfaces es directa.

Estructura posix_trace_status_info ***(Sección 1003.1q 24.2.1.1)***

Interfaz C

```
#define POSIX_TRACE_RUNNING      implementation-defined
#define POSIX_TRACE_SUSPENDED    implementation-defined

#define POSIX_TRACE_FULL         implementation-defined
#define POSIX_TRACE_NOT_FULL     implementation-defined

#define POSIX_TRACE_OVERRUN      implementation-defined
#define POSIX_TRACE_NO_OVERRUN   implementation-defined

#define POSIX_TRACE_FLUSHING     implementation-defined
#define POSIX_TRACE_NOT_FLUSHING implementation-defined


struct posix_trace_status_info {
   int posix_stream_status;         // POSIX_TRACE_RUNNING
                                    // or POSIX_TRACE_SUSPENDED
   int posix_stream_full_status;    // POSIX_TRACE_FULL
                                    // or POSIX_TRACE_NOT_FULL
   int posix_stream_overrun_status; // POSIX_TRACE_OVERRUN
                                    // or POSIX_TRACE_NO_OVERRUN
   int posix_stream_flush_status;   // POSIX_TRACE_FLUSHING
                                    // or POSIX_TRACE_NOT_FLUSHING
   int posix_stram_flush_error;
   int posix_log_overrun_status;    // POSIX_TRACE_OVERRUN
                                    // or POSIX_TRACE_NO_OVERRUN
   int posix_log_full_status;       // POSIX_TRACE_FULL
```

```
                                    // or POSIX_TRACE_NOT_FULL
}
```

Interfaz Ada

```
type Status is private;

type Operating_Mode is range implementation-defined;
Running   : constant Operating_Mode := implementation-defined;
Suspended : constant Operating_Mode := implementation-defined;

function Operating_Mode_Of (Trace_Status : Status) return Operating_Mode;
function Is_Full           (Trace_Status : Status) return Boolean;
function Is_Overrun        (Trace_Status : Status) return Boolean;
function Is_Flushing       (Trace_Status : Status) return Boolean;
function Flush_Error       (Trace_Status : Status) return Integer;
function Is_Log_Overrun    (Trace_Status : Status) return Boolean;
function Is_Log_Full       (Trace_Status : Status) return Boolean;
```

El tipo `Status` (`posix_trace_status_info`) representa el estado de una traza activa y de su registro. Este estado puede ser consultado utilizando los servicios definidos en la sección del estándar 24.3.15. El estado de una traza está compuesto respectivamente por su modo actual de operación (en funcionamiento o detenida), si se encuentra llena, si se ha producido desbordamiento, si se está realizando una operación de volcado en el registro, si se ha producido algún error durante el volcado, si se ha producido desbordamiento en el registro y si el registro está lleno.

El tipo estructura C `posix_trace_status_info` se ha transformado en Ada en el tipo privado `Status`. Para acceder a los componentes de dicha estructura, se definen en Ada funciones que consultan dichos componentes.

La mayor parte de los componentes de la estructura C `posix_trace_status_info` almacenan valores que son en esencia booleanos, aunque su representación en el lenguaje C, debido a la ausencia del tipo de datos booleano, es un tipo entero. Dada la naturaleza de esta información, en Ada se ha optado por utilizar valores booleanos para representar esta información. Como excepción está el modo de operación actual de la traza, el cual, aunque sólo tiene dos valores posibles en el estándar, cabe la posibilidad de que esta información pueda ser extendida en un futuro. Por ello, su representación es la de un subtipo entero restringido a los posibles valores definidos actualmente.

Estructura posix_trace_event_info structure *(Sección 1003.1q 24.2.1.2)*

Interfaz C

```
#define POSIX_TRACE_NOT_TRUNCATED    implementation-defined
#define POSIX_TRACE_TRUNCATED_RECORD implementation-defined
#define POSIX_TRACE_TRUNCATED_READ   implementation-defined

struct posix_trace_event_info {
   pid_t             posix_pid;
   trace_event_id_t  posix_event_id;
   void *            posix_prog_address;
   int               posix_truncation_status;
   //                  = POSIX_TRACE_NOT_TRUNCATED or
   //                    POSIX_TRACE_TRUNCATED_RECORD or
   //                    POSIX_TRACE_TRUNCATED_READ
   struct timespec   posix_timestamp;
   pthread_t         posix_thread_id;
};
```

Interfaz Ada

```
type Event_Info is private;

type Truncation_Status is range implementation-defined;

Not_Truncated    : constant Truncation_Status := implementation-defined;
Truncated_Record : constant Truncation_Status := implementation-defined;
Truncated_Read   : constant Truncation_Status := implementation-defined;

function Process_Identifier_Of (Info : in Event_Info)
            return POSIX.Process_Identification.Process_ID;
function Event_Of             (Info : in Event_Info)
            return Event_Identifier;
function Program_Address_Of    (Info : in Event_Info)
            return System.Address;
function Truncation_Status_Of  (Info : in Event_Info)
            return Truncation_Status;
function Time_Stamp_Of         (Info : in Event_Info)
            return Timespec;
function Task_Identifier_Of    (Info : in Event_Info)
            return Ada.Task_Identification.Task_Id;
```

Cuando se recupera un evento almacenado en una traza (utilizando los servicios de la sección del estándar 24.3.16), el sistema devuelve un objeto del tipo `Event_Info` (`posix_trace_event_info`). Este objeto contiene la información que 1003.1q requiere que se almacene para cada evento, la cual consiste en el proceso que lo originó, su tipo, la dirección del programa donde se originó, si sus datos asociados han sido truncados y cual es el motivo del truncado, el instante de tiempo en el que se originó y la tarea (hilo)

que lo originó.

Al igual que en el caso anterior, el tipo estructura `posix_trace_event_info` se ha transformado en Ada en el tipo privado `Event_Info` y se han definido funciones Ada para acceder a los miembros de dicha estructura.

Definiciones de tipos de eventos *(Sección 1003.1q 24.2.3)*

Interfaz C

```
#define POSIX_TRACE_START              implementation-defined
#define POSIX_TRACE_STOP               implementation-defined
#define POSIX_TRACE_FILTER             implementation-defined
#define POSIX_TRACE_OVERFLOW           implementation-defined
#define POSIX_TRACE_RESUME             implementation-defined
#define POSIX_TRACE_FLUSH_START        implementation-defined
#define POSIX_TRACE_FLUSH_STOP         implementation-defined
#define POSIX_TRACE_ERROR              implementation-defined
#define POSIX_TRACE_UNNAMED_USEREVENT  implementation-defined
```

Interfaz Ada

```
Event_Start             : constant Event_Identifier;
Event_Stop              : constant Event_Identifier;
Event_Filter            : constant Event_Identifier;
Event_Overflow          : constant Event_Identifier;
Event_Resume            : constant Event_Identifier;
Event_Flush_Start       : constant Event_Identifier;
Event_Flush_Stop        : constant Event_Identifier;
Event_Error             : constant Event_Identifier;
Event_Unnamed_Userevent : constant Event_Identifier;

function Filter_Of_Event_Start_Data
 (Data : Ada.Streams.Stream_Element_Array) return Event_Set;

function Old_Filter_Of_Event_Filter_Data
 (Data : Ada.Streams.Stream_Element_Array) return Event_Set;
function New_Filter_Of_Event_Filter_Data
 (Data : Ada.Streams.Stream_Element_Array) return Event_Set;

function Error_Of_Event_Error_Data
 (Data : Ada.Streams.Stream_Element_Array) return Integer;
```

1003.1q define ciertos tipos de eventos, los cuales corresponden a acciones realizadas por el propio subsistema de trazas, tales como iniciar o detener la acción de trazado. Estos

eventos son representados por los identificadores de eventos definidos en esta sección.

Algunos de estos tipos de eventos tienen información asociada. Para estos casos se definen varias funciones que permiten convertir la información no estructurada del evento, en el tipo de datos de la información que se asocia al tipo de evento.

Inicialización de atributos de trazas *(Sección 1003.1q 24.3.2)*

Interfaz C

```
int posix_trace_attr_init (trace_attr_t *attr);
int posix_trace_attr_destroy (trace_attr_t *attr);
```

Interfaz Ada

```
procedure Initialize (Attr : in out Attributes);
procedure Finalize   (Attr : in out Attributes);
```

Estas operaciones inicializan e invalidan un objeto atributos de traza. Estos objetos definen las características de la traza en el momento de su creación. Los servicios de las secciones 24.3.3, 24.3.4 y 24.3.5 del estándar permiten asignar y consultar los valores de estos objetos.

Funciones para recuperar y establecer información sobre una traza *(Sección 1003.1q 24.3.3)*

Interfaz C

```
#include <trace.h>

int posix_trace_getgenversion (const trace_attr_t *attr,
                               char *genversion);
int posix_trace_attr_getname  (const trace_attr_t *attr,
                               char *tracename);
int posix_trace_attr_setname  (trace_attr_t *attr,
                               const char *tracename);

#include <time.h>
#include <trace.h>

int posix_trace_attr_getcreatetime (const trace_attr_t *attr,
                                    struct timespec *createtime);
int posix_trace_attr_getclockres   (const trace_attr_t *attr,
                                    struct timespec * resolution);
```

Interfaz Ada

```
function  Get_Generation_Version (Attr : in Attributes)
   return POSIX_String;

function  Get_Name (Attr : in Attributes)
   return POSIX_String;

procedure Set_Name (Attr       : in out Attributes;
                    Trace_Name : in POSIX_String);

function  Get_Create_Time (Attr : in Attributes)
   return Timespec;

function  Get_Clock_Resolution (Attr : in Attributes)
   return Timespec;
```

Estos subprogramas permiten consultar información relativa a una traza: su versión, nombre, instante de creación y la resolución del reloj utilizada por el sistema. Con la excepción del nombre, el cual puede ser asignado por el programador, el resto de estos valores son asignados por el subsistema de trazas cuando la traza es creada.

Funciones para recuperar y establecer el comportamiento de una traza *(Sección 1003.1q 24.3.4)*

Interfaz C

```
#define POSIX_TRACE_LOOP             implementation-defined
#define POSIX_TRACE_UNTIL_FULL       implementation-defined
#define POSIX_TRACE_FLUSH            implementation-defined
#define POSIX_TRACE_APPEND           implementation-defined
#define POSIX_TRACE_INHERITED        implementation-defined
#define POSIX_TRACE_CLOSE_FOR_CHILD  implementation-defined


int posix_trace_attr_getinherited (const trace_attr_t *attr,
                                   int *inheritancepolicy);

int posix_trace_attr_setinherited (const trace_attr_t *attr,
                                   int inheritancepolicy);

// inheritancepolicy == POSIX_TRACE_INHERITED      or
//                      POSIX_TRACE_CLOSE_FOR_CHILD

int posix_trace_attr_getstreamfullpolicy (const trace_attr_t *attr,
```

```
                                        int *streampolicy);

int posix_trace_attr_setstreamfullpolicy (trace_attr_t *attr,
                                        int streampolicy);

// streampolicy == POSIX_TRACE_LOOP       or
//                 POSIX_TRACE_UNTIL_FULL or
//                 POSIX_TRACE_FLUSH

int posix_trace_attr_getlogfullpolicy (const trace_attr_t *attr,
                                       int *logpolicy);

int posix_trace_attr_setlogfullpolicy (trace_attr_t *attr,
                                       int logpolicy);

// logpolicy == POSIX_TRACE_LOOP       or
//              POSIX_TRACE_UNTIL_FULL or
//              POSIX_TRACE_APPEND
```

Interfaz Ada

```
type Inheritance_Policy is range implementation-defined;

Close_For_Child : constant Inheritance_Policy := implementation-defined;
Inherited       : constant Inheritance_Policy := implementation-defined;

function Get_Inheritance_Policy (Attr : in Attributes)
    return Inheritance_Policy;

procedure Set_Inheritance_Policy
    (Attr                  : in out Attributes;
     New_Inheritance_Policy : in Inheritance_Policy);

type Stream_Full_Policy is range implementation-defined;

Loop_Stream       : constant Stream_Full_Policy := implementation-defined;
Until_Full_Stream : constant Stream_Full_Policy := implementation-defined;
Flush_Stream      : constant Stream_Full_Policy := implementation-defined;

function Get_Stream_Full_Policy (Attr : in Attributes)
    return Stream_Full_Policy;

procedure Set_Stream_Full_Policy
    (Attr                  : in out Attributes;
     New_Stream_Full_Policy : in Stream_Full_Policy);


type Log_Full_Policy is range implementation-defined;
```

```
Loop_Log       : constant Log_Full_Policy := implementation-defined;
Until_Full_Log : constant Log_Full_Policy := implementation-defined;
Append_Log     : constant Log_Full_Policy := implementation-defined;

function Get_Log_Full_Policy (Attr : in Attributes) return Log_Full_Policy;

procedure Set_Log_Full_Policy
    (Attr               : in out Attributes;
     New_Log_Full_Policy : in Log_Full_Policy);
```

Los servicios de esta sección permiten gestionar las políticas aplicables a una traza. Estas políticas controlan tres aspectos relativos a una traza: la herencia de la traza por parte de procesos hijos, el comportamiento de la traza cuando su espacio de almacenamiento se agota y el comportamiento del registro cuando su espacio de almacenamiento se agota.

Funciones para establecer y recuperar atributos referentes a tamaños de la traza *(Sección 1003.1q 24.3.5)*

Interfaz C

```
int posix_trace_attr_getmaxusereventsize   (const trace_attr_t *attr,
                                            size_t data_len,
                                            size_t *eventsize);

int posix_trace_attr_getmaxsystemeventsize (const trace_attr_t *attr,
                                             size_t *eventsize);

int posix_trace_attr_getmaxdatasize        (const trace_attr_t *attr,
                                            size_t *maxdatasize);

int posix_trace_attr_setmaxdatasize        (trace_attr_t *attr,
                                            size_t maxdatasize);

int posix_trace_attr_getstreamsize         (const trace_attr_t *attr,
                                            size_t *streamsize);

int posix_trace_attr_setstreamsize         (trace_attr_t *attr,
                                            size_t streamsize);

int posix_trace_attr_getlogsize            (const trace_attr_t *attr,
                                            size_t *logsize);

int posix_trace_attr_setlogsize            (trace_attr_t *attr,
                                            size_t logsize);
```

Interfaz Ada

```
function Get_Maximum_User_Event_Size
    (Attr        : Attributes;
     Data_Length : Natural) return Natural;

function Get_Maximum_System_Event_Size
    (Attr : Attributes) return Natural;

function Get_Maximum_Data_Size
    (Attr : Attributes) return Natural;

procedure Set_Maximum_Data_Size
    (Attr              : in out Attributes;
     New_Max_Data_Size : in Natural);

function Get_Stream_Size
    (Attr : Attributes) return Natural;

procedure Set_Stream_Size
    (Attr            : in out Attributes;
     New_Stream_Size : in Natural);

function Get_Log_Size
    (Attr : Attributes) return Natural;

procedure Set_Log_Size
    (Attr         : in out Attributes;
     New_Log_Size : in     Natural);
```

Los servicios de esta sección permiten al programador determinar el espacio de almacenamiento que desea asignar a una traza y a su registro. Para ello, las dos primeras funciones notifican el espacio que requiere el subsistema de trazas para almacenar un evento de usuario o un evento del sistema. A partir de esta información, el programador puede establecer el espacio que se asigna a la información asociada a un evento, a la traza y al registro. Cabe destacar que el estándar no contempla reservar espacio para un número determinado de eventos, sino que requiere del programador que establezca el espacio en bytes para almacenar dicha información.

Creación, volcado y finalización de trazas *(Sección 1003.1q 24.3.6)*

Interfaz C

```
int posix_trace_create (pid_t pid,
                        const trace_attr_t *attr,
```

```
                        trace_id_t *trid);

int posix_trace_create_withlog (pid_t pid,
                                const trace_attr_t *attr,
                                int file_desc,
                                trace_id_t *trid);

int posix_trace_flush (trace_id_t trid);

int posix_trace_shutdown (trace_id_t trid);
```

Interfaz Ada

```
function Create
    (For_Process    : in  POSIX.Process_Identification.Process_ID;
     Attr           : in  Attributes;
     Masked_Signals : in  POSIX.Signal_Masking := POSIX.RTS_Signals)
  return Trace_Identifier;

function Create_With_Log
    (For_Process    : in POSIX.Process_Identification.Process_ID;
     Attr           : in Attributes;
     File           : in POSIX.IO.File_Descriptor;
     Masked_Signals : in POSIX.Signal_Masking := POSIX.RTS_Signals);
  return Trace_Identifier;

procedure Flush (Trid : in Trace_Identifier);

procedure Shutdown (Trid : in Trace_Identifier);
```

Con las dos primeras funciones, un proceso controlador puede crear una traza activa con o sin registro para otro proceso (o para el mismo). La traza se crea inactiva, y comenzará a registrar los eventos originados por el proceso trazado una vez sea activada con los servicios de la sección 24.3.12 del estándar. El identificador de traza que devuelven estas funciones se utiliza posteriormente para controlar la traza y recuperar eventos de ella. La operación `Flush` almacena los eventos de la traza en su registro, liberando así el espacio de almacenamiento de la traza. La operación `Shutdown` permite destruir una traza previamente creada.

Borrar el contenido de la traza y su registro *(Sección 1003.1q 24.3.7)*

Interfaz C

```
int posix_trace_clear (trace_id_t trid);
```

Interfaz Ada

```
procedure Clear (Trid : Trace_Identifier);
```

El procedimiento `Clear` elimina los eventos almacenados en la traza así como en su registo si lo hubiera.

Manipulación de identificadores de tipos de eventos *(Sección 1003.1q 24.3.8)*

Interfaz C

```
int posix_trace_trid_eventid_open (trace_id_t trid,
                                   const char *event_name,
                                   trace_event_id_t *event);

int posix_trace_eventid_get_name (trace_id_t trid,
                                  trace_event_id_t event,
                                  char *event_name);

int posix_trace_eventid_equal (trace_id_t trid,
                               trace_event_id_t event1,
                               trace_event_id_t event2);
```

Interfaz Ada

```
function Open     (Trid : Trace_Identifier;
                   Name : POSIX_String) return Event_Identifier;

function Get_Name (Trid  : Trace_Identifier;
                   Event : Event_Identifier) return POSIX_String;

function Equal    (Trid        : Trace_Identifier;
                   Left, Right : Event_Identifier) return Boolean;
```

La función `Open` es utilizada para definir nuevos tipos de eventos o acceder a tipos de eventos ya definidos en la traza. El argumento `Name` indica el nombre del tipo de evento del cual se desea obtener su identificador. Si este nombre de tipo de evento no existe, se crea un nuevo tipo de evento con dicho nombre. `Get_Name` devuelve el nombre del tipo de evento identificado por el argumento `Event`. `Equal` permite la comparación entre tipos de eventos definidos en la misma traza.

Iteración sobre los tipos de eventos definidos en una traza *(Sección 1003.1q 24.3.9)*

Interfaz C

```
int posix_trace_eventtypelist_getnext_id (trace_id_t trid,
                                          trace_event_id_t *event,
                                          int *unavailable);

int posix_trace_eventtypelist_rewind (trace_id_t trid);
```

Interfaz Ada

```
generic
with procedure Action (Event : in Event_Identifier;
                       Quit  : in out Boolean);

procedure For_Every_Event_Type (Trid : in Trace_Identifier);
```

En esta sección se definen servicios para iterar sobre los tipos de eventos que se encuentran definidos en una traza. Como puede observarse, ambas interfaces difieren notablemente, dado que en POSIX/Ada se prefiere utilizar procedimientos genéricos como iteradores frente al estilo de interfaz C, en el cual una función se encarga de devolver el siguiente elemento sobre el cual se iterará cada vez que es llamada. En la interfaz Ada el programador debe instanciar `For_Every_Event_Type` con un procedimiento que será ejecutado para cada uno de los tipos de eventos definidos en la traza.

Manipular conjuntos de eventos *(Sección 1003.1q 24.3.10)*

Interfaz C

```
#define  POSIX_TRACE_WOPID_EVENTS    implementation-defined
#define  POSIX_TRACE_SYSTEM_EVENTS   implementation-defined
#define  POSIX_TRACE_ALL_EVENTS      implementation-defined

int posix_trace_eventset_empty (trace_event_set_t *set);

int posix_trace_eventset_fill (trace_event_set_t *set,
                               int what);

int posix_trace_eventset_add (trace_event_id_t event_id,
                              trace_event_set_t *set);

int posix_trace_eventset_del (trace_event_id_t event_id,
                              trace_event_set_t *set);
```

```
int posix_trace_eventset_complement(trace_event_set_t *set);

int posix_trace_eventset_ismember (trace_event_id_t event_id,
                                   const trace_event_set_t *set,
                                   int *ismember);
```

Interfaz Ada

```
procedure Add_Event             (Set   : in out Event_Set;
                                 Event : in Event_Identifier);

procedure Add_All_Process_Independent_Events (Set : in out Event_Set);

procedure Add_All_System_Events (Set : in out Event_Set);

procedure Add_All_Events        (Set : in out Event_Set);

procedure Delete_Event          (Set   : in out Event_Set;
                                 Event : in     Event_Identifier);

procedure Delete_All_Events     (Set : in out Event_Set);

procedure Complement            (Set : in out Event_Set);

function  Is_Member             (Set   : in Event_Set;
                                 Event : in Event_Identifier)
    return Boolean;
```

El tipo `Event_Set` se utiliza en 1003.1q para definir conjuntos de tipos de eventos. Estos conjuntos se utilizan, por ejemplo, para establecer el filtrado de eventos en una traza. La interfaz Ada diseñada corresponde con la interfaz que se utiliza en 1003.5c para manipular conjuntos de señales. Tal como puede observarse, `posix_trace_eventset_fill` es una función multi-propósito, ya que realiza acciones distintas en función de su argumento `what`. En la interfaz Ada esta función se ha transformado en tres, ya que éste es el estilo preferido en 1003.5c.

Establecer el filtrado en trazas *(Sección 1003.1q 24.3.11)*

Interfaz C

```
#define  POSIX_TRACE_SET_EVENTSET    implementation-defined
#define  POSIX_TRACE_ADD_EVENTSET    implementation-defined
#define  POSIX_TRACE_SUB_EVENTSET    implementation-defined

int posix_trace_set_filter (trace_id_t trid,
```

```
                          const trace_event_set_t *set,
                          int how);

// how = POSIX_TRACE_SET_EVENTSET or
//       POSIX_TRACE_ADD_EVENTSET or
//       POSIX_TRACE_SUB_EVENTSET

int posix_trace_get_filter (trace_id_t trid,
                          trace_event_set_t *set);
```

Interfaz Ada

```
procedure Set_Filter
  (Trid           : in Trace_Identifier;
   Set            : in  Event_Set;
   Masked_Signals : in POSIX.Signal_Masking := POSIX.RTS_Signals);

procedure Add_To_Filter
  (Trid           : in Trace_Identifier;
   Set            : in Event_Set;
   Masked_Signals : in POSIX.Signal_Masking := POSIX.RTS_Signals);

procedure Subtract_From_Filter
  (Trid           : in Trace_Identifier;
   Set            : in  Event_Set;
   Masked_Signals : in POSIX.Signal_Masking := POSIX.RTS_Signals);

function Get_Filter
  (Trid           : in Trace_Identifier;
   Masked_Signals : in POSIX.Signal_Masking := POSIX.RTS_Signals)
  return Event_Set;
```

Una traza activa puede filtrar ciertos tipos de eventos. Los eventos cuyos tipos están filtrados no son registrados en la traza. Los servicios de esta sección permiten al programador establecer este filtrado. Al igual que en la sección anterior, la función multi-propósito `posix_trace_set_filter` se ha transformado en varias funciones Ada.

Activación y desactivación de trazas *(Sección 1003.1q 24.3.12)*

Interfaz C

```
int posix_trace_start (trace_id_t trid);

int posix_trace_stop (trace_id_t trid);
```

Interfaz Ada

```
procedure Start
  (Trid           : in Trace_Identifier;
   Masked_Signals : in POSIX.Signal_Masking := POSIX.RTS_Signals);

procedure Stop
  (Trid           : in Trace_Identifier;
   Masked_Signals : in POSIX.Signal_Masking := POSIX.RTS_Signals);
```

Estos servicios activan o desactivan el registro de eventos en una traza.

Funciones para instrumentar el código de una aplicación *(Sección 1003.1q 24.3.13)*

Interfaz C

```
int posix_trace_eventid_open (const char *event_name,
                              trace_event_id_t *event_id);

void posix_trace_event (trace_event_id_t event_id,
                        const void *data_ptr,
                        size_t data_len);
```

Interfaz Ada

```
function Open (Name :  in POSIX_String) return Event_Identifier;

procedure Trace_Event (Event : in Event_Identifier;
                       Data  : in Ada.Streams.Stream_Element_Array);
generic
  type T is private;
procedure Trace_Event (Event : in Event_Identifier;
                       Data  : in T);
```

Estos servicios permiten instrumentar el código de una aplicación. El programador puede definir nuevos tipos de eventos con la función `Open`. Estos eventos quedan definidos para todas las trazas (actuales o futuras) del proceso que realiza esta llamada. `Trace_Event` genera un evento del tipo especificado en el argumento `Event`. Este evento es almacenado en todas las trazas del proceso en las cuales no esté filtrado. El sistema de trazas almacena para este evento la información definida en la sección del estándar 24.2.1.2 junto con los datos dependientes de la aplicación que se encuentran en el argumento `Data`. Como puede observarse, en estos subprogramas no aparece el identificador de traza como argumento. Esto es debido a que estos servicios son los utilizados por el

proceso trazado, el cual desconoce si tiene trazas asociadas.

Gestión del registro de trazas *(Sección 1003.1q 24.3.14)*

Interfaz C

```
int posix_trace_open   (int file_desc, trace_id_t *trid);

int posix_trace_rewind (trace_id_t trid);

int posix_trace_close  (trace_id_t trid);
```

Interfaz Ada

```
function Open (File: in POSIX.IO.File_Descriptor;
               Masked_Signals : in POSIX.Signal_Masking := POSIX.RTS_Signals)
   return Trace_Identifier;

procedure Rewind (Trid : in Trace_Identifier);

procedure Close  (Trid : in Trace_Identifier);
```

Con los servicios de esta sección los procesos analizadores pueden acceder a trazas pre-grabadas, creadas a partir de trazas activas con registro.

Funciones para recuperar atributos y el estado de una traza *(Sección 1003.1q 24.3.15)*

Interfaz C

```
int posix_trace_get_attr (trace_id_t trid,
                          trace_attr_t *attr);

int posix_trace_get_status (trace_id_t trid,
                            struct posix_trace_status_info *statusinfo);
```

Interfaz Ada

```
function Get_Attributes (Trid : in Trace_Identifier) return Attributes;

function Get_Status     (Trid : in Trace_Identifier) return Status;
```

Estas funciones recuperan los atributos de creación y el estado actual de la traza. Los valores de `Status` pueden ser consultados utilizando las funciones definidas en la sección

del estándar 24.2.1.1.

Funciones para recuperar Eventos *(Sección 1003.1q 24.3.16)*

Interfaz C

```
int posix_trace_getnext_event (trace_id_t trid,
                               struct posix_trace_event_info *event,
                               void *data,
                               size_t  num_bytes,
                               size_t *data_len,
                               int *unavailable);

int posix_trace_timedgetnext_event (trace_id_t trid,
                                    struct posix_trace_event_info *event,
                                    void *data,
                                    size_t  num_bytes,
                                    size_t *data_len,
                                    int *unavailable,
                                    const struct timespec *abs_timeout);

int posix_trace_trygetnext_event (trace_id_t trid,
                                  struct posix_trace_event_info *event,
                                  void *data,
                                  size_t  num_bytes,
                                  size_t *data_len,
                                  int *unavailable);
```

Interfaz Ada

```
procedure Get_Next_Event
  (Trid           : in Trace_Identifier;
   Info           : out Event_Info;
   Data           : out Ada.Streams.Stream_Element_Array;
   Last           : out Ada.Streams.Stream_Element_Offset;
   Unavailable    : out Boolean;
   Masked_Signals : in POSIX.Signal_Masking := POSIX.RTS_Signals);

procedure Timed_Get_Next_Event
  (Trid           : in Trace_Identifier;
   Info           : out Event_Info;
   Data           : out Ada.Streams.Stream_Element_Array;
   Last           : out Ada.Streams.Stream_Element_Offset;
   Unavailable    : out Boolean;
   Timeout        : in Timespec;
   Masked_Signals : in POSIX.Signal_Masking := POSIX.RTS_Signals);
```

```
procedure Try_Get_Next_Event
  (Trid        : in Trace_Identifier;
   Info        : out Event_Info;
   Data        : out Ada.Streams.Stream_Element_Array;
   Last        : out Ada.Streams.Stream_Element_Offset;
   Unavailable : out Boolean);
```

Estos procedimientos recuperan un evento almacenado en una traza. La función `Get_Next_Event` puede utilizarse en trazas activas o pre-grabadas. Las funciones `Timed_Get_Next_Event` y `Try_Get_Next_Event` son aplicables tan solo a trazas activas.

El procedimiento `Get_Next_Event` suspende al proceso que lo invoca si no hay eventos en la traza. `Timed_Get_Next_Event` temporiza esta espera mientras que `Try_Get_Next_Event` nunca detiene la ejecución del proceso.

Asociado a cada evento se retorna al programa dos argumentos, `Info` y `Data`. `Info` corresponde con la información que 1003.1q requiere que el sistema de trazas asocie a cada evento. Esta información puede recuperarse utilizando las funciones de la sección del estándar 24.1.1.1. En `Data` se retorna información no estructurada, la cual es dependiente del tipo de evento y es asignada por el sistema o la aplicación instrumentada.

```
end POSIX.Trace;
```

Modificaciones a POSIX.Limits, POSIX.Options and POSIX.Configurable_System_Limits

1003.1q define varios límites y opciones. Estos valores deben definirse en los paquetes del estándar 1003.5c que recogen este tipo de valores.

Interfaz C

```
// Conforming Implementations Options (1.3.1 and 2.9.3)

#define POSIX_TRACE                 implementation-defined
#define POSIX_TRACE_EVENT_FILTER    implementation-defined
#define POSIX_TRACE_LOG             implementation-defined
#define POSIX_TRACE_INHERIT         implementation-defined

// Minimum values (2.8.2)

#define POSIX_TRACE_SYS_MAX           implementation-defined
#define POSIX_TRACE_USER_EVENT_MAX    implementation-defined
#define POSIX_TRACE_NAME_MAX          implementation-defined
#define POSIX_TRACE_EVENT_NAME_MAX    implementation-defined
```

```
// Run-Time Invariant Values (2.8.4)

#define POSIX_TRACE_SYS_MAX            implementation-defined
#define POSIX_TRACE_USER_EVENT_MAX     implementation-defined
#define POSIX_TRACE_NAME_MAX           implementation-defined
#define POSIX_TRACE_EVENT_NAME_MAX     implementation-defined

// Configurable System Variables (4.8)

_SC_POSIX_TRACE
_SC_POSIX_TRACE_EVENT_FILTER
_SC_POSIX_TRACE_LOG
_SC_POSIX_TRACE_INHERIT
```

Interfaz Ada

```
--  Modificaciones a POSIX.Limits

subtype Trace_Sys_Maxima        is Natural range implementation-defined;
subtype Trace_User_Event_Maxima is Natural range implementation-defined;
subtype Trace_Name_Maxima       is Natural range implementation-defined;
subtype Trace_Event_Name_Maxima is Natural range implementation-defined;

Portable_Trace_Sys_Maximum        : constant Natural
                                  := implementation-defined;
Portable_Trace_User_Event_Maximum : constant Natural
                                  := implementation-defined;
Portable_Trace_Name_Maximum       : constant Natural
                                  := implementation-defined;
Portable_Trace_Event_Name_Maximum : constant Natural
                                  := implementation-defined;

--  Modificaciones a POSIX.Options

subtype Trace              is Boolean range implementation-defined;
subtype Trace_Event_Filter is Boolean range implementation-defined;
subtype Trace_Log          is Boolean range implementation-defined;
subtype Trace_Inherit      is Boolean range implementation-defined;

--  Modificaciones a POSIX.Configurable_System_Limits

function Trace_Is_Supported
   return POSIX.Options.Trace;
function Trace_Event_Filter_Is_Supported
   return POSIX.Options.Trace_Event_Filter;
function Trace_Log_Is_Supported
   return POSIX.Options.Trace_Log;
```

```
function Trace_Inherit_Is_Supported
   return POSIX.Options.Trace_Inherit;
```

4.3. Adecuación a los Sistemas de Tiempo Real

Los servicios de trazas POSIX han sido diseñados teniendo en cuenta su posible incorporación en sistemas operativos tanto de propósito general como específicos, especialmente sistemas de tiempo real.

Con el fin de facilitar la incorporación de estos servicios en sistemas de tiempo real, 1003.1q divide su funcionalidad en varias opciones, tal como se ha descrito en la sección 4.1. A continuación se discute la adecuación de cada una de estas opciones para los sistemas de tiempo real.

_POSIX_TRACE_INHERIT (Herencia de trazas). Muchos sistemas de tiempo real permiten tan sólo un proceso, dentro del cual pueden crearse varios hilos de ejecución. Para este tipo de sistemas, esta opción es inaplicable, ya que la herencia de trazas establece que las trazas de un proceso sean heredadas por sus procesos hijos, y en estos sistemas no pueden crearse procesos hijos. Considerar esta funcionalidad como opcional es una decisión imprescindible para poder incorporar los servicios de trazas en sistemas de tiempo real de estas características.

_POSIX_TRACE_EVENT_FILTER. (Filtrado de eventos). El filtrado de eventos es una funcionalidad muy útil de los servicios de trazas y su coste de implementación es relativamente pequeño. Aunque se trata de una opción de implementación, es razonable su inclusión en sistemas de tiempo real, incluso en sistemas pequeños. En este tipo de sistemas en los cuales puede haber importantes limitaciones de recursos, especialmente de memoria central, el filtrado de eventos puede ser determinante para poder reducir el tamaño de las trazas.

_POSIX_TRACE_LOG. (Registros de Trazas). La funcionalidad de registros de trazas definida en 1003.1q es muy extensa. Se trata, sin lugar a dudas, de una opción con un coste de implementación elevado, lo que puede dificultar su incorporación en sistemas de tiempo real pequeños. Por otra parte, hay sistemas de tiempo real en los que el acceso a la entrada-salida puede ser muy básico, incluso inexistente, lo que también dificulta la incorporación de esta funcionalidad. Dentro de esta opción, se permite que se realice una implementación parcial en función del soporte para el sistema de archivos existente. Por ejemplo, ciertas características de esta opción requieren la existencia de ficheros permanentes en disco, ficheros que pueden no

estar disponibles en ciertos sistemas, mientras que otras características requieren tan sólo acceso secuencial a dispositivos, algo bastante más común en los sistemas de tiempo real.

_POSIX_TRACE. (Funcionalidad mínima) Al dejar como opciones el registro de trazas y el filtrado de eventos, es bastante razonable pensar que en un breve espacio de tiempo los sistemas de tiempo real incorporarán los servicios de trazas POSIX, al menos la parte que corresponde a la funcionalidad mínima. El mayor inconveniente puede encontrarse en los requerimientos de memoria central de estos servicios. La funcionalidad mínima es bastante extensa, lo que requiere una cantidad significativa de código. Las trazas, así como otras estructuras de datos tales como la representación del espacio de nombres de eventos, pueden requerir cantidades de memoria importantes. No obstante, la utilidad del trazado de eventos en sistemas de tiempo real es indiscutible, por lo que el coste de implementación de estos servicios estará seguramente justificado.

Desde el punto de vista de su utilización, hay una característica de los servicios de trazas POSIX claramente inadecuada para sistemas de tiempo real estricto, el volcado automático de trazas en registros. 1003.1q define una política denominada `POSIX_TRACE_LOOP`, que se aplica cuando una traza con registro está próxima a llenarse. Cuando esto sucede, los eventos de la traza son volcados en el registro y, por tanto, se vacía la traza. Como puede observarse, esta acción de volcado sucede de forma no predecible en el sistema, por lo que no puede ser tenida en cuenta en el análisis de planificabilidad de la aplicación. Por tanto, la utilización de esta característica puede comprometer, sin duda, la garantía de tiempos de respuesta. Los programadores deberían evitar utilizar esta característica y realizar el volcado de forma manual mediante la función `posix_trace_flush`.

Otro aspecto que los servicios POSIX parecen no contemplar es el hecho de que en muchos sistemas de tiempo real se diferencia entre el sistema de desarrollo y el de ejecución. En este tipo de entornos, parece razonable que ambos sistemas utilicen roles distintos en lo relativo a la gestión de las trazas. Por ejemplo, parecería razonable que en un sistema de ejecución se ofreciese la posibilidad de generar registros de trazas mientras que las funciones de analizar dichas trazas estén presentes tan sólo en el sistema de desarrollo. En cambio, la definición actual de los servicios de traza no permite implementar por separado las funciones de generación de registro y su análisis.

Actualmente los servicios de trazas están siendo considerados para su inclusión en los perfiles de sistemas de tiempo real [56]. En particular, parece ser que estarán presentes en todos los perfiles exceptuando el perfil de sistema de tiempo real mínimo. Los servicios de trazas resultan demasiado extensos para un sistema de tiempo real mínimo, pero aun así su funcionalidad sería interesante también para este perfil. En un sistema de tiempo

real mínimo, estos servicios podrían utilizase para supervisar en tiempo real requerimientos temporales y funcionales de la aplicación, y, a raíz de esta supervisión, poder realizar acciones correctoras, tales como pasar a un modo de funcionamiento degradado. Parecería razonable, por tanto, abordar una definición reducida de los servicios de trazas que aportase un mínimo de funcionalidad y facilitase una implementación sencilla y compacta.

4.4. Implementación de los Servicios de Trazas POSIX en MaRTE OS

Para el sistema operativo MaRTE OS se ha realizado una implementación parcial de los servicios de trazas POSIX. Las características más destacables de esta implementación son las siguientes:

- La implementación está totalmente escrita en el lenguaje Ada, al igual que la mayor parte de MaRTE OS.
- Al igual que sucede en MaRTE OS, se han desarrollado tres interfaces a estos servicios. La primera es la propia interfaz de este sistema, la cual es similar a la interfaz POSIX/Ada propuesta en la sección 4.2.2, con la diferencia de no hacer referencia a entidades definidas en el paquete `POSIX` y sus paquetes hijos, sino a entidades definidas en el propio MaRTE OS. Sobre esta interfaz, se han construido las interfaces POSIX/C y POSIX/Ada para los servicios de trazas.
- Al tratarse MaRTE OS de una implementación del perfil de sistema de tiempo real mínimo, no existe más que un proceso, con lo que la opción de herencia de trazas no ha sido por tanto implementada.
- La opción de registro de trazas ha sido implementada de una forma bastante parcial por los siguientes motivos.

  MaRTE OS no dispone de un sistema de ficheros de propósito general, y en la versión utilizada en esta tesis (MaRTE 0.86) tampoco existen interfaces básicas de entrada-salida POSIX. Para solventar este problema se ha definido una interfaz muy básica de acceso a ficheros en memoria, lo que permite al menos crear registros de trazas en memoria. Estos registros pueden ser analizados por la propia aplicación mientras permanecen en memoria o transferidos utilizando el puerto serie a otro ordenador, donde pueden ser también analizados.

  Dada las limitaciones de esta solución, varias de las características que requiere la opción de registro de trazas no han sido implementadas, tales como el volcado

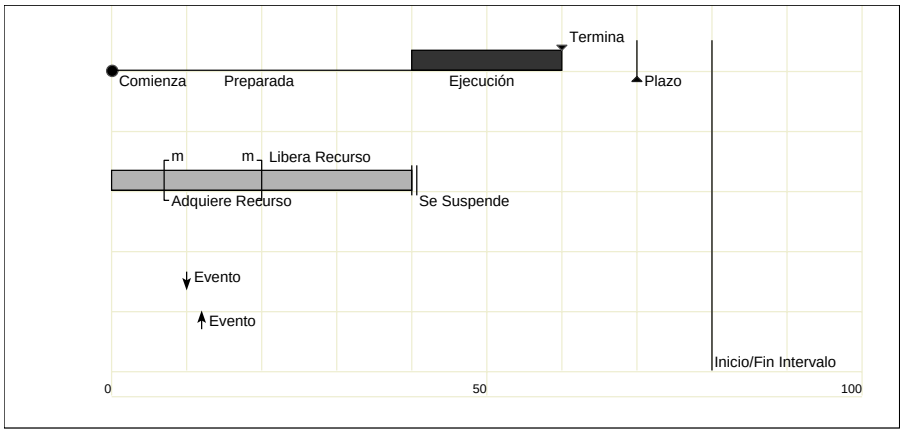


Figura 4.1: Símbolos utilizados por Quivi

manual o automático de trazas o las diferentes políticas de tratamiento del desbordamiento del registro.

En la última versión de MaRTE OS (1.2) se dispone ya de una interfaz de acceso a servicios de entrada-salida POSIX y en la actualidad se está desarrollando una nueva implementación del subsistema de trazas para MaRTE OS en la que la gestión del registro de trazas utilizará estos servicios.

- Con el fin de poder analizar trazas en un ordenador con un sistema de propósito general, se ha implementado en Linux la parte de 1003.1q que permite a un proceso analizador acceder y extraer información de trazas pre-grabadas. De esta forma, las trazas que se generan en el ordenador donde se ejecuta la aplicación con MaRTE OS son guardadas en registros en memoria, los cuales son transferidos al ordenador de desarrollo donde pueden ser posteriormente analizadas.

- También se ha implementado una aplicación de representación de cronogramas, programada en Tcl/TK, denominada Quivi [2], así como un traductor de las trazas POSIX obtenidas de MaRTE OS al formato utilizado por esta aplicación. Todos los cronogramas que aparecen en este trabajo han sido generados en MaRTE OS y representados posteriormente en Linux con Quivi. El significado de los símbolos utilizados por Quivi se muestra en la figura 4.1.

A continuación se muestra un ejemplo de utilización de los servicios para trazas POSIX en MaRTE OS.

[2] Quivi está disponible en http://www.dsic.upv.es/aespinos/quivi.html

El siguiente fragmento de programa corresponde con la tarea que asume el rol de controlador de la traza. Esta tarea crea una traza con registro, selecciona los eventos que podrán ser guardados en la traza, inicia el trazado, activa las tareas de la aplicación y, tras cinco segundos, detiene la acción de trazado y envía la traza al ordenador de desarrollo.

```
-- Creación de la traza

 Initialize (Attr);
 Set_Stream_Size (Attr, 40_000);
 Set_Stream_Full_Policy (Attr, Until_Full_Stream);

 File := Open_Or_Create ("trace.log", Read_Write, Owner_Permission_Set);
 Trid := Create_With_Log (Null_Process_ID, Attr, File);

 Event_Exec
    := Open (Trid, "Event_Exec");
 Event_Ready
    := Open (Trid, "Event_Ready");
 Event_Mutex_Lock
    := Open (Trid, "Event_Mutex_Lock");
 Event_Mutex_Unlock
    := Open (Trid, "Event_Mutex_Unlock");
 Event_System_Call_Mutex_Lock
    := Open (Trid, "Event_System_Call_Mutex_Lock");
 Event_System_Call_Mutex_Unlock
    := Open (Trid, "Event_System_Call_Mutex_Unlock");
 Event_System_Call_Cond_Timedwait
    := Open (Trid, "Event_System_Call_Cond_Timedwait");
 Event_System_Call_Return
    := Open (Trid, "Event_System_Call_Return");
 Event_Timer
    := Open (Trid, "Event_Timer");
 Event_Gnat_Delay_Until
    := Open (Trid, "Event_Gnat_Delay_Until");
 Event_Gnat_Return
    := Open (Trid, "Event_Gnat_Return");

 Add_Event (Interesting_Events, Event_Exec);
 Add_Event (Interesting_Events, Event_Ready);
 Add_Event (Interesting_Events, Event_Mutex_Lock);
 Add_Event (Interesting_Events, Event_Mutex_Unlock);
 Add_Event (Interesting_Events, Event_System_Call_Mutex_Lock);
 Add_Event (Interesting_Events, Event_System_Call_Mutex_Unlock);
 Add_Event (Interesting_Events, Event_System_Call_Cond_Timedwait);
 Add_Event (Interesting_Events, Event_System_Call_Return);
 Add_Event (Interesting_Events, Event_Timer);
 Add_Event (Interesting_Events, Event_Gnat_Delay_Until);
```

```
Add_Event (Interesting_Events, Event_Gnat_Return);

Filtered_Events := Interesting_Events;

Complement (Filtered_Events);

-- Asignamos filtro

Set_Filter (Trid, Filtered_Events);

-- Comienza el trazado

Start (Trid);

-- Se activan las tareas

T1.Go;
T2.Go;
T3.Go;


delay 5.0;

-- Se detiene el trazado y se envía la traza al ordenador de desarrollo

Stop (Trid);
Shutdown (Trid);
Close (File);
Send_File ("trace.log");
```

Una vez la traza ha sido generada y enviada al ordenador de desarrollo puede ser analizada. Las siguientes líneas muestran un fragmento de una traza obtenida desde MaRTE OS en formato textual.

```
......
0.010001 :Event_Timer                   1
0.010009 :Event_Mutex_Lock              1 13 31
0.010034 :Event_Ready                   1
0.010050 :Event_Mutex_Lock              0 12 31
0.010071 :Event_Ready                   0
0.010086 :Event_Exec                    1
0.010088 :Event_System_Call_Return      1
0.010098 :Event_System_Call_Mutex_Unlock 1
0.010101 :Event_Mutex_Unlock            1 13
0.010113 :Event_Exec                    0
0.010115 :Event_System_Call_Return      0
0.010124 :Event_System_Call_Mutex_Unlock 0
0.010127 :Event_Mutex_Unlock            0 12
```

```
 0.010130 :Event_System_Call_Return       0
 0.010149 :Event_Gnat_Return             10
 ......
```

Para poder representar visualmente la traza es necesario convertirla al formato requerido por la aplicación Quivi. Las siguientes líneas muestran el fragmento de traza anterior convertido a este formato y su visualización se muestra en la figura 4.2.

```
 ......
10001 {{4  3 #FFFFFF}}
10001 {{3  3}}
10001 {{10 1 TIM #000000}}
10009 {{7  1 {13 (31)}}}
10034 {{0  1}}
10034 {{5  1}}
10050 {{7  0 {12 (31)}}}
10071 {{0  0}}
10071 {{5  0}}
10086 {{4  3 #000000}}
10086 {{3  1}}
10088 {{4  1 #000000}}
10088 {{3  1}}
10098 {{4  1 #AAAAAA}}
10098 {{3  1}}
10101 {{8  1 13}}
10113 {{4  1 #000000}}
10113 {{3  0}}
10115 {{4  0 #000000}}
10115 {{3  0}}
10124 {{4  0 #AAAAAA}}
10124 {{3  0}}
10127 {{8  0 12}}
10130 {{4  0 #000000}}
10130 {{3  0}}
10149 {{10 0 RTS_RET #000000}}
10149 {{4  0 #AAAAAA}}
10149 {{3  0}}
 ......
```

En la figura 4.2 se muestra la activación de las tareas periódicas T1 y T2 en un instante en el que el procesador está ocioso (Línea IDLE en la figura). La prioridad de T1 es mayor que la de T2. La tarea T3 está suspendida y por tanto no interviene. El código que ejecutan estas tareas es el siguiente:

```
task body Periodic_Task is
begin
   loop
```

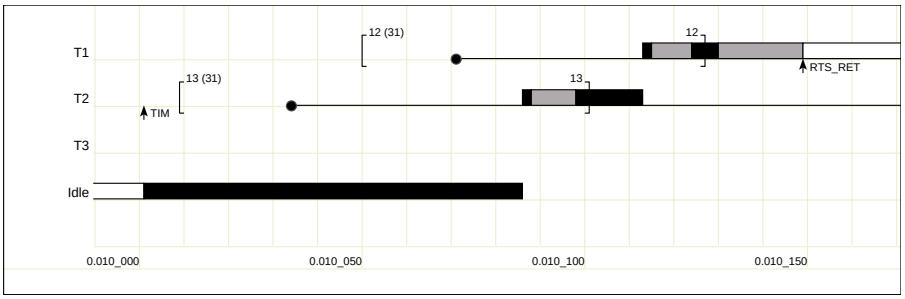


Figura 4.2: Visualización en Quivi.

```
      Do_Something;
      Next_Activation := Next_Activation + Period;
      delay until Next_Activation;
   end loop;
end Periodic_Task;
```

Inicialmente ambas tareas están detenidas tras haber llamado a la sentencia Ada `delay until`, cuya implementación está basada en el uso de la función POSIX `pthread_cond_timedwait`. Esta implementación se ilustra de forma resumida en el siguiente fragmento de código:

```
pthread_mutex_lock (current_task.L);
pthread_cond_timedwait (current_task.C, current_task.L, timeout);
pthread_mutex_unlock (current_task.L);
```

L y C son un respectivamente un mutex y una condición definidos para cada tarea por el soporte del compilador GNAT. El techo de cada uno de estos mutex es igual a la máxima prioridad del sistema (31 para nuestro caso). Dada esta implementación, se observa que las tareas T1 y T2 están en realidad detenidas en la función `pthread_cond_timedwait`.

Cuando vence el temporizador de MaRTE OS ($\uparrow TIM$), las tareas T1 y T2 adquieren su mutex respectivo (identificados como 12 y 13) y son activadas. Dado que su prioridad es igual en este instante, primero se ejecuta la tarea T2, puesto que se ha activado en primer lugar. A continuación, T2 libera su mutex y recupera su prioridad normal, por lo que pasa a ejecución T1. A su vez, T1 libera su mutex, recupera su prioridad normal (mayor que la de T2), concluye la ejecución de la sentencia `delay until` ($\uparrow RTS\_RET$) y prosigue con la ejecución de su código. La ejecución del sistema MaRTE OS se muestra en color negro y en el contexto de la tarea que está en ejecución. La ejecución del soporte de GNAT aparece en color gris.

4.5. Obtención de Métricas Temporales

Una de las posibles aplicaciones de los servicios de trazas POSIX es la obtención de métricas temporales del sistema. Instrumentando adecuadamente el sistema pueden obtenerse métricas temporales de interés, siempre que las secuencias de eventos que se guardan en la traza sean correctamente interpretados. Desafortunadamente, la interpretación de estas secuencias de eventos puede ser virtualmente inviable para un programador que no conozca en detalle la implementación del sistema. Con el fin de resolver este inconveniente, en esta sección se presenta una interfaz que permite a los programadores obtener métricas a partir de la información almacenada en trazas POSIX.

4.5.1. Interfaz

Definimos una métrica como una propiedad temporal del sistema, tal como la duración de un cambio de contexto o el tiempo de ejecución de una tarea. El paquete `Metrics` ofrece al programador una interfaz que permite obtener métricas a partir de los eventos guardados en una traza POSIX.

```
with Ada.Real_Time;
with Ada.Task_Identification;
with Posix.Trace;

package Metrics is

   type Metric                          is private;
   type Metric_Result (Size : Natural) is private;
   type Metric_Result_Element           is private;

   function System_Metric (Metric_Name   : String)
            return Metric;

   function New_User_Metric (Metric_Name    : String;
                             Initial_Event  : POSIX.Trace.Event_Identifier;
                             Final_Event    : POSIX.Trace.Event_Identifier;
                             Include_System : Boolean := True)
            return Metric;

   function Events_Required_By (M    : Metric;
                                Trid : POSIX.Trace.Trace_Identifier)
            return POSIX.Trace.Event_Set;

   procedure Get_Metric_Result (From   : in Metric;
                                Trid   : in POSIX.Trace.Trace_Identifier;
                                Result : in out Metric_Result);
```

```
   function Metric_Result_Element_Count (From : Metric_Result)
            return Natural;

   function Get_Metric_Result_Element (From : Metric_Result;
                                       I    : Natural)
            return Metric_Result_Element;

   function Begin_Time    (From : Metric_Result_Element)
            return Ada.Real_Time.Time;

   function End_Time      (From : Metric_Result_Element)
            return Ada.Real_Time.Time;

   function Length        (From : Metric_Result_Element)
            return Ada.Real_Time.Time_Span;

   function Task_Id       (From : Metric_Result_Element)
            return Ada.Task_Identification.Task_Id;

   function Intermediate_Events (From : Metric_Result_Element)
            return Natural;

end Metrics;
```

En esta interfaz, las métricas se identifican con nombres y se utilizan mediante objetos del tipo `Metric`. Existen dos tipos de métricas, del sistema y del usuario.

Las métricas del sistema están definidas por la implementación y, normalmente, corresponden con propiedades temporales relativas al soporte en tiempo de ejecución o al sistema operativo. A estas métricas se accede utilizando la función `System_Metric`:

```
M1 : Metric;
M2 : Metric;

M1 := System_Metric ("Delay_Until_Wakeup");
M2 := System_Metric ("Schedule_And_Dispath");
```

El programador puede definir sus propias métricas de usuario, las cuales permiten medir el tiempo de ejecución que transcurre entre dos puntos de su programa, estando estos puntos definidos por dos eventos. Estas métricas se crean utilizando la función `New_User_Metric`:

```
M3 : Metric;
Initial_Event : POSIX.Trace.Event_Identifier;
Final_Event   : POSIX.Trace.Event_Identifier;
```

```
Initial_Event := POSIX.Trace.Open ("Initial_Event");
Final_Event   := POSIX.Trace.Open ("Final_Event");

M3 := New_Metric ("My_Metric", Initial_Event, Final_Event);
```

Cada métrica se obtiene a partir de un conjunto de eventos que deben estar presentes en la traza que se está analizando, por lo que es importante que estos eventos sean almacenados en la traza cuando ésta es creada. El programador, cuando prepara el trazado de su aplicación, puede utilizar la función `Events_Required_By` para obtener el conjunto de eventos de los cuales depende una métrica, e indicar así al sistema de trazado que los eventos de este conjunto deben ser almacenados en la traza. El siguiente fragmento de código ilustra cómo puede prepararse una aplicación para crear una traza a partir de la cual poder obtener las métricas en las que el programador está interesado.

```
Attr : POSIX.Trace.Attributes;
Trid : POSIX.Trace.Trace_Identifier;
File : POSIX.IO.File_Descriptor;

Interesting_Events : POSIX.Trace.Event_Set;
All_Events         : POSIX.Trace.Event_Set;


POSIX.Trace.Initialize (Attr);
File := POSIX.IO.Open("Logfile",POSIX.IO.Read_Write);
Trid := POSIX.Trace.Create_With_Log
             (POSIX.Process_Identification.Null_Process_ID,
              Attr,
              File);

POSIX.Trace.Add_Event_Set
   (Interesting_Events, Events_Required_By (M1, Trid));
POSIX.Trace.Add_Event_Set
   (Interesting_Events, Events_Required_By (M2, Trid));
POSIX.Trace.Add_Event_Set
   (Interesting_Events, Events_Required_By (M3, Trid));

POSIX.Trace.Add_All_Events (All_Events);
POSIX.Trace.Set_Filter (Trid, All_Events);
POSIX.Trace.Subtract_From_Filter (Trid, Interesting_Events);

POSIX.Trace.Start (Trid);

Start_Aplication;

POSIX.Trace.Shutdown (Trid);
```

En el anterior fragmento de código, se crea en primer lugar una traza con registro llamando a la función `POSIX.Trace.Create_With_Log`. A continuación se determina el conjunto de eventos necesarios para poder obtener las métricas que desea el programador, en este caso M1, M2 y M3, y se configura la traza para que desestime todos los eventos que se generen exceptuando los eventos de los cuales dependen estas métricas. Una vez preparada la traza, ésta se activa llamando a `POSIX.Trace.Start` y se inicia la aplicación. A partir de este instante, en la traza se almacenarán los eventos no filtrados que genere el sistema. Una vez la aplicación termina se detiene la traza llamando a `POSIX.Trace.Shutdown`. Tras esta acción, en el fichero `logfile` queda almacenada la traza, la cual podrá ser analizada posteriormente y obtener a partir de ella los resultados de las métricas M1, M2 y M3.

El resultado de una métrica es un objeto del tipo `Metric_Result`. Cada uno de estos objetos es una lista formada por todas las mediciones individuales (objetos del tipo `Time_Metric_Result_Element`) detectadas en la traza que está siendo analizada. Una medida individual registra los siguientes cinco valores:

Valor	Descripción
Task_Id	Identificador de la tarea
Length	Tiempo medido
Begin_Time	Instante en el que se detecta el inicio de la medición
End_Time	Instante en el que se detecta la finalización de la medición
Intermediate_Events	Número de eventos detectados al tomar la medición

Los valores `Task_ID` y `Length` son los más importantes de cada medición individual, e indican la tarea para la que se ha obtenido un resultado de métrica y la duración de la medición. `Begin_Time` y `End_Time` indican los instantes de tiempo que delimitan la medición obtenida, y permiten localizar dicha medición en la traza. El valor `Intermediate_Events` puede utilizarse para estimar la sobrecarga del sistema de trazas. Conociendo el tiempo que este sistema consume para generar un evento, puede ajustarse la duración de la medición restando el tiempo que el sistema ha utilizado para generar los eventos de los que depende.

El resultado de una métrica se obtiene llamando al procedimiento `Get_Metric_Result`. Este procedimiento, al ser llamado, recorre la traza indicada en el parámetro `Trid` en busca de mediciones que correspondan con la métrica indicada en el parámetro `From`. Las mediciones encontradas son almacenadas en el parámetro `Result`. El número

de mediciones detectadas puede consultarse llamando a la función `Metric_Result_Element_Count`. La función `Get_Metric_Result_Element` se utiliza para obtener una de las mediciones de un resultado. Los valores de cada medición se consultan utilizando las funciones `Begin_Time`, `End_Time`, `Length`, `Task_Id` y `Intermediate_Events`. A continuación se ilustra cómo el programador obtiene los resultados de una métrica a partir de una traza previamente creada. La tabla 4.1 muestra la salida de este ejemplo.

```
File := POSIX.IO.Open("Logfile",POSIX.IO.Read_Write);
Trid := POSIX.Trace.Open (File);

Get_Metric_Result (M1, Trid, Result);

N := Metric_Result_Element_Count (Result);
for I in 1 .. N loop
   Measure := Get_Metric_Result_Element (Result, I);
   L := Length (Measure);
   T := Task_Id (Measure);
   ....
   Show_Result_Element;
end loop;
```

Task	Length	Begin_Time	End_Time	Int_Events
8	44	14393	14445	5
9	51	24392	24495	5
10	47	24409	24521	5
6	45	24424	24545	5
7	43	24438	24569	5
11	44	45393	45445	5
6	50	49391	49464	5
7	45	49409	49489	5
12	51	54393	54482	5
13	46	54410	54507	5
..	..	.....	.....	.

Tabla 4.1: Resultado de una Métrica. (Valores en μs)

4.5.2. Implementación

La obtención de los resultados de una métrica se realiza recorriendo una traza pregrabada en busca de secuencias de eventos que corresponden a la métrica deseada. Cada

métrica proporcionada por el paquete `Metrics` es analizada por un autómata de estados finitos o, de forma más precisa, por un conjunto de autómatas equivalentes, cada uno de ellos asociado a un identificador de tarea diferente. Estos autómatas son internos a la implementación y, por tanto, los usuarios del paquete `Metrics` no los utilizan directamente.

La utilización de autómatas se justifica dada la naturaleza de la información a tratar, dado que está formada por secuencias de eventos, donde cada evento indica un cambio de estado en el sistema. Otra ventaja de utilizar autómatas estriba en que facilitan la implementación de nuevas métricas, lo que es particularmente importante si se deseara trasladar la implementación del paquete `Metrics` a otro sistema diferente a MaRTE Os.

El alfabeto utilizado por estos autómatas está formado por tuplas de tres elementos (`Event_Type`, `Task_Id`, `Timestamp`). Estas tuplas se obtienen a partir de los eventos almacenados en una traza pre-grabada. Los elementos `Event_Type` y `Task_Id` se utilizan para determinar cuándo se aplica una transición, mientras que el elemento `Timestamp` se utiliza para obtener las mediciones de tiempo.

Una transición entre dos estados se aplica en función de la tupla de entrada y del etiquetado de la transición según las siguientes reglas:

Etiqueta	La transición se aplica si ...
Event_Type, i	el evento de entrada es igual a Event_Type y la tarea de entrada es la tarea asociada al autómata.
Event_Type, not i	el evento de entrada es igual a Event_Type y la tarea de entrada no es la tarea asociada al autómata.
Event_Type	el evento de entrada es Event_Type.
λ	el etiquetado es igual a λ
*	la tupla de entrada no concuerda con ninguna de las reglas anteriores.

Los estádos del autómata pueden ser de uno de estos tipos: $begin$, end, $cancel$, in, out y $cond$. El cálculo de una medición se realiza aplicando las siguientes reglas:

- Regla 1. Al transitar a un estado $begin$, se inicia una nueva medición.
- Regla 2. Al transitar a un estado $cancel$, se descarta la medición actual.
- Regla 3. Al transitar a un estado end, se concluye la medición actual. Los valores asociados a esta medición se determinan de la siguiente forma:

Valor	Cálculo
Task_Id	Identificador de la tarea asociada al autómata.
Begin_Time	Elemento `Timestamp` de la tupla de entrada que causó la aplicación de la regla 1.
End_Time	Elemento `Timestamp` de la tupla de entrada que causó la aplicación de la regla 3.
Length	La suma de la duración de los segmentos detectados por el autómata. Un segmento esta formado por una secuencia de estados formada exclusivamente por estados de tipo in, o por una secuencia de estados que comienza con estados del tipo $cond$ y concluye con estados de tipo in. La duración de un segmento se calcula como la diferencia entre los valores `Timestamp` de las tuplas de entrada que determinaron los estados inicial y final del segmento.
Intermediate_Events	La suma de las longitudes de los segmentos detectados por el autómata. La longitud de un segmento es el número de estados que lo componen menos uno.

Tal como se ha expuesto en las reglas anteriores, una medición comienza cuando el autómata alcanza un estado $begin$. A partir de este punto, se busca en los eventos de la traza una serie de segmentos que determinan la métrica que se desea obtener. Estos segmentos se detectan en función del tipo de los estados del autómata in, out y $cond$. Los estados in y out son estados que están, respectivamente, dentro o fuera de un segmento. Los estados $cond$ son estados cuya pertenencia a un segmento está condicionada a posteriores estados del autómata. Si tras una secuencia de estados $cond$ se alcanza un estado in, la secuencia de estados $cond$ pertenece al segmento, mientras que si se alcanza un estado out, la secuencia de estados $cond$ se descarta. De esta forma, los estados de tipo $cond$ permiten detectar segmentos cuya pertenencia a la medición está condicionada por eventos que todavía no se han obtenido de la traza. Una medición termina cuando se alcanza un estado del tipo end o $cancel$. Si se alcanza un estado end, la duración de la medición es la suma de las duraciones de los segmentos detectados. Por contra, si se alcanza un estado $cancel$, la medición actual se descarta. Este estado se utiliza para poder detectar que la situación actual no corresponde con la métrica que se está obteniendo.

A título de ejemplo, se describe a continuación la construcción de un autómata que permite medir la duración de la activación de una tarea suspendida en una sentencia `delay until`. A esta métrica la denominaremos `Delay_Until_Wakeup`.

En nuestro sistema, cuando una instancia de una tarea ejecuta la sentencia `delay until`, se produce una llamada al soporte en tiempo de ejecución de GNAT, el cual registra el nuevo estado de la tarea y llama a la función POSIX `pthread_cond_timedwait` para que la tarea se suspenda en una condición POSIX. MaRTE OS atiende esta llamada, prepara un evento temporal para que venza en el instante indicado y selecciona una nueva tarea para ejecución. El autómata de la figura 4.3 detecta y mide las acciones que realiza el sistema para activar de nuevo a la tarea que se ha suspendido.

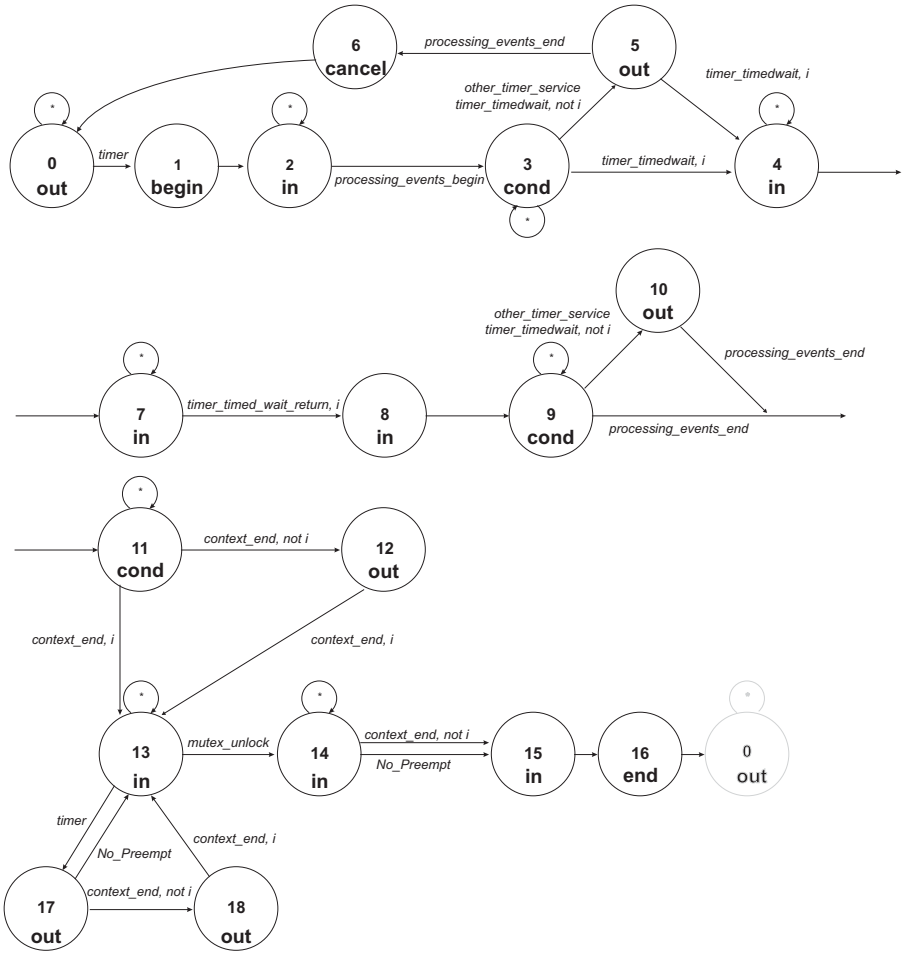


Figura 4.3: Autómata $Delay\_Until\_Wakeup_i$

Este autómata, estando en el estado 0, espera a que se produzca una interrupción del

temporizador hardware. Cuando esta interrupción se produce, se ejecuta el manejador del temporizador de MaRTE OS y comienza una medición (estado 1). Tras una primera fase del manejador del temporizador se alcanza el estado 3 y MaRTE OS comienza a extraer los eventos temporales que hayan vencido en este instante. Si uno de estos eventos implica despertar a la tarea i, el autómata transita al estado 7. Toda la actividad relacionada con otros eventos temporales es descartada por los estados 5 y 10.

Cuando MaRTE OS termina el procesamiento de eventos temporales, si no se ha encontrado el evento temporal que implica despertar a la tarea i, la medición es cancelada en el estado 6 y el autómata queda a la espera de una nueva interrupción en el estado 0. En caso contrario, el autómata transita al estado 11, puesto que la tarea i ha sido despertada. Si la tarea i no es seleccionada para ejecución se alcanza el estado 12 y cualquier actividad, del sistema o de otras tareas, es descartada hasta que la tarea i es seleccionada para ejecución.

Una vez la tarea i continua en ejecución (estado 13), el autómata espera a que el soporte de GNAT libere el mutex que había adquirido al principio de la ejecución de la sentencia `delay until` (estado 14). La medición concluye cuando MaRTE OS decide qué tarea prosigue en ejecución (estado 16). Estando el autómata en el estado 13 es posible que se produzcan nuevas interrupciones del temporizador y se ejecuten otras tareas distintas a i. Los estados 17 y 18 descartan toda la ejecución que pueda producirse debido a estas interrupciones.

4.6. Resumen y Conclusiones

En este capítulo se han realizado diversos trabajos relacionados con los servicios de trazas POSIX definidos por la enmienda 1003.1q.

Estos servicios han sido implementados en MaRTE OS con el fin de disponer de una herramienta de análisis del comportamiento del sistema, así como de un instrumento de medida de tiempos. Como herramienta de análisis, estos servicios han sido cruciales para el diseño y la implementación de las políticas de planificación definidas en el capítulo 3. Como instrumento de medida, han sido utilizados en el análisis de la eficiencia que se realiza en el capítulo 5.

Esta implementación está disponible para la versión 0.86 de MaRTE OS [3] y proyectamos realizar una versión mejorada que incluirá mayor funcionalidad para la gestión de registros de trazas. Esta nueva implementación se incorporaría de forma permanente en una próxima versión del sistema MaRTE OS.

[3] Esta implementación de los servicios de trazas POSIX puede descargarse de la Web de MaRTE OS http://marte.unican.es

Dado que en la enmienda 1003.1q los servicios de trazas están definidos sólo para el lenguaje C, se consideró interesante proponer una interfaz POSIX/Ada para estos servicios. El diseño de esta nueva interfaz se ha realizado a partir de las directrices de diseño de la interfaz POSIX/Ada actual, definida en el estándar POSIX 1003.5c. Las directrices de diseño aplicadas se han discutido en la sección 4.2.1 y la nueva interfaz POSIX/Ada está definida en la sección 4.2.2.

La adecuación de los servicios de trazas a los sistemas de tiempo real ha sido tratada en la sección 4.3. En general se observa que en el diseño de estos servicios se tuvo en cuenta su posible uso en sistemas de tiempo real. No obstante, se han observado algunas deficiencias, entre las que se destacarían las siguientes:

- Los servicios definidos son muy extensos, lo que puede ocasionar un cierto rechazo entre los fabricantes de sistemas de tiempo real, ya que incorporar estos servicios supone aumentar significativamente el tamaño del código del sistema.
- No se ha contemplado la utilización de estos servicios en entornos en los que los sistemas de ejecución y desarrollo son distintos. El estándar POSIX debería tratar este tipo de entornos y definir de forma precisa qué parte de la funcionalidad debería ofrecer por un lado el sistema de ejecución y por otro el de desarrollo.

Estas deficiencias podrían abordarse definiendo un mayor número de opciones de implementación, de tal forma que los servicios de trazas estarían formados por unidades más pequeñas que las actuales. Un sistema operativo de tiempo real podría seleccionar las opciones que más se adecuen a sus necesidades, atendiendo a criterios de funcionalidad y eficiencia.

También en este capítulo se ha propuesto una interfaz para la obtención de métricas temporales del sistema a partir de trazas. La utilidad de esta interfaz reside en que, extraer la información requerida a partir de los eventos almacenados en una traza, no es posible sin conocer cómo el sistema ha sido implementado. De esta forma, se ofrece al programador un mecanismo sencillo de utilizar con el que poder evaluar las prestaciones del sistema que utiliza. La implementación de esta interfaz se ha resuelto mediante autómatas que analizan el contenido de una traza. La utilización de autómatas tiene la ventaja de facilitar la incorporación de nuevas métricas y la implementación de esta interfaz en otros sistemas diferentes a MaRTE OS.

Capítulo 5

Análisis de Eficiencia

En este capítulo se aborda el análisis de la eficiencia de las políticas basadas en la extracción de holgura y en la asignación dual de prioridades definidas en el capítulo 3. Este análisis ha sido realizado a partir de la implementación de estas políticas desarrollada en el sistema operativo MaRTE OS.

La eficiencia de estas políticas ha sido valorada desde dos puntos de vista distintos. En primer lugar, se ha valorado cómo influyen estas políticas en el tiempo de respuesta de las tareas en el peor caso. En segundo lugar, se ha valorado la eficiencia respecto a la sobrecarga que introduce el sistema en un intervalo de funcionamiento normal de la aplicación, es decir, no sólo durante un instante crítico.

En la sección 5.1 se ha caracterizado el impacto que tiene la sobrecarga del sistema operativo MaRTE OS cuando se utiliza la política de planificación basada en prioridades fijas con desalojo. Para ello, se han descrito las actividades del sistema que influyen en el tiempo de respuesta de una tarea y se han introducido estas actividades como términos en la ecuación que permite calcular el tiempo de respuesta para una tarea.

Partiendo de los resultados obtenidos en la sección 5.1, en las secciones 5.2 y 5.3 se ha caracterizado la sobrecarga del sistema que introducen las políticas basadas en la extracción de holgura y en la asignación dual de prioridades respectivamente.

En la sección 5.4 se han aplicado las ecuaciones de cálculo del tiempo de respuesta obtenidas en las secciones anteriores a un caso de estudio. A partir de los servicios de trazas descritos en el capítulo 4, se han obtenido los valores necesarios para aplicar las ecuaciones de cálculo de los tiempos de respuesta. Estas ecuaciones se han aplicado a un mismo conjunto de tareas ejecutado bajo las tres políticas de planificación objeto de estudio y los resultados obtenidos se han comparado con los tiempos de respuesta que se obtendrían en un sistema ideal en el que la sobrecarga del sistema fuese nula.

En la sección 5.5 se ha repetido el estudio anterior utilizando una gestión diferente de las interrupciones del temporizador hardware, basada en el aplazamiento de dichas interrupciones.

En la sección 5.6 se propone una optimización de la política basada en la extracción de holgura cuyo objetivo es reducir los tiempos de respuesta.

El análisis de la sobrecarga que el sistema introduce en un intervalo de funcionamiento normal se ha realizado en la sección 5.7. En esta sección se ha ejecutado un conjunto de tareas críticas bajo las tres políticas de planificación estudiadas y se ha medido la sobrecarga del sistema en un hiperperiodo.

Finalmente, en la sección 5.8 se evalúa la sobrecarga que ocasiona la utilización de secciones de cómputo opcional.

5.1. Análisis de Planificabilidad. Prioridades Fijas con Desalojo

Una aplicación de tiempo real generalmente está formada por un conjunto de tareas periódicas y esporádicas. La ejecución de cada tarea consiste en una sucesión de instancias, donde cada instancia se activa de forma periódica o esporádica. Aplicando a estas tareas a una serie de restricciones, conocemos que el conjunto de tareas de tiempo real es planificable si se cumple que el tiempo de respuesta en el peor caso para cada tarea es inferior a su plazo, es decir:

$$\forall i : R_i < D_i \tag{5.1}$$

Para calcular el peor tiempo de respuesta para una tarea τ_i puede utilizarse la siguiente ecuación [7]:

$$R_i = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{5.2}$$

R_i	tiempo de respuesta para la tarea τ_i.
C_i	tiempo de cómputo en el peor caso de la tarea τ_i.
B_i	factor de bloqueo de la tarea τ_i.
T_j	periodo de la tarea τ_j.
$hp(i)$	conjunto de tareas con prioridad superior a la de la tarea τ_i.

La ecuación 5.2 no incluye en sus términos la sobrecarga del sistema es nula. En cambio, esta sobrecarga puede influir de forma muy determinante en el tiempo de respuesta

de una tarea, llegando en ocasiones a ser superior al tiempo de ejecución realizado a nivel de aplicación [16, 17]. El resto de esta sección se dedica a analizar esta sobrecarga en nuestro sistema, el cual está formado por el sistema de tiempo real MaRTE OS y por el soporte en tiempo de ejecución del compilador GNAT.

MaRTE OS ofrece los servicios POSIX definidos en el perfil de sistema de tiempo real mínimo. El soporte en tiempo de ejecución del compilador GNAT utiliza parte de estos servicios para implementar ciertas construcciones del lenguaje Ada, tales como sentencias `delay`, objetos protegidos, citas, etc. El acceso al soporte en tiempo de ejecución de GNAT es transparente al programador, ya que es el compilador el encargado de transformar ciertas construcciones del lenguaje Ada en las llamadas correspondientes al soporte en tiempo de ejecución. Es importante destacar que esta transformación dificulta notablemente el análisis de la sobrecarga que el sistema origina en la aplicación de tiempo real, puesto que el programador de la aplicación en general desconoce los mecanismos internos del soporte en tiempo de ejecución de GNAT.

Según la clasificación de Katcher [35], MaRTE OS es un sistema dirigido por eventos no integrado, es decir, el temporizador hardware se programa para que interrumpa en el siguiente instante de tiempo relevante, y esta acción puede interrumpir la ejecución de cualquier tarea, independientemente de su prioridad. En este tipo de sistemas, la sobrecarga del sistema en una aplicación de tiempo real se produce de cuatro formas distintas:

- Como parte del tiempo de cómputo en el peor caso C_i.
- Como secciones críticas ejecutadas a la prioridad más alta, dependiendo el factor de bloqueo B_i de la duración de estas secciones críticas.
- Como causa de bloqueo adicional producido por interrupciones.
- Como causa de variación en el instante de activación *(jitter)*.

Estas formas de sobrecarga se describen en detalle a continuación.

Como parte del tiempo de cómputo en el peor caso C_i.

El tiempo de cómputo en el peor caso C_i debe incluir todas las actividades del sistema ocasionadas por la tarea τ_i, tales como la activación de una instancia de la tarea, posibles llamadas que la tarea haga al sistema durante su ejecución y la terminación de una instancia de una tarea. La figura 5.1 ilustra estas actividades del sistema.

En esta figura se muestra la ejecución de dos tareas periódicas, T1 y T2, las cuales utilizan la sentencia `delay until` para implementar su comportamiento periódico. Como puede observarse, la tarea T1 ejecuta dos instancias mientras la tarea T2 se encuentra en ejecución, interrumpiendo por tanto la ejecución de esta tarea. Cuando se produce la

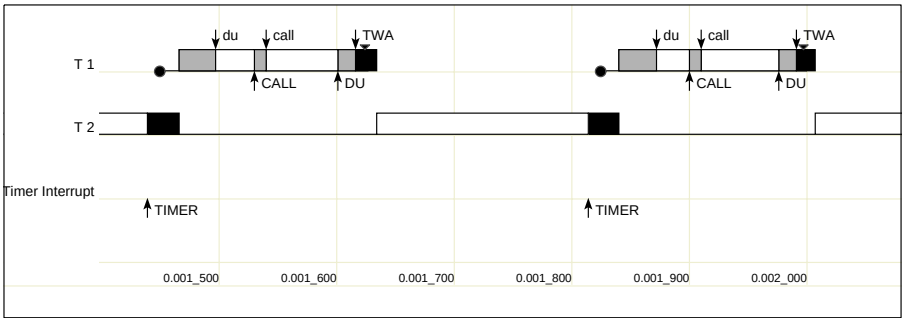


Figura 5.1: Actividades del Sistema Relacionadas con Cada Instancia. (Valores en μs)

interrupción del temporizador hardware (↑TIMER) que despierta a la tarea T1 se ejecuta en el contexto de la tarea interrumpida T2 el manejador de interrupciones del temporizador hardware de MaRTE OS (mostrado en negro). MaRTE OS activa a la tarea T1 y la selecciona para ejecución puesto que es más prioritaria que T2. La tarea T1 prosigue con la ejecución de la sentencia `delay until` en la que se había detenido, y por tanto, ejecutando el código del soporte en tiempo de ejecución de GNAT (mostrado en gris). Cuando termina la ejecución del soporte en tiempo de ejecución de GNAT (↓du), la tarea T1 comienza la ejecución de su código a nivel de aplicación. Durante su ejecución, la tarea T1 puede realizar llamadas al soporte en tiempo de ejecución de GNAT, o a MaRTE OS (↑CALL) y finalmente suspende su ejecución llamando a la sentencia `delay until` (↑DU), lo que de nuevo provoca que se inicie la ejecución del soporte en tiempo de ejecución, el cual solicita la suspensión de la tarea a MaRTE OS llamando a la función `pthread_cond_timedwait` ((↓TWA).

Tal como se observa en la figura 5.1, cada vez que se ejecuta una instancia de la tarea T1, se produce la actividad del sistema descrita en el párrafo anterior. Este tiempo de ejecución forma parte del tiempo de cómputo de la tarea T1 y de la interferencia que T1 produce en tareas menos prioritarias como T2. De esta forma, el tiempo de cómputo en el peor caso C_i está formado por los siguientes componentes:

$$C_i = C_i^{aplic} + C_i^{SIST} \tag{5.3}$$

C_i^{aplic} tiempo de cómputo en el peor caso a nivel de aplicación de la tarea τ_i.
C_i^{SIST} tiempo de cómputo en el peor caso a nivel de sistema de la tarea τ_i.

C_i^{SIST} podemos subdividirlo a su vez en tres componentes, lo cual nos será útil en las siguientes secciones de este capítulo donde se analiza la planificabilidad del sistema en presencia de la extracción de holgura y de la asignación dual de prioridades.

$$C_i^{SIST} = (C_i^{EXP} + C_i^{SINST} + C_i^{FIN}) \tag{5.4}$$

C_i^{EXP} tiempo de cómputo consumido por el sistema para activar a la tarea τ_i expulsando a otra tarea menos prioritaria.

C_i^{SINST} tiempo de cómputo consumido por el sistema atendiendo llamadas de la tarea, excluyendo la llamada para finalizar la instancia actual y reanudar una nueva instancia.

C_i^{FIN} tiempo de cómputo consumido para finalizar la instancia actual.

Como secciones críticas ejecutadas a la prioridad más alta

Ciertas partes de la actividad del sistema deben considerarse como secciones críticas que se ejecutan al máximo nivel de prioridad. Estas secciones se producen cuando el soporte en tiempo de ejecución de GNAT debe proteger sus estructuras de datos internas. En estos casos, el soporte en tiempo de ejecución de GNAT adquiere un mutex POSIX cuyo techo de prioridad es igual a la máxima prioridad del sistema. De esta forma, cualquier tarea puede verse bloqueada como consecuencia de estas secciones críticas. La figura 5.2 ilustra este tipo de situaciones.

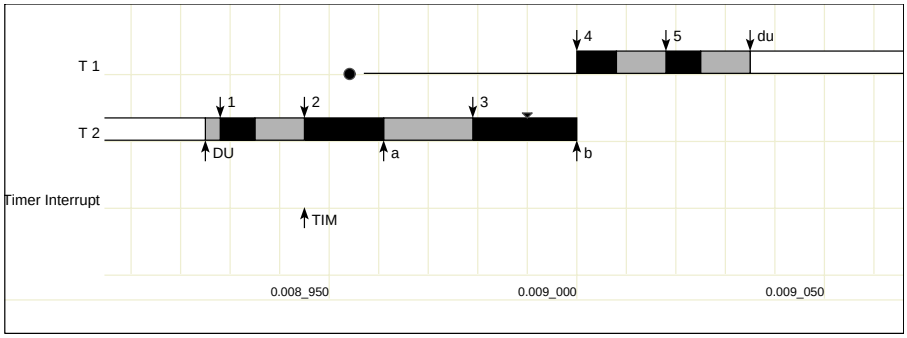


Figura 5.2: Bloqueo Causado por el Sistema. (Valores en μs)

La tarea T2, en el instante ↑DU, ejecuta la sentencia `delay until`, y se inicia la ejecución del soporte de GNAT (mostrado en gris), el cual en el instante ↓1 llama a MaRTE OS (mostrado en negro) para adquirir el mutex que utilizará cuando llame a la función `pthread_cond_timedwait` en el instante ↓3. Cuando en el instante ↓2 se produce una interrupción de reloj MaRTE OS activa a la tarea T1, cuya prioridad base es mayor que la de T2. Una vez concluye el tratamiento de la interrupción, prosigue la tarea T2 puesto que su prioridad actual es la máxima del sistema, al poseer un mutex del soporte de GNAT. De

esta forma, desde el instante ↑a al instante ↑b, la tarea T1 es bloqueada por la actividad de una tarea menos prioritaria, en este caso T2.

Si se utiliza, por ejemplo, el protocolo CSP, dado que el factor de bloqueo B_i para una tarea τ_i es igual a la duración de la sección crítica más larga que puede bloquear a τ_i, es necesario conocer la sección crítica de mayor duración ejecutada por el sistema en favor de tareas menos prioritarias que τ_i. De esta forma, el término B_i de la ecuación 5.2 se calcularía de la forma siguiente:

$$B_i = max((BS_k : k \in lp(i)) \cup B_i^{aplic}) \tag{5.5}$$

B_i^{aplic} bloqueo causado por secciones críticas a nivel de aplicación.
BS_k bloqueo causado por el sistema imputable a la tarea τ_k.

Como causa de bloqueo adicional producido por interrupciones.

Cuando una tarea τ_j se activa por causa de una interrupción puede producirse un bloqueo adicional para tareas más prioritarias que τ_j. En la figura 5.3 se ilustra este tipo de bloqueo en tareas que utilizan la sentencia `delay until` para implementar su activación periódica. Como puede observarse, durante la ejecución de la tarea T1, es posible que se activen tareas menos prioritarias tales como T2, T3 y T4. La rutina de tratamiento de interrupciones de MaRTE OS (mostrada en negro) causa bloqueo a la tarea T1 cada vez que se activa una de estas tareas. Además, puesto que tras la activación estas tareas adquieren un mutex del soporte de GNAT y su prioridad se eleva a la máxima, se ejecutan inmediatamente hasta que liberan dicho mutex. Esta ejecución también genera bloqueo para T1.

En el caso de utilizar programación POSIX, es decir, cuando no interviene el soporte de GNAT, las activaciones de las tareas T2, T3 y T4 producen un efecto notablemente distinto. La figura 5.4 muestra el comportamiento de las tareas cuando utilizan la función `clock_nanosleep` para implementar su activación periódica. En este caso, no se producen cambios de contexto entre tareas ya que no intervienen mutex con techo máximo de prioridad.

Al utilizar Ada sí se producen estos cambios de contexto y, dado que pueden influir de forma negativa en la planificabilidad del sistema, se optó por modificar la implementación que hace MaRTE OS de la función `pthread_cond_timdewait`, de tal forma que la tarea que es despertada adquiera el mutex indicado en esta función una vez pase a ejecución, no antes, evitando así la aparición de estos cambios de contexto Ada.

Tal como puede observarse, este tipo de bloqueo es diferente al descrito en el apartado anterior, puesto que la ejecución de una instancia de una tarea puede verse bloqueada

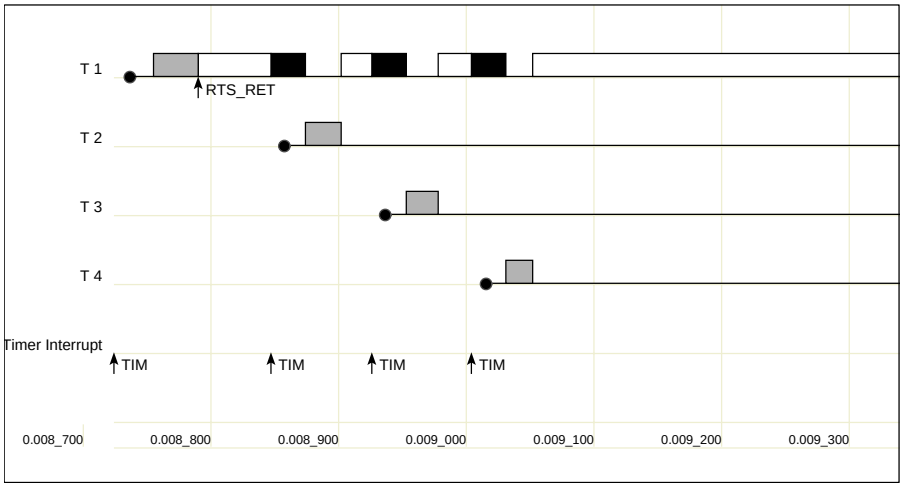


Figura 5.3: Activación de Tareas Menos Prioritarias en Ada. (Valores en μs)

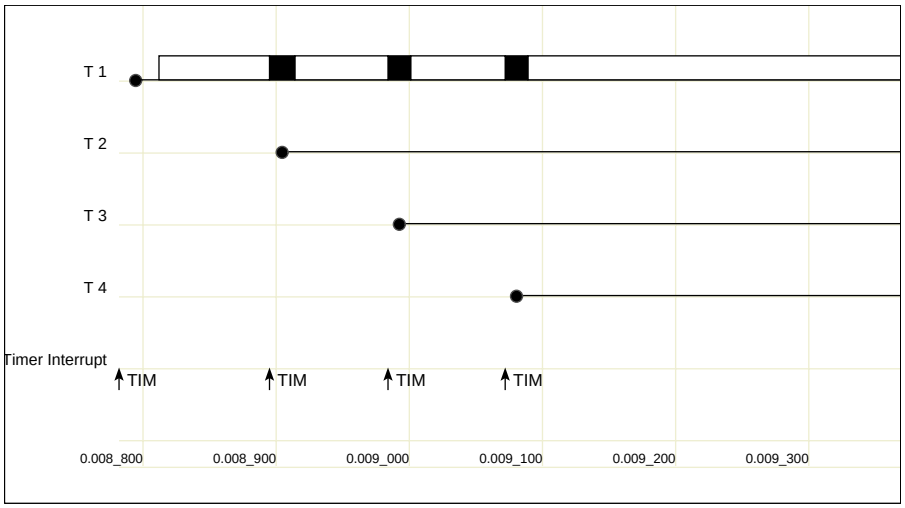


Figura 5.4: Activación de Tareas Menos Prioritarias en POSIX. (Valores en μs)

por varias activaciones de tareas menos prioritarias. Dada esta diferencia, es conveniente considerar este bloqueo de forma explícita como otro término de la ecuación 5.2 al que denominaremos A_i.

$$R_i = C_i + B_i + A_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{5.6}$$

$$A_i = \sum_{\forall k \in lp(i)} \left\lceil \frac{R_i}{T_k} \right\rceil ACT_k \tag{5.7}$$

ACT_k tiempo del sistema necesario para activar la tarea τ_k.

El cálculo de A_i puede simplificarse cuando el plazo de las tareas es inferior a su periodo ($D_i <= T_i$), ya que en estos casos sabemos que para que una tarea τ_k sea planificable es necesario que se cumpla que

$$\forall k \in lp(i) D_k >= R_i \tag{5.8}$$

De esta manera, el término $\left\lceil \frac{R_i}{T_k} \right\rceil$ es siempre igual a 1 y la ecuación 5.7 pasa a ser la siguiente:

$$A_i = \sum_{\forall k \in lp(i)} ACT_k \tag{5.9}$$

Como causa de variación en el instante de activación *(jitter)*.

Generalmente se asume que todas las tareas de la aplicación son activadas con una periodicidad perfecta, es decir, si una tarea τ_i tiene un periodo igual a T_i sus activaciones se producen exactamente a dicha frecuencia. Las tareas esporádicas se incorporan en el modelo considerándolas como tareas periódicas y asumiendo que su periodo T_i es equivalente al tiempo mínimo de activación entre activaciones sucesivas.

Esta suposición, en cambio, puede ser poco realista. Es posible que la periodicidad de una tarea no sea perfecta, lo que ocasiona una diferencia entre el instante de llegada de una tarea y el instante en que dicha tarea realmente se activa. Existen diversas causas que pueden producir esta diferencia: activación de tareas esporádicas mediante consulta, resolución insuficiente del reloj del sistema o secciones en las que el sistema se ejecuta con interrupciones inhibidas.

Cuando estas causas intervienen, puede suceder que la diferencia entre activaciones de una tarea τ_i sea inferior al periodo T_i de dicha tarea. La ecuación 5.2 no contempla

esta posibilidad y su aplicación puede dar como resultado tiempos de respuesta inferiores a los reales. Para corregir este problema se introduce en dicha ecuación el término J_i que representa la variación entre instantes de activación *(jitter)* que una tarea τ_i puede manifestar [7]. Las ecuaciones 5.10 y 5.11 muestran esta corrección.

$$R_i = C_i + B_i + A_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j \tag{5.10}$$

$$R'_i = R_i + J_i \tag{5.11}$$

El sistema operativo MaRTE OS puede producir variación entre activaciones cuando una tarea llega mientras el procesador tiene las interrupciones inhibidas. La figura 5.5 muestra esta situación. En el intervalo $[t_0, t_1]$ el procesador se está ejecutando con interrupciones inhibidas, ya que MaRTE OS está gestionando un cambio de contexto entre las tareas $T2$ y $T3$. En el instante ↑TIM' llega la tarea $T1$, pero dicha llegada es atendida (↑TIM) cuando el procesador habilita las interrupciones en el instante t_1. En este ejemplo, la activación de la tarea $T1$ ha sufrido un retraso igual a la diferencia entre los instantes ↑TIM y ↑TIM'.

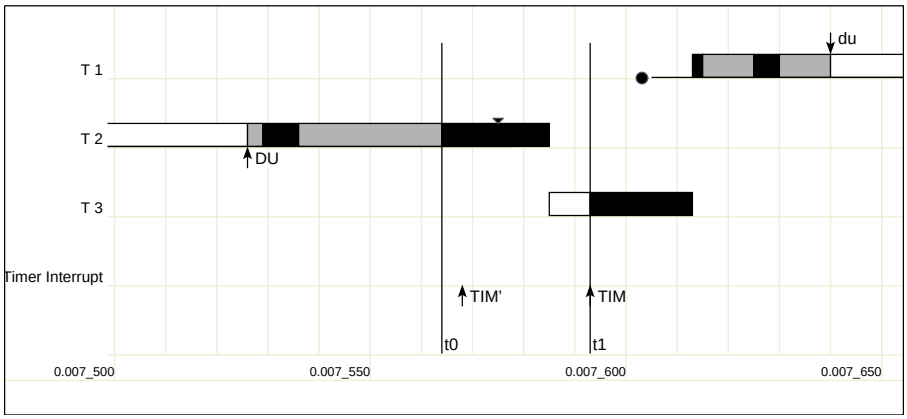


Figura 5.5: Variación del Instante de Activación. (Valores en μs)

5.2. Análisis de Planificabilidad. Extracción de Holgura

En esta sección se describe cómo afecta la incorporación de la extracción de holgura al análisis de la planificabilidad de una aplicación de tiempo real descrito en la sección

5.1.

El peor caso para una tarea τ_i tiene lugar cuando dicha tarea se activa, no existe holgura en su nivel de prioridad o inferiores y se produce un instante crítico para la misma, es decir, todas las tareas más prioritarias que τ_i se activan. Es importante destacar que si existe holgura en su nivel de prioridad e inferiores, el tiempo de respuesta de τ_i puede incrementarse si la holgura es consumida por tareas aperiódicas. No obstante, la propia definición de la holgura garantiza que τ_i no perderá su plazo. De esta forma, para determinar si τ_i es planificable, debemos estudiar su tiempo de respuesta cuando no hay holgura disponible en su nivel de prioridad o inferiores. Bajo estas condiciones analicemos cómo debe modificarse el cálculo del peor tiempo de respuesta.

La implementación de la política basada en la extracción de holgura, tal como ha sido definida en la sección 3.6, ha consistido fundamentalmente en incorporar las siguientes funciones en MaRTE OS.

- Cálculo de la holgura para una tarea.
- Actualización de la holgura de las tareas.
- Temporización de la ejecución limitada por la holgura del sistema.
- Detección de la situación de fin de holgura del sistema.

El cálculo de la holgura para cada tarea se realiza cuando dicha tarea inicia la ejecución de cada una de sus instancias. El tiempo de cómputo de esta función forma parte por tanto del término C_i^{EXP} de la ecuación 5.4.

La holgura se actualiza cada vez que se produce un cambio de contexto en el sistema. Siendo τ_i la tarea que abandona el procesador y τ_j la tarea que pasa a utilizarlo, sabemos que el cambio de contexto se produce, bien porque τ_i es expulsada por τ_j o porque τ_i se suspende y cede el procesador a τ_j. El primer caso se produce cuando τ_j se activa y es más prioritaria que τ_i y, por tanto, el coste de la actualización de la holgura es parte del término C_j^{EXP} para la tarea τ_j (ecuación 5.4). En el segundo caso el coste de la actualización sería parte del término C_i^{FIN} para la tarea τ_i.

Cuando se cede el procesador a una tarea no crítica, MaRTE OS planifica un evento temporizado para limitar la ejecución de dicha tarea a la cantidad de holgura disponible. Dado que esta acción sólo se realiza cuando hay holgura en el sistema, y por tanto el sistema no se encuentra en un instante crítico, su coste no debe considerarse para el análisis de planificabilidad de las tareas críticas.

Finalmente, queda por analizar el efecto de la detección de la situación de fin de holgura. Cuando vence el evento temporizado que limita la ejecución de una tarea no

crítica se produce una interrupción del temporizador hardware que activa a MaRTE OS, el cual restablece la asignación de prioridades de las tareas críticas para que recuperen su prioridad base. En un principio, esta acción puede suceder para cada instancia de una tarea, por lo que sería razonable pensar que su coste forma parte del término C_i^{SIST} en la ecuación 5.3 y del término A_i en la ecuación 5.6. No obstante, analizando detenidamente qué sucede cuando se detecta el fin de la holgura, descubrimos que esto no es correcto.

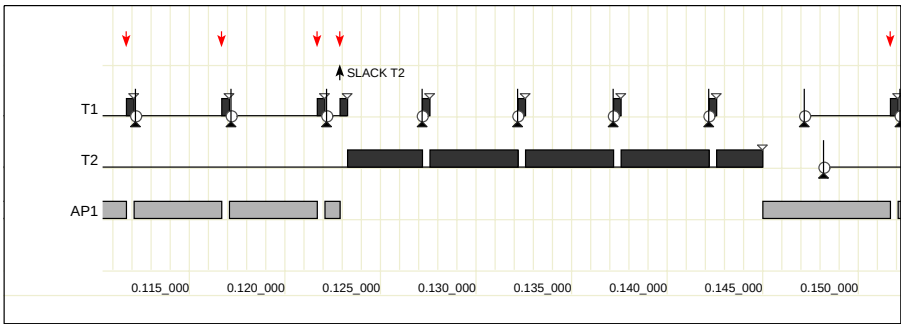


Figura 5.6: Detección de Fin de Holgura

Observando la figura 5.6, podemos ver que en el instante ↑SLACK T2 se detecta que la holgura del sistema se agota debido a que la holgura de la tarea $T2$ se hace igual a cero. Desde este instante hasta que finaliza la instancia actual de la tarea T2 no se produce ninguna otra detección de fin de holgura, debido a que en este intervalo la holgura de T2 no puede hacerse superior a cero. Sería erróneo por tanto considerar la detección del fin de holgura como parte de C_i^{SIST}, puesto que entonces se consideraría como parte de la interferencia que una tarea τ_i causa en otras tareas menos prioritarias. Análogamente, tampoco podemos considerar que forma parte de A_i, puesto que no es posible que tareas menos prioritarias que τ_i generen un evento de fin de holgura puesto que ésta ya se ha agotado. La forma correcta de considerar esta acción del sistema es incluyendo un término adicional $FINSLACK_i$ en la ecuación 5.11 que considere la detección de fin de holgura como una causa de posible retraso en la ejecución de τ_i.

$$R_i = C_i + B_i + A_i + J_i + FINSLACK_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j \qquad (5.12)$$

$FINSLACK_i$ tiempo utilizado por el sistema cuando se detecta que la holgura del sistema se agota debido a que la holgura de la tarea τ_i se hace igual a cero.

5.3. Análisis de Planificabilidad. Asignación Dual de Prioridades

En esta sección se describe cómo afecta la incorporación de la extracción de holgura al análisis de la planificabilidad de una aplicación de tiempo real descrito en la sección 5.1.

El peor caso para una tarea τ_i tiene lugar cuando dicha tarea es promocionada, todavía no ha ejecutado ninguna parte de su instancia actual y en ese momento se produce un instante crítico para la misma, es decir, todas las tareas más prioritarias que τ_i se activan. Bajo estas condiciones analicemos cómo debe variarse el cálculo del peor tiempo de respuesta.

La implementación de la política basada en la asignación dual de prioridades, tal como ha sido definida en la sección 3.7, ha consistido fundamentalmente en incorporar las siguientes funciones en MaRTE OS.

- Asignación de la prioridad baja a una tarea.
- Temporización de la acción de promoción.
- Acción de promoción que devuelve a una tarea su prioridad normal.

MaRTE OS asigna la prioridad baja a una tarea cuando ésta finaliza la ejecución de su instancia actual. De esta manera pasará a estar preparada con esta prioridad cuando se active la siguiente instancia de dicha tarea. El tiempo de cómputo de esta función forma parte, por tanto, del término C_i^{FIN} de la ecuación 5.4.

Cuando se activa una instancia de una tarea se planifica un evento temporal para que se realice la promoción de la tarea. El coste de esta acción es por tanto parte del término C_i^{EXP}.

Cuando vence el plazo del evento temporizado citado en el párrafo anterior, MaRTE OS promociona la tarea correspondiente asignándole su prioridad normal. Esta acción se produce una vez por cada instancia de una tarea, independientemente de si otras tareas están o no promocionadas, tal como se observa en la figura 5.7. En esta figura, desde que la tarea T2 se promociona en t_0 hasta que se suspende en t_1, la tarea T1 se promociona (indicado por el símbolo $\downarrow$) una vez por cada instancia que ejecuta . De esta forma, la promoción de una tarea se convierte en interferencia para tareas menos prioritarias. El coste de promoción $PROMO_i$ debe incluirse, por tanto, en la ecuación 5.4, es decir:

$$C_i^{SIST} = (C_i^{EXP} + C_i^{SINST} + PROMO_i + C_i^{FIN}) \tag{5.13}$$

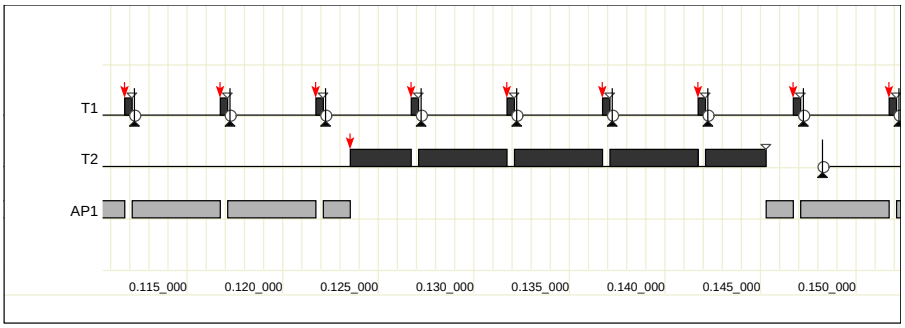


Figura 5.7: Instantes de Promoción

El coste de promoción para una tarea también puede causar bloqueo adicional para tareas de más prioridad, dado que se produce a raíz de una interrupción hardware. Una primera aproximación para incluir este bloqueo consistiría en modificar la ecuación 5.7 de la siguiente forma:

$$A_i = \sum_{\forall k \in lp(i)} \left\lceil \frac{R_i}{T_k} \right\rceil (ACT_k + PROMO_k) \tag{5.14}$$

Esta aproximación tiene, no obstante, el inconveniente de introducir un pesimismo innecesario en el tiempo de respuesta de la tarea τ_i debido a que puede ser imposible que la activación y la promoción de una tarea τ_k se produzcan mientras está en ejecución una sola instancia de la tarea τ_i. En particular, si $PROMO_k > Ri$ no puede suceder que la activación y la promoción de τ_k bloqueen la ejecución de una instancia de la tarea τ_i. En este caso, el bloqueo máximo producido sería igual a $max(ACT_k, PROMO_k)$ y no a $(ACT_k + PROMO_k)$. Esta consideración plantea algunas dificultades, ya que para conocer el tiempo de respuesta de una tarea τ_i es necesario conocer primero el tiempo de promoción, y por tanto el de respuesta de sus tareas menos prioritarias. Para resolver este problema basta con comenzar el cálculo de los tiempos de respuesta por la tarea menos prioritaria, para la cual el termino A_i es igual a cero, y continuar este cálculo en orden inverso de prioridad.

5.4. Comparación de Algoritmos Respecto a la Planificabilidad

En esta sección se estudia el impacto que tiene la sobrecarga del sistema en la planificabilidad de aplicaciones de tiempo real en función del tipo de algoritmo de planificación

utilizado.

Para realizar este estudio se ha utilizado el conjunto de tareas periódicas mostrado en la tabla 5.1. Estas tareas hacen uso del sistema tan sólo para implementar su activación periódica. El estudio se ha realizado tanto en Ada como en POSIX. En el primer caso las tareas utilizan la sentencia `delay until` y en el segundo la función `clock_nanosleep`. El cómputo mostrado en la tabla corresponde exclusivamente al nivel de aplicación. Estas tareas no comparten recursos de forma explícita, por lo que no existe factor de bloqueo a nivel de aplicación. Esta decisión es debida a que nuestro interés se centra en el bloqueo que pueda ocasionar el sistema.

Tarea	Prioridad	Periodo	Plazo	Cómputo Peor Caso
1	26	200000	5000	750
2	25	25000	25000	500
3	24	25000	25000	1250
4	23	40000	40000	250
5	22	50000	50000	750
6	21	50000	50000	1250
7	20	50000	50000	1750
8	19	80000	80000	2250
9	18	80000	80000	500
10	17	100000	100000	1250
11	16	200000	200000	250
12	15	200000	200000	750
13	14	200000	200000	250
14	13	200000	200000	250
15	12	200000	200000	750

Tabla 5.1: Conjunto de Tareas de Ejemplo

Para aplicar las ecuaciones de cálculo de tiempos de respuesta descritas en las secciones anteriores es necesario obtener valores para los términos de estas ecuaciones en los que interviene la sobrecarga del sistema. Con este fin se han utilizado los servicios de trazas POSIX y el paquete `Metrics` descritos en el capítulo 4, definiéndose las siguientes métricas:

Ada_Delay_Until_Wakeup_No_Preempt Tiempo transcurrido desde la interrupción del temporizador hardware, que indica el vencimiento del plazo de la función `pthread_cond_timedwait`, hasta que finaliza la ejecución de la sentencia `delay_until`. Condicionado a que tras la interrupción la tarea despertada no expulse a

la que actualmente está en ejecución. Se excluyen los segmentos en los cuales la ejecución del soporte de GNAT es interrumpido y, de producirse, las posibles ejecuciones de otras tareas. Corresponde con el término ACT_k de la ecuación 5.7.

Ada_Delay_Until_Wakeup_Preempt Tiempo transcurrido desde la interrupción del temporizador hardware, que indica el vencimiento del plazo de la función `pthread_cond_timedwait`, hasta que finaliza la ejecución de la sentencia `delay_until`. Condicionado a que tras la interrupción la tarea despertada expulse a la que actualmente está en ejecución. Se excluyen los segmentos en los cuales la ejecución del soporte de GNAT es interrumpido y, de producirse, las posibles ejecuciones de otras tareas. Corresponde con el término C_i^{EXP} de la ecuación 5.4.

Ada_Delay_Until_Sleep Tiempo transcurrido desde el inicio de la sentencia `delay_until` hasta que se produce el cambio de contexto a otra tarea. Se excluyen los segmentos en los cuales la ejecución del soporte de GNAT es interrumpido y, de producirse, las posibles ejecuciones de otras tareas. Corresponde con el término C_i^{FIN} de la ecuación 5.4.

POSIX_Clock_Nanosleep_Wakeup_No_Preempt Tiempo transcurrido desde la interrupción del temporizador hardware, que indica el vencimiento del plazo de la función `clock_nanosleep`, hasta que finaliza el tratamiento de dicha interrupción. Condicionado a que tras la interrupción la tarea despertada no expulse a la que actualmente está en ejecución. Corresponde con el término ACT_k de la ecuación 5.7.

POSIX_Clock_Nanosleep_Wakeup_Preempt Tiempo transcurrido desde la interrupción del temporizador hardware, que indica el vencimiento del plazo de la función `clock_nanosleep`, hasta que finaliza la ejecución de dicha función. Condicionado a que tras la interrupción la tarea despertada expulse a la que actualmente está en ejecución. Corresponde con el término C_i^{EXP} de la ecuación 5.4.

POSIX_Clock_Nanosleep_Sleep Tiempo transcurrido desde el inicio de la función `clock_nanosleep` hasta que se produce el cambio de contexto a otra tarea. Corresponde con el término C_i^{FIN} de la ecuación 5.4.

Ada_System_Blocking Tiempo transcurrido desde que el soporte de GNAT adquiere un mutex no teniendo ninguno hasta que libera el último de sus mutex. Se entiende como instante de liberación, el instante en que se decide no realizar un cambio de contexto o, si éste se produce, cuando finaliza dicho cambio de contexto. Se excluyen los segmentos en los cuales la ejecución del soporte de GNAT es interrumpido y, de producirse, las posibles ejecuciones de otras tareas. Corresponde con el término BS_k de la ecuación 5.5.

Interrupt_Service Tiempo transcurrido desde que el procesador deshabilita interrupciones teniéndolas habilitadas hasta que las habilita. Corresponde con el término J_i de la ecuación 5.11.

Slack_Exhausted Tiempo transcurrido desde la interrupción del temporizador hardware, que indica la situación de fin de holgura, hasta que se decide no realizar un cambio de contexto o, si éste se produce, hasta que finaliza dicho cambio de contexto. Corresponde con el término $FINSLACK_i$ de la ecuación 5.12.

Promotion Tiempo transcurrido desde la interrupción del temporizador hardware, que indica que una tarea debe promocionarse, hasta que se decide no realizar un cambio de contexto o, si éste se produce, hasta que finaliza dicho cambio de contexto. Corresponde con el término $PROMO_i$ de la ecuación 5.13.

Para obtener estas métricas se ejecutó, en un procesador Pentium MMX a 200 Mhz, el conjunto de tareas descritas en la tabla 5.1 bajo los tres algoritmos de planificación objeto de estudio: el sistema normal, la asignación dual de prioridades y la extracción de holgura. Los valores máximos encontrados se han utilizado para determinar los valores de los términos que forman parte de las ecuaciones de cálculo de los tiempos de respuesta. Las figuras 5.8 y 5.9 muestran estos valores obtenidos para Ada y POSIX respectivamente. Examinando estos resultados cabe realizar las siguientes valoraciones.

La sobrecarga del soporte de GNAT es muy significativa. Si comparamos los resultados obtenidos para el sistema normal en Ada y POSIX se observa que los términos en los que influye el soporte de GNAT llegan a duplicarse, tales como la activación C_{EXP} y la terminación C_{FIN} de una instancia. Estas diferencias no son tan acusadas en otros términos, lo que indica que el coste de las funciones `pthread_cond_timedwait` (en la que se basa la sentencia `delay until`) y `clock_nanosleep` son similares. El bloqueo de sistema BS, presente sólo en el caso Ada, corresponde con la primera parte de la ejecución de la sentencia `delay until`, desde que la tarea adquiere el mutex del soporte de GNAT hasta que se produce el cambio de contexto. Los resultados obtenidos para la asignación dual de prioridades y la extracción de holgura no dependen de utilizar Ada o POSIX, ya que mantienen similares diferencias respecto al sistema normal. Esto es razonable, ya que la implementación de estos algoritmos ha sido realizada a nivel de MaRTE OS y no a nivel del soporte de GNAT.

La sobrecarga que introduce la asignación dual de prioridades se observa que es escasa, siendo más notable en el término C_{EXP}. Este término recoge la planificación del evento temporizado que promocionará a la tarea y una posible programación del temporizador hardware, siendo la acción más costosa, por tanto, que realiza este algoritmo.

Por contra, la sobrecarga que introduce la extracción de holgura es muy significativa.

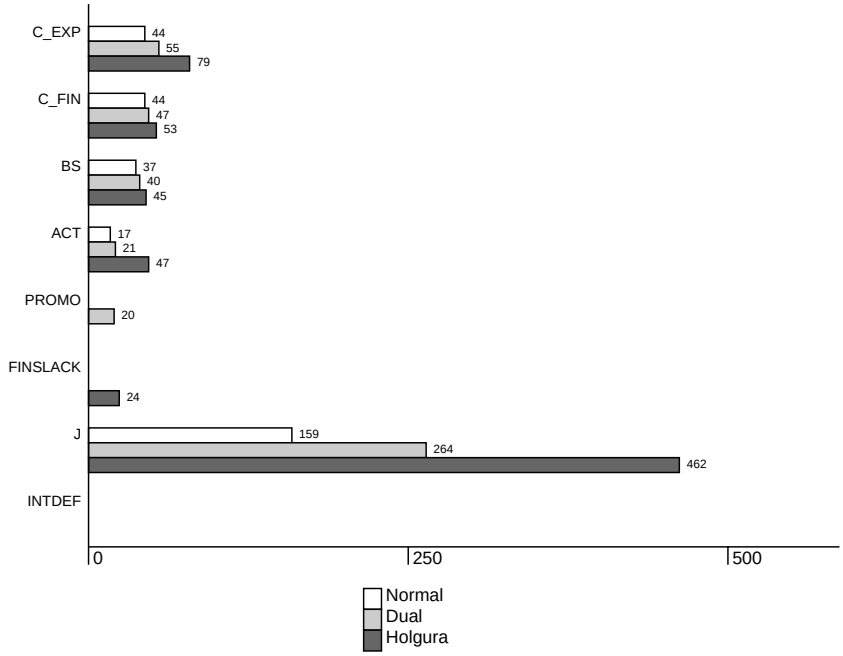


Figura 5.8: Sobrecarga del Sistema. Caso Ada. (Valores en μs)

El cálculo de la holgura de la tarea aumenta notablemente el resultado obtenido para los términos C_{EXP} y ACT.

Por último, es importante discutir los resultados obtenidos para el retardo de activación J. El motivo de estos valores tan elevados estriba en que reflejan la situación en la que todas las tareas se activan de forma simultánea. En este caso, MaRTE OS procesa todas estas activaciones en el contexto de una interrupción del temporizador hardware y, por tanto, las interrupciones del procesador están deshabilitadas mientras se activan una tras otra todas las tareas. Esta situación se agrava en el caso de la asignación dual de prioridades y mucho más en el caso de la extracción de holgura, puesto que la activación de una tarea es la acción más costosa para ambos algoritmos.

Es importante resaltar que estos valores tan elevados obtenidos para el retardo de activación introducen un grado de pesimismo innecesario, especialmente para las tareas más prioritarias, por el siguiente motivo. Hay que tener en cuenta que la activación de tareas menos prioritarias aparece en este caso duplicada en la ecuación del cálculo de respuesta. Por un lado forma parte del retardo de activación J tal como hemos visto y, por otro, como parte del término A_i, y en la práctica es imposible que una tarea sufra más

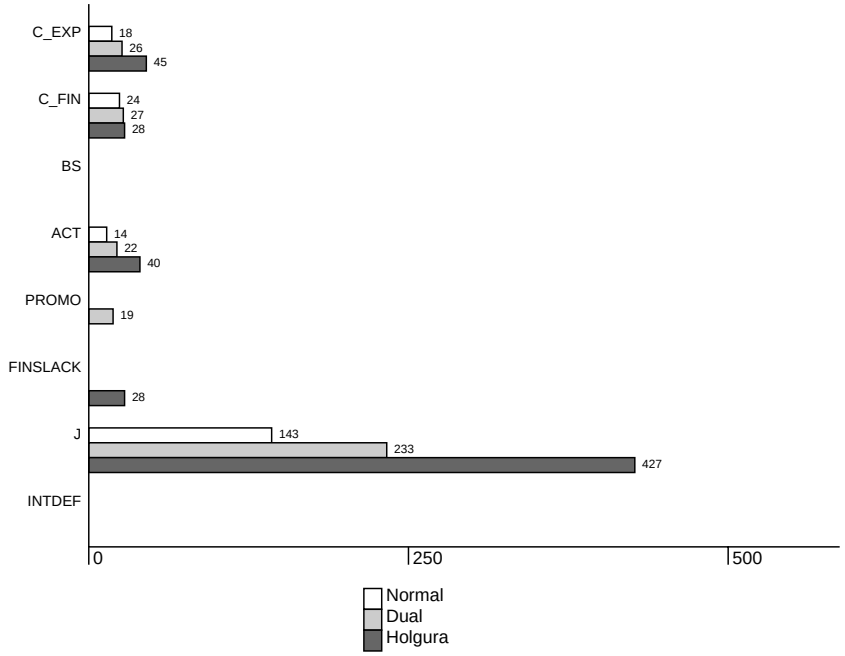


Figura 5.9: Sobrecarga del Sistema. Caso POSIX. (Valores en μs)

de una activación por parte de otra tarea de menor prioridad. Desafortunadamente, no es sencillo tratar de forma correcta este problema en la ecuación del cálculo del tiempo de respuesta, ya que habría que establecer en ella una correlación entre los términos J y A_i.

Aplicando estos resultados, se han calculado los tiempos de respuesta para nuestro conjunto de tareas ejemplo. Las tablas 5.2 y 5.3 muestran los resultados obtenidos para Ada y POSIX, respectivamente, y en cada caso para los algoritmos objeto de estudio. Se ha incorporado también el tiempo de respuesta que se obtendría en una situación ideal en la cual la sobrecarga del sistema fuese cero, situación que hemos denominado sistema teórico. El objeto de incluir este caso es poder observar en qué forma afecta la sobrecarga del sistema a la planificabilidad para los tres casos. Las figuras 5.10 y 5.11 muestran las diferencias existentes entre los tiempos de respuesta de los tres algoritmos objeto de estudio y el sistema teórico.

En estas figuras se observa de nuevo las importantes diferencias que existen entre Ada y POSIX, así como la sobrecarga adicional que introducen la asignación dual de prioridades y, especialmente, la extracción de holgura. No obstante, es importante resaltar que la asignación dual de prioridades se aleja considerablemente del sistema normal en

las tareas menos prioritarias, acercándose en este caso a la extracción de holgura. La explicación a este comportamiento reside, en primer lugar, en que en ambas políticas el término C_i^{SIST} es similar, ya que en la política basada en la asignación dual de prioridades este término incluye el término $PROMO_i$, lo que compensa los valores obtenidos por la extracción de holgura en los términos C_i^{EXP} y C_i^{FIN}. En segundo lugar, el término A_i influye en menor medida en las tareas menos prioritarias, lo que reduce las diferencias entre estas políticas.

	Teo	Normal	Dual	Holgura
1	750	1267	1464	2071
2	1250	1838	2065	2655
3	2500	3160	3416	3990
4	2750	3481	3767	4324
5	3500	4303	4619	5158
6	4750	5624	5970	6493
7	6500	7446	7821	8327
8	8750	9767	10172	10661
9	9250	10339	10774	11246
10	10500	11660	12125	12580
11	10750	11981	12476	12914
12	11500	12803	13328	13749
13	11750	13124	13679	14083
14	12000	13446	14030	14418
15	12750	14267	14881	15252

Tabla 5.2: Tiempos de Respuesta. Caso Ada. (Valores en μs)

	Teo	Normal	Dual	Holgura
1	750	1129	1366	1839
2	1250	1658	1915	2373
3	2500	2936	3215	3656
4	2750	3215	3514	3939
5	3500	3993	4313	4722
6	4750	5271	5613	6006
7	6500	7050	7412	7789
8	8750	9328	9712	10072
9	9250	9856	10261	10606
10	10500	11135	11560	11889
11	10750	11413	11860	12172
12	11500	12191	12659	12955
13	11750	12470	12958	13239
14	12000	12748	13258	13522
15	12750	13526	14057	14305

Tabla 5.3: Tiempos de Respuesta. Caso POSIX. (Valores en μs)

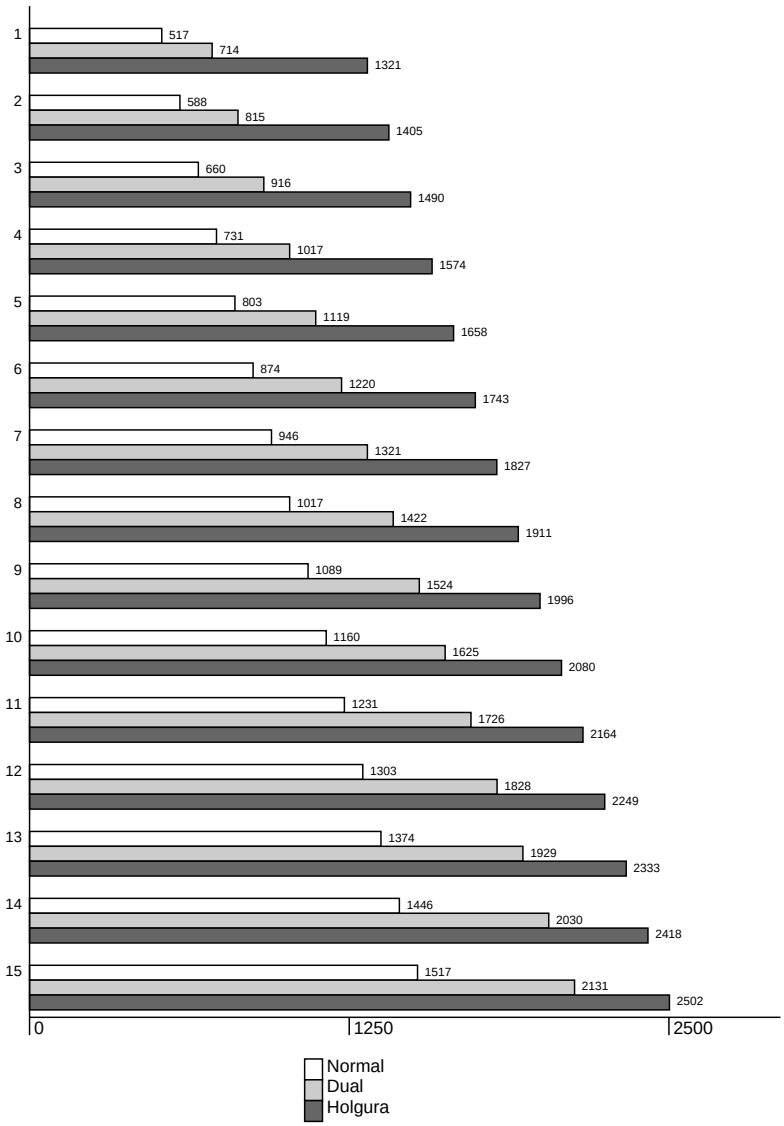


Figura 5.10: Diferencias en Tiempos de Respuesta. Caso Ada.(Valores en μs)

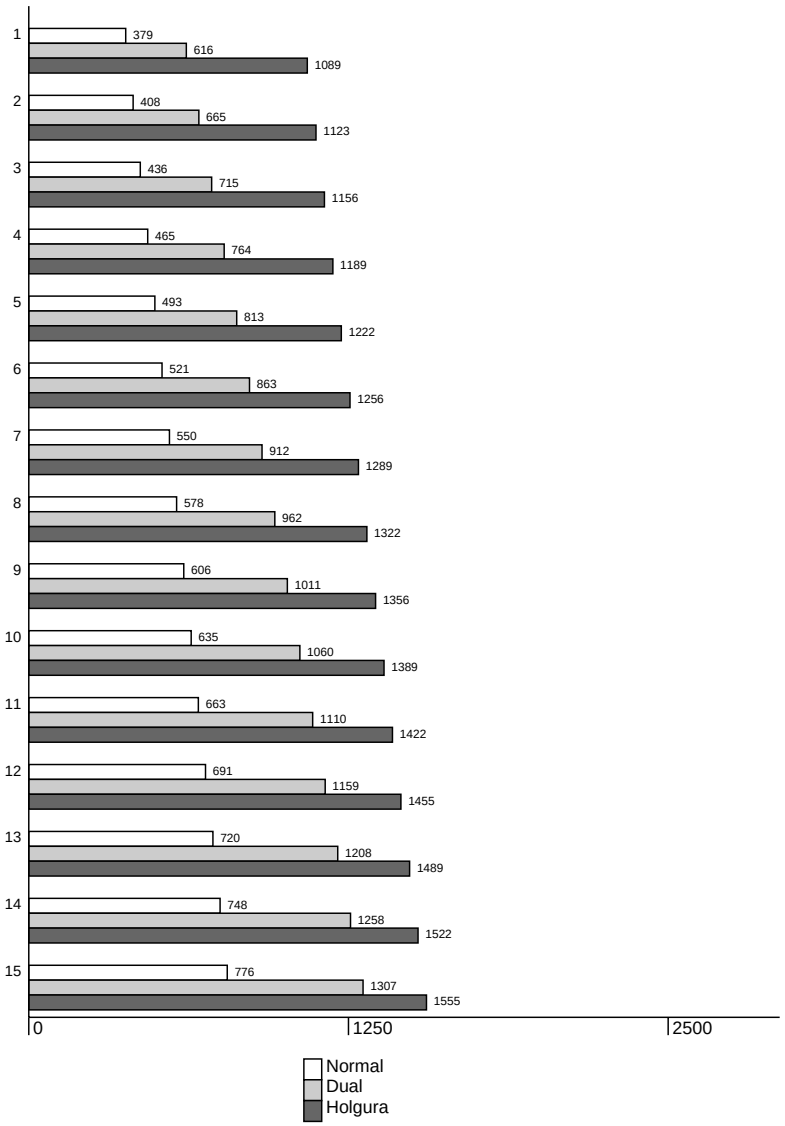


Figura 5.11: Diferencias en Tiempos de Respuesta. Caso POSIX. (Valores en μs)

5.5. Utilización de Interrupciones Diferidas

Tal como se ha observado en la sección anterior, los términos que más influyen en el tiempo de respuesta de las tareas, especialmente de las tareas más prioritarias, son la activación de tareas de menor prioridad A_i y el retardo de activación J_i. Es posible disminuir los valores de estos términos si se utiliza la técnica de interrupciones diferidas presente en otros sistemas, como por ejemplo RT-Linux [71].

Esta técnica consiste en asociar a cada evento temporal una prioridad. En principio se asocia a todos los eventos temporales la prioridad máxima del sistema, excepto a aquellos eventos temporales que se sabe con seguridad que van a causar que una tarea suspendida pase a estado preparado. En estos casos, a dichos eventos temporales se les asocia la prioridad de la tarea que van a despertar. Un ejemplo de este tipo de eventos temporales puede ser el evento temporal que despertará a una tarea suspendida en una sentencia `delay until`.

En los instantes en los que se debe programar el temporizador hardware, éste es programado con el evento temporal más cercano cuya prioridad sea superior a la de la próxima tarea en ejecución. En el caso de no existir ningún evento que cumpla esta condición, el temporizador hardware simplemente no es programado. De esta forma, ciertos eventos temporales pueden quedar aplazados. Cuando la tarea en ejecución abandona el procesador y se determina la nueva tarea en ejecución, se examina si hay algún evento temporal aplazado con mayor prioridad que la nueva tarea en ejecución y, de ser así, se procesa.

Como consecuencia de aplicar esta técnica, el término A_i sólo debe considerar la activación de tareas de menor prioridad que no utilicen eventos temporales para activarse, por ejemplo, tareas esporádicas que se activen a raíz de una interrupción externa. Si todas las tareas menos prioritarias son periódicas, el término A_i se hace igual a cero.

Otra ventaja de esta técnica es que puede reducir el tiempo que el procesador permanece con interrupciones deshabilitadas. Tal como se observó en la sección anterior, el intervalo más largo en el cual las interrupciones son deshabilitadas es consecuencia de la activación simultánea de todas las tareas del sistema. Cuando se utilizan las interrupciones diferidas esta activación simultánea no se produce, ya que sólo una tarea es activada en este caso. De esta forma, el retardo de activación J_i puede verse notablemente reducido.

Aunque esta técnica tiene un coste asociado, puesto que es necesario realizar operaciones de búsqueda en la cola de eventos temporales para poder implementarla, sus beneficios parecen interesantes, por lo que se ha optado por incorporarla en MaRTE OS y repetir el estudio realizado en la sección anterior.

Antes de proseguir, es necesario estudiar cómo debe incorporarse el coste de esta técnica en la ecuación de cálculo del tiempo de respuesta. Las acciones asociadas a esta

técnica forman parte de los términos C_i^{EXP} y C_i^{FIN}. Las modificaciones realizadas en el manejador de interrupciones del temporizador hardware están contempladas en el término C_i^{EXP}. Por otra parte, la programación del temporizador hardware está contemplada en el término C_i^{FIN}. Finalmente, debemos analizar cómo incorporar el procesamiento de eventos temporales aplazados. Si analizamos un instante crítico para una tarea τ_i, podemos concluir que el procesamiento de eventos temporales aplazados se produce cuando finaliza la instancia actual de la tarea τ_i, pero no cuando finalizan las instancias de tareas más prioritarias que τ_i. Esto es debido a que cualquier evento temporal que pueda estar aplazado tendrá una prioridad inferior a la de τ_i, puesto que ésta se encuentra preparada para ejecución. Con el fin de considerar este coste debemos modificar las ecuaciones 5.11 y 5.12 para que incorporen un nuevo término específico que trate este coste:

$$R_i = C_i + B_i + A_i + J_i + INTDIF_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j \tag{5.15}$$

$$R_i = C_i + B_i + A_i + J_i + FINSLACK_i + INTDIF_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j \tag{5.16}$$

$INTDIF_i$ tiempo del sistema necesario para procesar interrupciones diferidas.

Puesto que el coste del procesamiento de interrupciones diferidas no forma parte del término C_i^{FIN}, se ha optado por sustituir las métricas `Ada_Delay_Until_Sleep` y `POSIX_Clock_Nanosleep_Sleep` por las métricas que se definen a continuación:

Ada_Delay_Until_Sleep_No_Int_Deferred Tiempo transcurrido desde el inicio de la sentencia `delay_until` hasta que se produce el cambio de contexto a otra tarea. Condicionado a que no se produzca el tratamiento de interrupciones diferidas. Se excluyen los segmentos en los cuales la ejecución del soporte de GNAT es interrumpido y, de producirse, las posibles ejecuciones de otras tareas. Corresponde con el término C_i^{FIN} de la ecuación 5.4.

POSIX_Clock_Nanosleep_Sleep_No_Int_Deferred Tiempo transcurrido desde el inicio de la función `clock_nanosleep` hasta que se produce el cambio de contexto a otra tarea. Condicionado a que no se produzca el tratamiento de interrupciones diferidas. Corresponde con el término C_i^{FIN} de la ecuación 5.4.

El coste del procesamiento de interrupciones diferidas se ha calculado restando estas nuevas métricas de las utilizadas anteriormente.

Las figuras 5.12 y 5.13 muestran los valores obtenidos para los términos de las ecuaciones de cálculo del tiempo de respuesta. Los tiempos de respuesta obtenidos para el conjunto de tareas de la tabla 5.1 se detallan en las tablas 5.4 y 5.5, y las figuras 5.14 y 5.15 muestran las diferencias en los tiempos de respuesta respecto al sistema teórico [1].

Comparando las figuras 5.12 y 5.13 con las figuras respectivas 5.8 y 5.9, podemos observar que la mayoría de términos mantienen valores similares (algo superiores en algunos casos) excepto los términos ACT y J. El término ACT es nulo en este nuevo caso, dado que todas las tareas son periódicas. El término J se ha reducido notablemente puesto que ya no se produce la activación simultánea de todas las tareas del sistema. En lo referente al nuevo término $INTDIF$, vemos como el valor más elevado corresponde a la extracción de holgura, dado que cuando se procesa una interrupción diferida se calcula la holgura para la tarea que es despertada.

Comparando las figuras 5.14 y 5.15 con las figuras 5.10 y 5.11, podríamos destacar lo siguiente. Los tiempos de respuesta mejoran en todos los casos, gracias a la ausencia del término A_i y a la disminución del retardo de activación J_i. Las diferencias existentes entre la asignación dual de prioridades y la extracción respecto al sistema normal, se hacen más acusadas conforme las tareas tienen menos prioridad. Finalmente, las diferencias entre la asignación dual de prioridades y la extracción de holgura siguen siendo significativas, pero se han reducido de forma notable. Por ejemplo, la diferencia entre ambas políticas para el tiempo de respuesta de la tarea 1 5.11 es igual a 473 μs en la figura, mientras que en la figura 5.15 es igual a 60 μs.

	Teo	Normal	Dual	Holgura	Holgura2
1	750	940	987	1045	972
2	1250	1530	1615	1675	1567
3	2500	2870	2992	3055	2912
4	2750	3210	3370	3436	3257
5	3500	4050	4248	4316	4102
6	4750	5390	5626	5696	5447
7	6500	7230	7504	7576	7291
8	8750	9570	9882	9956	9636
9	9250	10160	10510	10586	10231
10	10500	11500	11887	11967	11576
11	10750	11840	12265	12347	11921
12	11500	12680	13143	13227	12766
13	11750	13020	13521	13607	13111
14	12000	13359	13899	13987	13456
15	12750	14199	14777	14868	14301

Tabla 5.4: Tiempos de Respuesta. Caso Ada con Interrupciones Diferidas. (Valores en μs)

[1] En estas tablas y figuras aparece un nuevo algoritmo (holgura2) descrito en la siguiente sección.

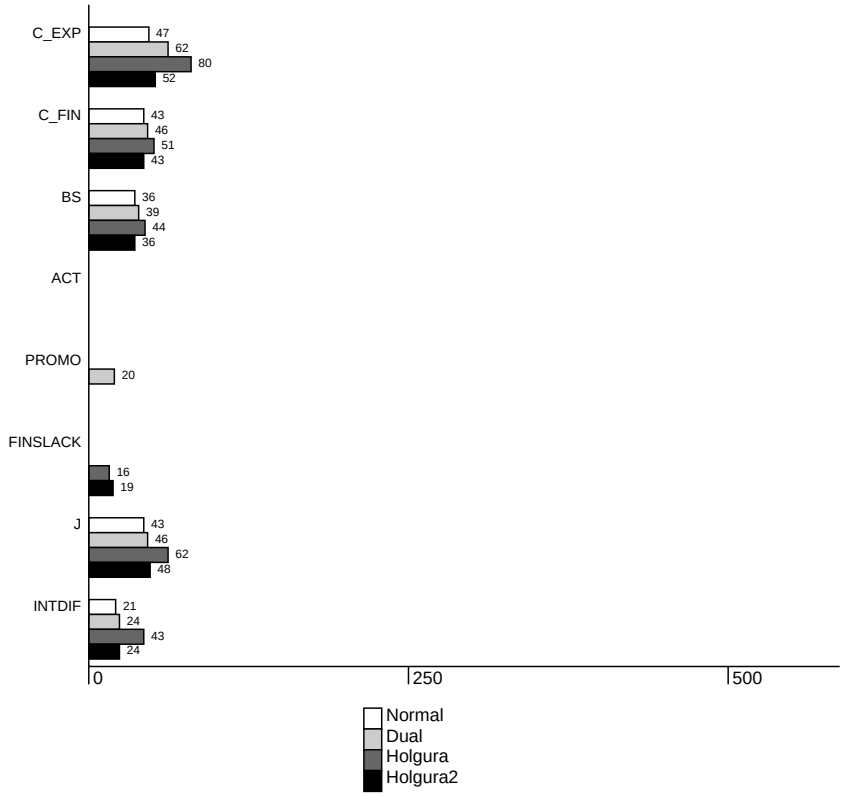


Figura 5.12: Sobrecarga del Sistema. Caso Ada con Interrupciones Diferidas. (Valores en μs)

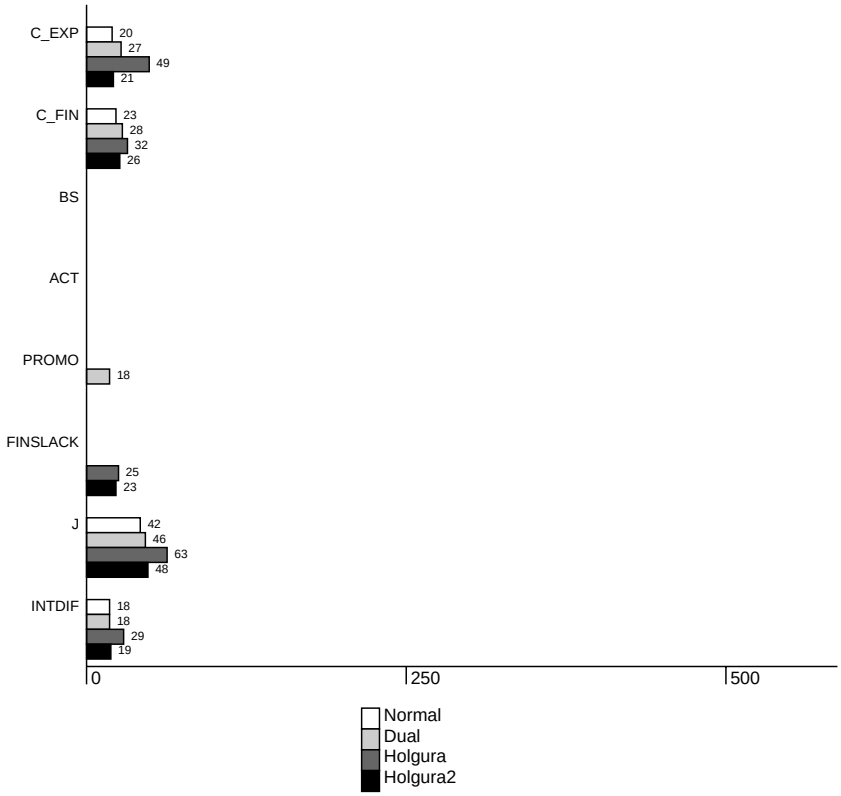


Figura 5.13: Sobrecarga del Sistema. Caso POSIX con Interrupciones Diferidas. (Valores en μs)

	Teo	Normal	Dual	Holgura	Holgura2
1	750	852	887	947	887
2	1250	1394	1461	1528	1434
3	2500	2687	2784	2860	2731
4	2750	2979	3108	3191	3028
5	3500	3771	3931	4022	3825
6	4750	5063	5255	5353	5122
7	6500	6856	7078	7184	6919
8	8750	9148	9401	9515	9217
9	9250	9690	9975	10096	9764
10	10500	10982	11298	11427	11061
11	10750	11275	11622	11758	11358
12	11500	12067	12445	12589	12155
13	11750	12359	12769	12920	12452
14	12000	12651	13092	13251	12749
15	12750	13443	13916	14082	13546

Tabla 5.5: Tiempos de Respuesta. Caso POSIX con Interrupciones Diferidas. (Valores en μs)

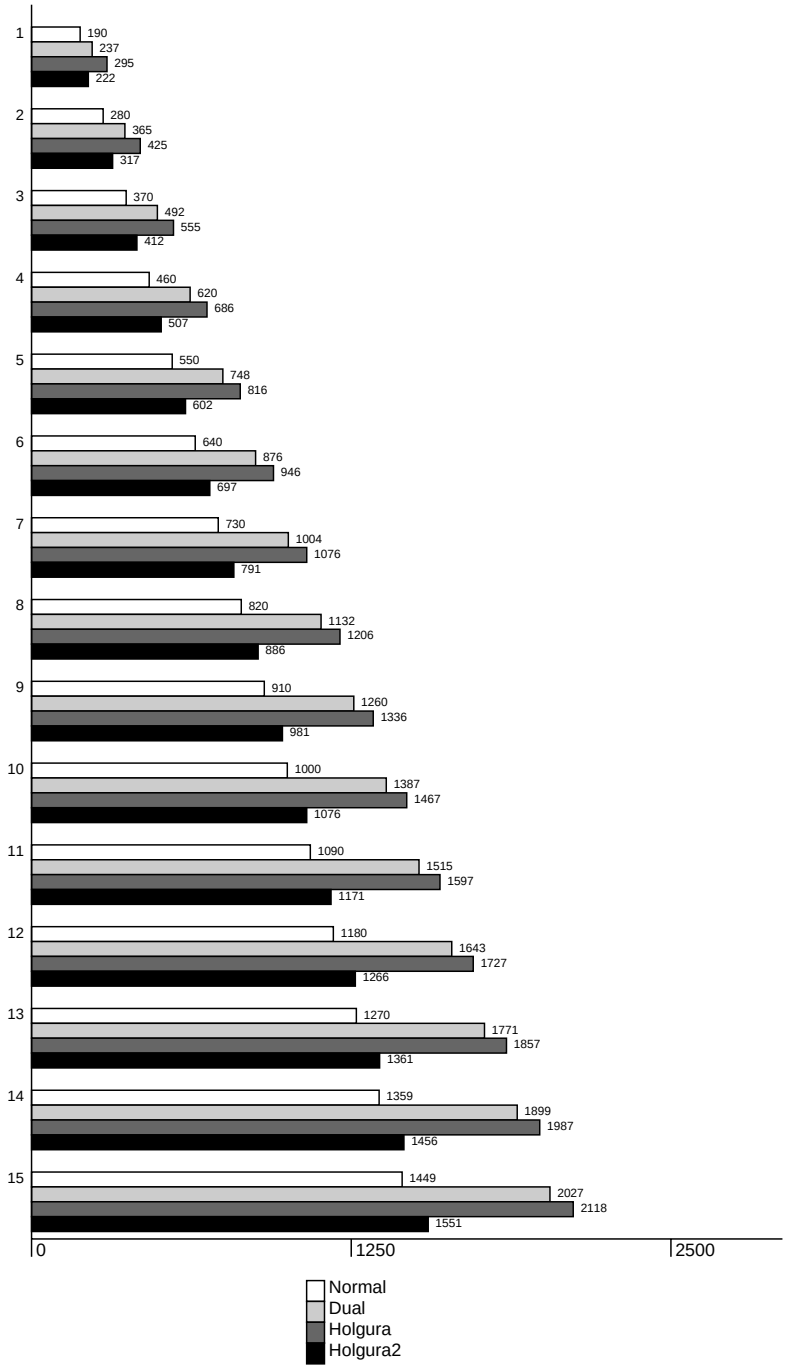


Figura 5.14: Diferencias en Tiempos de Respuesta. Caso Ada con Interrupciones Diferidas. (Valores en μs)

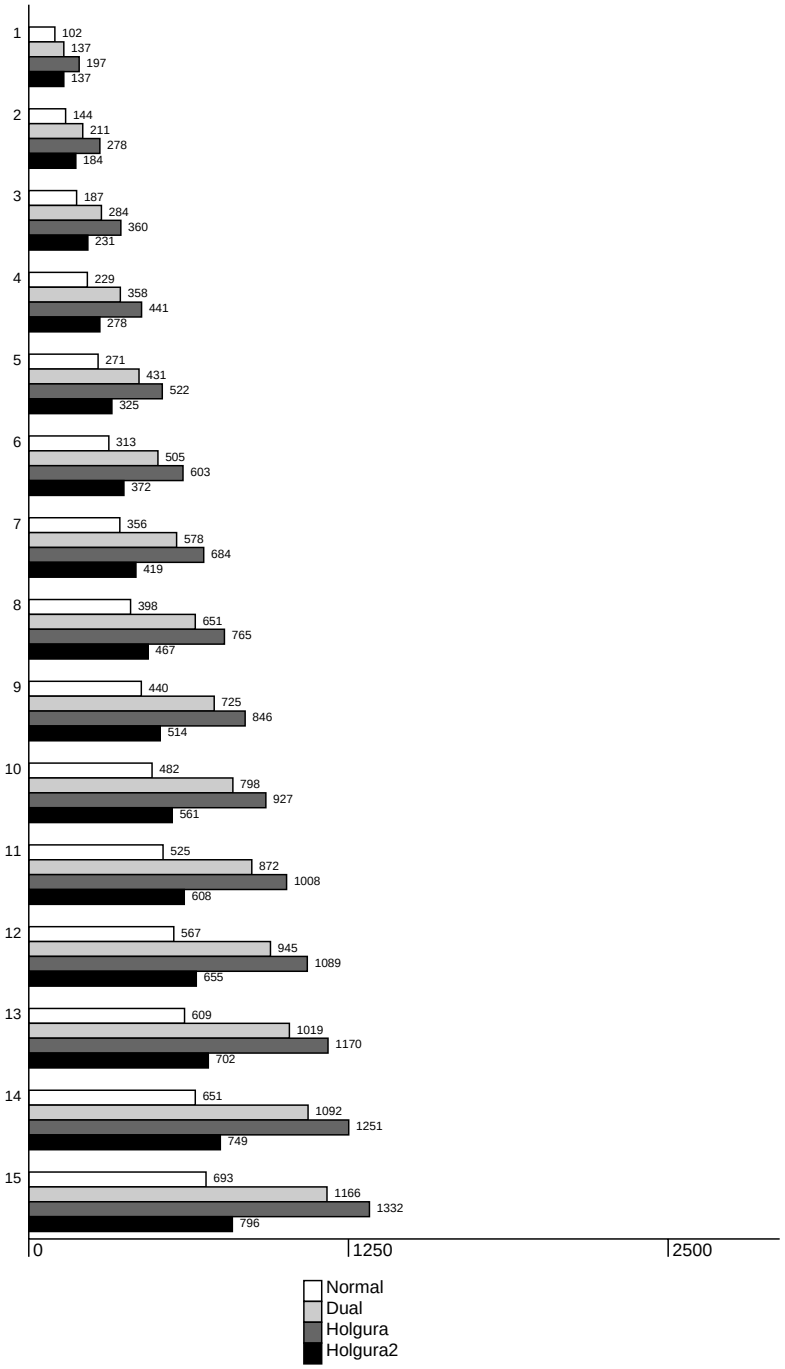


Figura 5.15: Diferencias en Tiempos de Respuesta. Caso POSIX con Interrupciones Diferidas. (Valores en μs)

5.6. Optimización del Algoritmo Extractor de Holgura

Analizando el comportamiento de las tareas cuando se utiliza el algoritmo extractor de holgura, se observa que es posible realizar una sencilla y eficaz optimización de este algoritmo. Esta optimización permite reducir el tiempo de respuesta en el peor caso de una tarea, especialmente si se utiliza conjuntamente con la técnica de las interrupciones diferidas descrita en la sección anterior. Es importante comentar que sólo es aplicable cuando se utiliza la política basada en la extracción de holgura tal fue definida en la sección 3.6, no cuando se utiliza la definición original de esta política. Esto es debido a que se requiere, por una parte, que la holgura se calcule cuando se activen las tareas, y por otra, que cuando venza la holgura de una tarea, ésta no se recupere hasta que la instancia actual de dicha tarea finalice.

Cuando vence la holgura para una tarea τ_i (instante t_1) sabemos que la holgura del sistema sólo podrá recuperarse cuando la instancia actual de τ_i finalice (instante t_2), momento en el que su holgura se hará igual a `Holgura_Infinita`. Ahora bien, si τ_i ha finalizado su instancia actual, también lo habrán hecho las instancias de las tareas más prioritarias que τ_i y su holgura será igual a `Holgura_Infinita`. De esta forma, no es necesario calcular la holgura de una tarea cuando la holgura en el sistema es igual a cero, lo que supone una importante reducción de la sobrecarga del sistema. De forma análoga, cuando se produce un cambio de contexto no es necesario actualizar la holgura de las tareas más prioritarias que la que abandona el procesador.

En t_2, la holgura del sistema dependerá exclusivamente de las holguras de las tareas menos prioritarias que τ_i. Si una de estas tareas ya estaba activa en t_1, su holgura será conocida puesto que no puede haber variado en el intervalo (t_1, t_2). Si una de estas tareas no estaba activa en t_1 y se activa antes de t_2, el procesamiento de las interrupciones diferidas hará que su holgura se calcule en t_2.

De esta forma, podemos concluir que, cuando se utiliza esta optimización, el valor asignado a la holgura del sistema no difiere del que se le habría asignado en caso de no utilizar esta optimización. El comportamiento de esta versión optimizada del algoritmo extractor de holgura aparece en las figuras 5.12, 5.13, 5.14 y 5.15 y en las tablas 5.4 y 5.5 con la etiqueta `holgura2`. Dado que esta optimización actúa cuando se produce un instante crítico para una tarea, se observa que los valores obtenidos son muy similares a los del sistema normal.

Estos resultados no indican que esta versión del algoritmo extractor de holgura sea más eficiente, puesto que el sistema normalmente no se encuentra en el instante crítico (o es posible que nunca se produzca), pero sí indica que cuando el sistema deba dejar de dar preferencia a tareas aperiódicas, volverá a poder atender tareas aperiódicas cuanto antes, puesto que todas las acciones específicas de la extracción de holgura son canceladas.

Si se utiliza la asignación dual de prioridades cabría plantearse una optimización similar. Por ejemplo, mientras una tarea τ_i ya está promocionada, las activaciones de tareas más prioritarias podrían no planificar la acción de promoción y asignar a la tarea su prioridad normal. Desafortunadamente, esto varía el comportamiento de las tareas, puesto que la instancia actual de τ_i puede finalizar antes de que se produzca la promoción de tareas más prioritarias. Dado que este comportamiento diferente no pone en riesgo el plazo de ninguna tarea, parece razonable aplicar esta optimización. A lo sumo puede empeorar levemente la capacidad del algoritmo de asignación dual de prioridades para ofrecer una mejor respuesta a tareas aperiódicas. Aunque no ha sido implementada, es de esperar unos resultados muy similares a los obtenidos con la versión optimizada del algoritmo extractor de holgura.

5.7. Comparación de Algoritmos Respecto a la Sobrecarga del Sistema

A diferencia del estudio de la planificabilidad, que se centra en el instante crítico, en esta sección se estudia la sobrecarga del sistema que generan los algoritmos de extracción de holgura y la asignación dual de prioridades durante la ejecución de una aplicación de tiempo real, la cual normalmente no se encuentra en un instante crítico. Para ello se ha diseñado el siguiente experimento.

El conjunto de tareas descrito en la tabla 5.1 se ha ejecutado en tres situaciones diferentes. En la primera no se ha introducido ninguna tarea aperiódica, en la segunda se introduce una tarea aperiódica con periodo medio igual a 10.000 μs y cómputo aproximado de 5.000 μs (utilización aperiódica del 50 %) y en la tercera se introduce una tarea aperiódica con periodo medio igual a 10.000 μs y cómputo aproximado de 10.000 μs (utilización aperiódica del 100 %).

Estos tres conjuntos de tareas se han ejecutado utilizando diferentes sistemas, combinando por un lado programación Ada y POSIX, por otro lado el sistema normal, la asignación dual de prioridades, la extracción de holgura y la versión optimizada de la extracción de holgura y finalmente la utilización o no utilización de interrupciones diferidas.

Para cada una de estas ejecuciones se ha medido en un hiperperiodo H (400.000 μs en nuestro caso) el tiempo a nivel de aplicación que utilizan las tareas críticas T_{CRI} (que es igual en todos los casos), el tiempo que el procesador está ocioso T_{OCI} y el tiempo a nivel de aplicación T_{APER} que la tarea aperiódica se ha ejecutado en dicho hiperperiodo.

Utilizando estos valores se ha determinado la utilización del sistema calculando el tiempo de ejecución de MaRTE OS y el soporte de GNAT como la diferencia existente entre el hiperperiodo y el trabajo a nivel de usuario u ocioso del procesador:

$$U_{SIS} = \frac{H - T_{CRI} - T_{OCI} - T_{APER}}{H} \tag{5.17}$$

Los valores obtenidos para la utilización del sistema se muestran en las figuras 5.16, 5.17, 5.18 y 5.19. Las mediciones realizadas para obtener estos valores se muestran la final de esta sección.

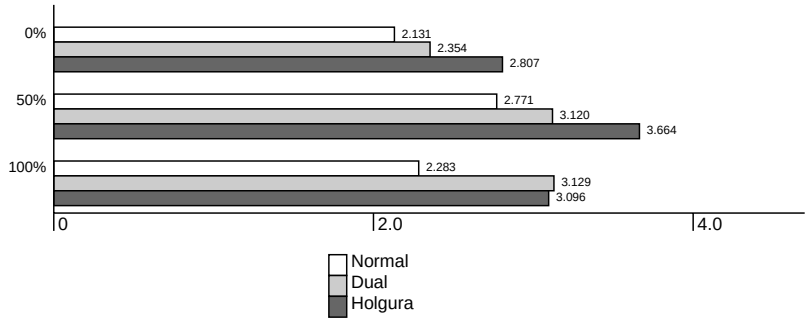


Figura 5.16: Utilización del Sistema (%). Caso Ada.

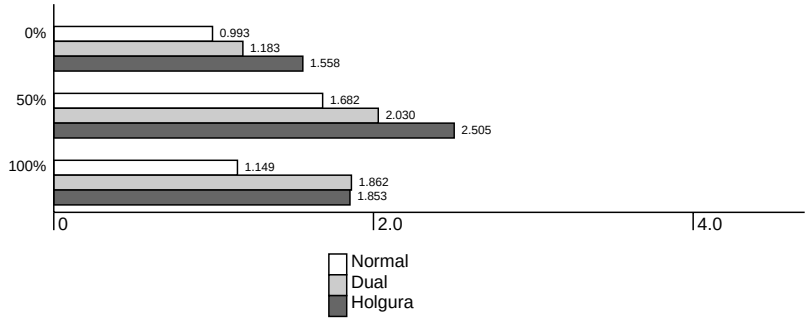


Figura 5.17: Utilización del Sistema (%). Caso POSIX.

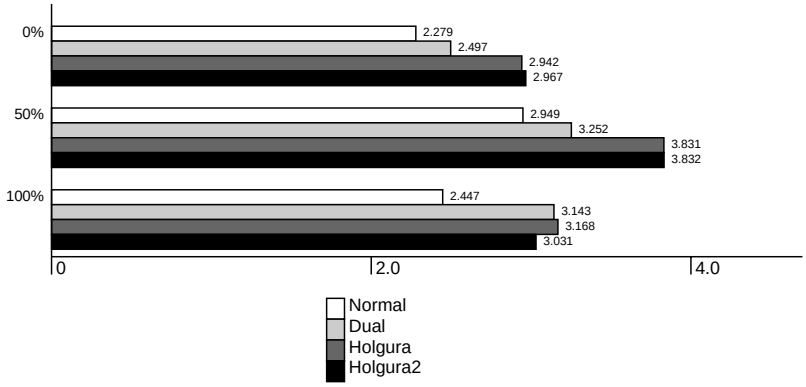


Figura 5.18: Utilización del Sistema (%). Caso Ada con Interrupciones Diferidas.

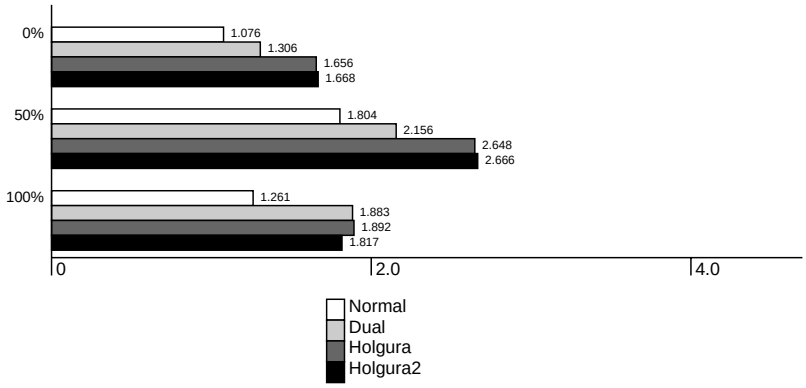


Figura 5.19: Utilización del Sistema (%). Caso POSIX con Interrupciones Diferidas.

Estas figuras muestran de nuevo las importantes diferencias que existen entre el uso de Ada y POSIX, llegando la sobrecarga introducida por Ada a duplicar la introducida por POSIX en algunos casos. En lo referente a la utilización de la técnica de interrupciones diferidas, su coste ha supuesto un incremento relativo de la sobrecarga de entre un 6 % y un 10 %.

Cuando la carga aperiódica es del 0 %, la asignación dual de prioridades planifica una acción de promoción cada vez que una tarea se activa, pero muchas de ellas no tienen lugar puesto que muchas de las tareas concluyen antes de alcanzar su promoción. En el caso de la extracción de holgura, al no haber carga aperiódica, no se planifica en ningún momento el evento temporal que indica que la holgura del sistema se agota. Se trata de una situación en la que ambos algoritmos ejecutan sólo parte de su actividad. Observando los resultados obtenidos se ve claramente que la sobrecarga que introduce la extracción de holgura es muy elevada en comparación con la introducida por la asignación dual de prioridades (aproximadamente un 20 % superior en el caso Ada y un 35 % en el caso POSIX).

En el caso en que la carga aperiódica es del 50 %, la actividad del sistema aumenta. Parte de este aumento es debido a la tarea aperiódica, la cual realiza llamadas al sistema y genera interrupciones del temporizador hardware. Este aumento puede observarse examinando los resultados que se obtienen con el sistema normal, siendo aproximadamente de un 30 % en el caso Ada y de un 65 % en el caso POSIX. La diferencia entre la asignación dual de prioridades y el sistema normal es mayor que en el caso de carga aperiódica del 0 %. Esto es debido a que se produce un mayor número de acciones de promoción. Esta diferencia también aumenta cuando se utiliza la extracción de holgura, en este caso debido a que se planifica el evento temporal de fin de holgura cada vez que la tarea aperiódica entra en ejecución y hay más cambios de contexto y, por tanto, más actualizaciones de la holgura de las tareas. De nuevo, la sobrecarga que introduce la extracción de holgura es muy elevada en comparación con la introducida por la asignación dual de prioridades (aproximadamente un 20 % superior en el caso Ada y un 30 % en el caso POSIX).

Es importante destacar que, en este caso, no se consume todo el tiempo de procesamiento disponible y, para todas las políticas, el tiempo de cómputo a nivel de usuario de tarea aperiódica es prácticamente idéntico, tal como se observa en las tablas 5.8, 5.11, 5.14 y 5.17.

Cuando la carga aperiódica es del 100 %, se observan unos resultados en principio algo inesperados, puesto que disminuye la utilización para el sistema normal y la extracción de holgura en mayor medida que para la asignación dual de prioridades. Veamos los motivos de este comportamiento. En este caso, la tarea aperiódica no provoca interrupciones del temporizador hardware, puesto que cuando solicita esperar a su próxima activación, ésta ya está en el pasado, y por tanto nunca se suspende. De ahí la disminución de la

utilización del sistema que se observa para la política estándar. Respecto al caso de carga aperiódica del 0 %, la sobrecarga del sistema normal aumenta tan sólo un 7 % en el caso Ada y un 15 % en el caso POSIX. Esta disminución también se produce para la asignación dual de prioridades y la extracción de holgura, aunque queda enmascarada por otras acciones realizadas por estos algoritmos. Como consecuencia de que la tarea aperiódica siempre está preparada, la asignación dual de prioridades se encuentra en la peor situación posible, puesto que todas las acciones de promoción planificadas se llevan a cabo, de ahí el considerable aumento en la utilización del sistema observado respecto a la política estándar (entre el 30 % y el 60 %). La extracción de holgura, por contra, se beneficia de que la tarea aperiódica esté siempre preparada, ya que se producen menos cambios de contexto y, por tanto, menos actualizaciones de la holgura de las tareas y menos acciones de planificación del evento temporal de fin de holgura. De hecho, los resultados que se obtienen son similares e incluso inferiores que los obtenidos por la asignación dual de prioridades.

Este comportamiento observado plantea una interesante duda en lo referente a la eficiencia relativa de la extracción de holgura y la asignación dual de prioridades. La asignación de holgura es más ineficiente cuando la carga aperiódica no consume todo el tiempo de procesamiento disponible, pero en este caso podríamos decir que la eficiencia es un criterio de escaso interés puesto que ambos algoritmos son capaces de ejecutar toda la carga aperiódica que se solicita. En cambio, cuando la carga aperiódica excede el tiempo de procesamiento disponible ambos algoritmos se comportan de forma muy similar. Por supuesto, extraer este tipo de conclusiones es, cuanto menos, aventurado a partir de un único ejemplo, y sería necesaria una experimentación más elaborada para poder dar una respuesta más concluyente ante esta cuestión. En cualquier caso, se han realizado varios experimentos con otros conjuntos de tareas y los resultados han sido bastante similares a los descritos.

Es importante también comentar el comportamiento de la versión optimizada de la extracción de holgura, la cual ofrece los mejores resultados cuando la carga aperiódica es del 100 %, aunque la mejora es bastante escasa. No obstante, hay que tener en cuenta que esta optimización se hace más patente cuando en el sistema hay tareas cuyo tiempo de cómputo es notablemente superior al de los periodos de otras tareas más prioritarias, lo que no sucede en nuestro conjunto de tareas de ejemplo. De nuevo, en este caso, una experimentación más compleja que la realizada en este apartado se hace necesaria para caracterizar mejor esta versión optimizada de la extracción de holgura.

Término	Descripción
H	Hiperperiodo. Común para todos los casos.
T_{CRI}	Tiempo de ejecución de carga crítica. Común para todos los casos.
T_{OCI}	Tiempo ocioso.
T_{APER}	Tiempo de ejecución de carga crítica.
T_{SIS}	Tiempo de ejecución del sistema. Igual a $H - T_{CRI} - T_{OCI} - T_{APER}$
U_{CRI}	Porcentaje de carga crítica. Igual a (T_{CRI}/H) * 100
U_{OCI}	Porcentaje de tiempo ocioso. Igual a (T_{OCI}/H) * 100
U_{APER}	Porcentaje de carga crítica. Igual a (T_{APER}/H) * 100
U_{SIS}	Porcentaje de sistema. Igual a (T_{SIS}/H) * 100

Tabla 5.6: Descripción de los Términos Utilizados

Mediciones

5.8. Secciones de Cómputo Opcional

En esta sección se analiza la sobrecarga que introduce la utilización de secciones de cómputo opcional, definidas en la sección 3.9. Uno de los posibles usos de este servicio, descrito en dicha sección, consiste en programar tareas críticas que ejecutan una parte opcional en su contexto, tal como se ilustra en el siguiente fragmento de código:

```
task body T1 is
begin
   loop
      Initial_Part;
      select
         Ada.Scheduling.Scheduler.Wait_Slack_Exhausted;
      then abort
         Optional_Part;
      end select;
      Final_Part;

      Next_Activation := Next_Activation + Period;
      To_Wait_Activation;
      delay until Next_Activation;
   end loop;
end T1;
```

Dada la definición de la operación `Wait_Slack_Exhausted`, la parte opcional de la tarea puede ser abortada para garantizar que la tarea ejecuta su parte final sin perder su plazo.

El uso de partes opcionales requiere utilizar la sentencia de transferencia asíncrona de control. Las acciones asociadas a esta sentencia son ejecutadas por el soporte de GNAT,

	Normal	Dual	Holgura
H	400000.000	400000.000	400000.000
T_{CRI}	63640.300	63640.300	63640.300
T_{OCI}	327835.537	326944.848	325131.400
T_{APER}	0.000	0.000	0.000
T_{SIS}	8524.163	9414.852	11228.300
U_{CRI}	15.910	15.910	15.910
U_{OCI}	81.959	81.736	81.283
U_{APER}	0.000	0.000	0.000
U_{SIS}	2.131	2.354	2.807

Tabla 5.7: Mediciones (μs). Carga Aperiódica del 0 %. Caso Ada.

	Normal	Dual	Holgura
H	400000.000	400000.000	400000.000
T_{CRI}	63640.300	63640.300	63640.300
T_{OCI}	117635.768	116192.361	113999.967
T_{APER}	207641.804	207687.167	207703.324
T_{SIS}	11082.128	12480.172	14656.409
U_{CRI}	15.910	15.910	15.910
U_{OCI}	29.409	29.048	28.500
U_{APER}	51.910	51.922	51.926
U_{SIS}	2.771	3.120	3.664

Tabla 5.8: Mediciones (μs). Carga Aperiódica del 50 %. Caso Ada.

	Normal	Dual	Holgura
H	400000.000	400000.000	400000.000
T_{CRI}	63640.300	63640.300	63640.300
T_{OCI}	0.000	0.000	0.000
T_{APER}	327226.149	323843.188	323974.273
T_{SIS}	9133.551	12516.512	12385.427
U_{CRI}	15.910	15.910	15.910
U_{OCI}	0.000	0.000	0.000
U_{APER}	81.807	80.961	80.994
U_{SIS}	2.283	3.129	3.096

Tabla 5.9: Mediciones (μs). Carga Aperiódica del 100 %. Caso Ada.

	Normal	Dual	Holgura
H	400000.000	400000.000	400000.000
T_{CRI}	63640.300	63640.300	63640.300
T_{OCI}	332386.771	331626.448	330125.732
T_{APER}	0.000	0.000	0.000
T_{SIS}	3972.929	4733.252	6233.968
U_{CRI}	15.910	15.910	15.910
U_{OCI}	83.097	82.907	82.531
U_{APER}	0.000	0.000	0.000
U_{SIS}	0.993	1.183	1.558

Tabla 5.10: Mediciones (μs). Carga Aperiódica del 0 %. Caso Posix.

	Normal	Dual	Holgura
H	400000.000	400000.000	400000.000
T_{CRI}	63640.300	63640.300	63640.300
T_{OCI}	122015.807	120575.690	118659.096
T_{APER}	207616.601	207662.845	207680.174
T_{SIS}	6727.292	8121.165	10020.430
U_{CRI}	15.910	15.910	15.910
U_{OCI}	30.504	30.144	29.665
U_{APER}	51.904	51.916	51.920
U_{SIS}	1.682	2.030	2.505

Tabla 5.11: Mediciones (μs). Carga Aperiódica del 50 %. Caso Posix.

	Normal	Dual	Holgura
H	400000.000	400000.000	400000.000
T_{CRI}	63640.300	63640.300	63640.300
T_{OCI}	0.000	0.000	0.000
T_{APER}	331762.884	328913.445	328948.099
T_{SIS}	4596.816	7446.255	7411.601
U_{CRI}	15.910	15.910	15.910
U_{OCI}	0.000	0.000	0.000
U_{APER}	82.941	82.228	82.237
U_{SIS}	1.149	1.862	1.853

Tabla 5.12: Mediciones (μs). Carga Aperiódica del 100 %. Caso Posix.

	Normal	Dual	Holgura	Holgura2
H	400000.000	400000.000	400000.000	400000.000
T_{CRI}	63640.300	63640.300	63640.300	63640.300
T_{OCI}	327244.545	326372.139	324589.911	324490.478
T_{APER}	0.000	0.000	0.000	0.000
T_{SIS}	9115.155	9987.561	11769.789	11869.222
U_{CRI}	15.910	15.910	15.910	15.910
U_{OCI}	81.811	81.593	81.147	81.123
U_{APER}	0.000	0.000	0.000	0.000
U_{SIS}	2.279	2.497	2.942	2.967

Tabla 5.13: Mediciones (μs). Carga Aperiódica del 0 %. Caso Ada con Interrupciones Diferidas.

	Normal	Dual	Holgura	Holgura2
H	400000.000	400000.000	400000.000	400000.000
T_{CRI}	63640.300	63640.300	63640.300	63640.300
T_{OCI}	116913.836	115665.348	113330.254	113325.097
T_{APER}	207648.735	207685.793	207707.023	207706.211
T_{SIS}	11797.129	13008.559	15322.423	15328.392
U_{CRI}	15.910	15.910	15.910	15.910
U_{OCI}	29.228	28.916	28.333	28.331
U_{APER}	51.912	51.921	51.927	51.927
U_{SIS}	2.949	3.252	3.831	3.832

Tabla 5.14: Mediciones (μs). Carga Aperiódica del 50 %. Caso Ada con Interrupciones Diferidas.

	Normal	Dual	Holgura	Holgura2
H	400000.000	400000.000	400000.000	400000.000
T_{CRI}	63640.300	63640.300	63640.300	63640.300
T_{OCI}	0.000	0.000	0.000	0.000
T_{APER}	326572.838	323788.071	323686.329	324234.168
T_{SIS}	9786.862	12571.629	12673.371	12125.532
U_{CRI}	15.910	15.910	15.910	15.910
U_{OCI}	0.000	0.000	0.000	0.000
U_{APER}	81.643	80.947	80.922	81.059
U_{SIS}	2.447	3.143	3.168	3.031

Tabla 5.15: Mediciones (μs). Carga Aperiódica del 100 %. Caso Ada con Interrupciones Diferidas.

	Normal	Dual	Holgura	Holgura2
H	400000.000	400000.000	400000.000	400000.000
T_{CRI}	63640.300	63640.300	63640.300	63640.300
T_{OCI}	332056.014	331135.276	329736.448	329688.558
T_{APER}	0.000	0.000	0.000	0.000
T_{SIS}	4303.686	5224.424	6623.252	6671.142
U_{CRI}	15.910	15.910	15.910	15.910
U_{OCI}	83.014	82.784	82.434	82.422
U_{APER}	0.000	0.000	0.000	0.000
U_{SIS}	1.076	1.306	1.656	1.668

Tabla 5.16: Mediciones (μs). Carga Aperiódica del 0 %. Caso POSIX con Interrupciones Diferidas.

	Normal	Dual	Holgura	Holgura2
H	400000.000	400000.000	400000.000	400000.000
T_{CRI}	63640.300	63640.300	63640.300	63640.300
T_{OCI}	121509.034	120073.000	118086.570	118016.631
T_{APER}	207633.653	207661.524	207679.977	207680.773
T_{SIS}	7217.013	8625.176	10593.153	10662.296
U_{CRI}	15.910	15.910	15.910	15.910
U_{OCI}	30.377	30.018	29.522	29.504
U_{APER}	51.908	51.915	51.920	51.920
U_{SIS}	1.804	2.156	2.648	2.666

Tabla 5.17: Mediciones (μs). Carga Aperiódica del 50 %. Caso POSIX con Interrupciones Diferidas.

	Normal	Dual	Holgura	Holgura2
H	400000.000	400000.000	400000.000	400000.000
T_{CRI}	63640.300	63640.300	63640.300	63640.300
T_{OCI}	0.000	0.000	0.000	0.000
T_{APER}	331314.385	328826.046	328791.691	329090.445
T_{SIS}	5045.315	7533.654	7568.009	7269.255
U_{CRI}	15.910	15.910	15.910	15.910
U_{OCI}	0.000	0.000	0.000	0.000
U_{APER}	82.829	82.207	82.198	82.273
U_{SIS}	1.261	1.883	1.892	1.817

Tabla 5.18: Mediciones (μs). Carga Aperiódica del 100 %. Caso POSIX con Interrupciones Diferidas.

lo que incrementa el tiempo de cómputo a nivel de sistema de la tarea. En [45] puede consultarse una descripción detallada de las acciones que realiza este soporte. Veamos a continuación las acciones que realiza el sistema cuando una tarea ejecuta una parte opcional.

La primera acción que realiza la tarea consiste en notificar al soporte de GNAT que inicia una sentencia de transferencia asíncrona de control, asociada a la aceptación de la entrada `Wait_Slack_Exhausted`. Esta acción se representa en la figura 5.20. En t_0 se produce la llamada al soporte de GNAT, llamada que termina en t_1, instante en que comienza la ejecución de la parte opcional. En este intervalo el soporte de GNAT ejecuta una sección crítica con el fin de preservar la integridad de las estructuras de datos internas de este soporte. Esta sección crítica (al igual que el resto de secciones críticas que se muestran en este apartado) está guardada por un mutex cuyo techo de prioridad es igual a la prioridad más alta del sistema.

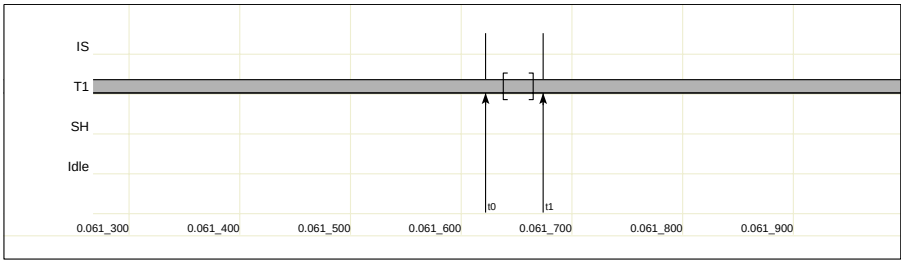


Figura 5.20: La Parte Opcional es Activada. (Valores en μs)

Si la parte opcional termina antes de que se acepte la entrada `Wait_Slack_Exhausted`, la tarea notifica al soporte de GNAT que ha terminado la sentencia de transferencia asíncrona de control, acción que se muestra en la figura 5.21 en el intervalo $[t_0, t_1]$. Tras el instante t_1 comienza la ejecución de la parte final de la tarea.

Por el contrario, si la entrada `Wait_Slack_Exhausted` es aceptada, la parte opcional es abortada, tal como se ilustra en la figura 5.22. En este caso, las acciones que se realizan son bastante complejas.

En t_0 interrumpe el temporizador hardware para notificar al sistema que vence la holgura o que la tarea se promociona, dependiendo de la política que se esté utilizando. En este momento MaRTE OS genera la señal `SIGSLACK`, acción que despierta en t_1 a una tarea interna del soporte de GNAT denominada Servidor de Interrupciones (IS). Esta tarea es la encargada de ejecutar las acciones que la aplicación ha asociado a interrupciones (señales en el caso de GNAT).

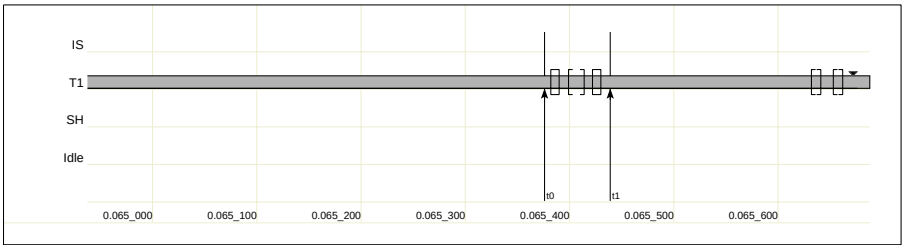


Figura 5.21: La Parte Opcional Termina. (Valores en μs)

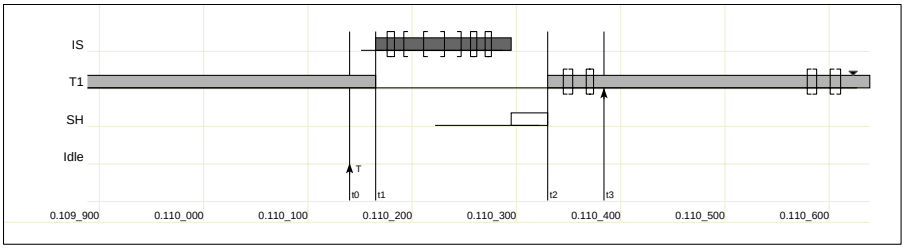


Figura 5.22: La Parte Opcional es Abortada. (Valores en μs)

En nuestro caso, IS modifica el estado del objeto `Scheduler` para que se acepte la entrada `Wait_Slack_Exhausted`. Esta aceptación implica abortar la ejecución de la tarea $T1$, para lo cual IS envía la señal `SIGABRT` a la tarea $T1$. Esta señal está capturada por la tarea $T1$ y su manejador es ejecutado por la tarea interna de MaRTE OS Gestor de Señales (SH). Cuando la tarea $T1$ vuelve a ejecución en t_2, aborta la ejecución de su código y termina la ejecución de la sentencia de transferencia asíncrona de control. En t_3 comienza la ejecución de la parte final de $T1$.

Es importante destacar que las tareas IS y SH utilizan la prioridad más alta del sistema, por lo que el intervalo $[t_0, t_1]$ debería considerarse como una sección crítica cuyo techo es igual a la máxima prioridad, por lo que es una causa de bloqueo para todas las tareas de la aplicación.

Los tiempos de ejecución de las acciones descritas anteriormente, para las dos políticas objeto de estudio, se muestran en la tabla 5.19. La sobrecarga que ocasiona la utilización de partes opcionales es igual a la suma de las acciones de activar y abortar una parte opcional, ya que la duración de esta última acción es mayor que la de la acción de terminar una parte opcional. Esta sobrecarga forma parte del tiempo de cómputo en el peor caso C_i de la tarea. El mayor bloqueo detectado corresponde con la ejecución conjunta de las tareas IS y SH (figura 5.22), y por tanto, debe ser tenido en cuenta para el cálculo del factor de bloqueo B_i.

Acción	Dual	Holgura
Activar parte opcional	52	53
Terminar parte opcional	63	64
Abortar parte opcional	244	252
Sobrecarga parte opcional	296	305
Bloqueo Mayor	165	168

Tabla 5.19: Sobrecarga Partes Opcionales. (Valores en μs)

En la sección 3.9 se describe otra forma de programar tareas con partes opcionales. En este caso, la parte opcional se ejecuta en una tarea distinta, tal como se muestra en el siguiente fragmento de código:

```
task body T1 is
begin
 loop
   Initial_Part;
   select
      Ada.Scheduling.Scheduler.Wait_Slack_Exhausted;
      Sync.Stop;
   then abort
      Sync.Start;
      Sync.Wait_Terminated;
   end select;
   Final_Part;
 end loop;

   Next_Activation := Next_Activation + Period;
   To_Wait_Activation;
   delay until Next_Activation;
end T1;


task body OT1 is
begin
 loop
   Sync.Wait_Started;
   select
      Sync.Wait_Stopped;
   then abort
      Optional_Part;
      Sync.Terminated;
   end select;
 end loop;
end OT1;
```

La tarea T1 solicita la ejecución de su parte opcional activando a la tarea OT1 y a continuación, espera a que ésta notifique que la ejecución de la parte opcional ha finalizado. Esta espera puede ser cancelada si la entrada `Wait_Slack_Exhausted` es aceptada. En este caso, la tarea T1 notifica a la tarea OT1 que la ejecución de la parte opcional debe ser abortada. La sincronización entre ambas tareas es llevada a cabo por el objeto protegido `Sync`.

Al igual que en el caso anterior, el soporte de GNAT interviene en la gestión de la ejecución de la parte opcional, aunque de forma más compleja dada la sincronización

necesaria entre las tareas T1 y OT1. Veamos a continuación las acciones que se ejecutan en el sistema.

La activación de la tarea OT1 se muestra en la figura 5.23. La tarea T1 notifica al soporte de GNAT en t_0 que inicia la ejecución de una sentencia de transferencia asíncrona de control. A continuación, modifica el estado del objeto Sync, lo que causa que se despierte la tarea OT1. Finalmente, la tarea T1, en el instante t_1, se suspende en el objeto protegido Sync. La tarea OT1 comienza la ejecución de la parte opcional en t_2.

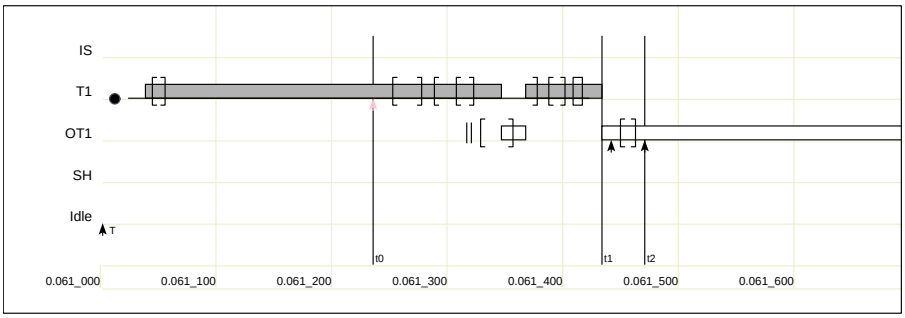


Figura 5.23: La Tarea Opcional es Activada. (Valores en μs)

La figura 5.24 muestra las acciones que se producen cuando la tarea OT1 termina la parte opcional sin ser abortada. En t_0 la tarea OT1 notifica su terminación al objeto protegido Sync, lo que produce que se despierte la tarea T1. A continuación, la tarea T1 notifica al soporte de GNAT que ha terminado su sentencia de transferencia asíncrona de control. En t_1 comienza la ejecución de la parte final de T1.

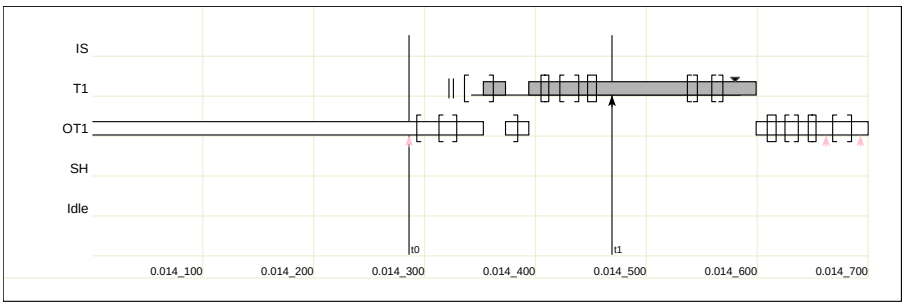


Figura 5.24: La Tarea Opcional Termina. (Valores en μs)

Las acciones que se ejecutan cuando la entrada `Wait_Slack_Exhausted` es aceptada se muestran en la figura 5.25.

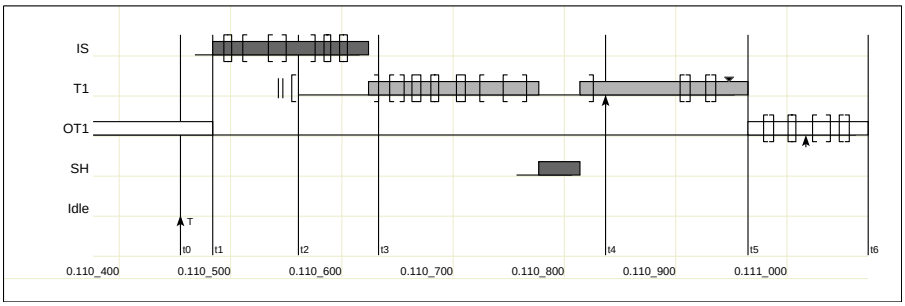


Figura 5.25: La Tarea Opcional es Abortada. (Valores en μs)

En t_0 interrumpe el temporizador hardware para notificar al sistema que vence la holgura o que la tarea se promociona, dependiendo de la política que se esté utilizando. La tarea interna IS se activa en t_1 como consecuencia de la señal `SIGSLACK`, al igual que sucede cuando la parte opcional es ejecutada en el contexto de la tarea crítica. En este caso se aborta la espera de la tarea T1 y ésta se activa en t_2. A continuación, T1 notifica al objeto protegido `Sync` que debe ser abortada la parte opcional, lo que origina el envío de la señal `SIGABRT` a la tarea OT1. El manejador de esta señal es ejecutado por la tarea interna de MaRTE OS IS. En t_4 comienza la parte final de la tarea T1. La tarea OT1 finaliza su sentencia de transferencia asíncrona de control una vez T1 termina la ejecución de su instancia actual en t_5 y queda a la espera de ser activada de nuevo en t_6.

La duración de las acciones descritas se muestra en la tabla 5.20. En este caso la sección crítica de mayor duración se produce cuando se aborta la espera de la tarea T1 (intervalo $[t_1, t_2]$ de la figura 5.25).

Acción	Dual	Holgura
Activar parte opcional	199	207
Terminar parte opcional	183	188
Abortar parte opcional	382	389
Sobrecarga parte opcional	581	596
Bloqueo Mayor	149	151

Tabla 5.20: Sobrecarga Partes Opcionales en Tareas. (Valores en μs)

5.9. Resumen y Conclusiones

En este capítulo se ha estudiado el coste de introducir en el sistema las políticas de planificación basadas en la extracción de holgura y en la asignación dual de prioridades. Este estudio se ha realizado utilizando la implementación de ambas políticas desarrollada en MaRTE OS y los servicios de trazas y obtención de métricas descritos en el capítulo 4.

En primer lugar, en la sección 5.1, se ha analizado el comportamiento del sistema MaRTE OS, cuando se aplica la política de planificación estándar basada en prioridades fijas con desalojo. A raíz de este análisis, se ha enunciado la ecuación que permite calcular el tiempo de respuesta de una tarea crítica teniendo en cuenta la sobrecarga que introduce el sistema.

Partiendo del estudio anterior, se ha analizado cómo afecta al comportamiento del sistema la aplicación de las políticas basadas en la extracción de holgura y en la asignación dual de prioridades, obteniendo las ecuaciones que permiten calcular el tiempo de respuesta cuando se utilizan estas políticas (secciones 5.2 y 5.3).

Estas ecuaciones se han aplicado a un caso de estudio en la sección 5.4, con el fin de valorar cómo estas políticas afectan a los tiempos de respuesta de las tareas críticas. Para ello, se ha programado un conjunto de tareas utilizando, por una parte, los servicios del lenguaje Ada y, por otra parte, los servicios POSIX. Este conjunto de tareas se ha ejecutado en MaRTE OS aplicando las políticas de planificación objeto de estudio: prioridades fijas con desalojo, extracción de holgura y asignación dual de prioridades. A partir de estas ejecuciones, se han obtenido mediante mediciones los valores necesarios para determinar los términos de las ecuaciones de cálculo de los tiempos de respuesta. Finalmente, se han calculado los tiempos de respuesta para cada tarea del caso de estudio, y se han comparado los resultados obtenidos junto con los tiempos de respuesta que se obtendrían en un sistema ideal, en el cual el sistema operativo no introduce sobrecarga alguna.

Las principales conclusiones que se han obtenido a partir de este estudio son las siguientes:

- Las diferencias observadas entre las programaciones basadas en Ada y POSIX son muy acusadas. El tiempo de ejecución de los servicios Ada duplica el tiempo de ejecución de los servicios POSIX equivalentes. Queda patente por tanto el hecho ya conocido de que la sobrecarga que introduce el soporte en tiempo de ejecución del compilador GNAT es muy significativa.

- Los tiempos de ejecución requeridos por la política basada en la extracción de holgura son bastante superiores a los de la política basada en la asignación dual de prioridades, mientras que esta última política se encuentra bastante próxima a la política basada en prioridades fijas con desalojo. Es en el servicio de activación de

una tarea cuando las diferencias entre la extracción de holgura y el resto de políticas estudiadas es más acusado. Esta diferencia es debida a que es en la activación de una tarea cuando se realiza el cálculo de la holgura. En cualquier caso, se esperaban unas diferencias bastante más acusadas.

- Respecto a los tiempos de respuesta obtenidos, cabe destacar que la política de asignación dual de prioridades ofrece un comportamiento relativo peor a medida que disminuye la prioridad de las tareas. Este efecto es debido a dos factores: la interferencia que una tarea causa a tareas menos prioritarias es similar a la de la extracción de holgura y, en las tareas menos prioritarias, se reduce el efecto causado por las activaciones de otras tareas.

- La activación de tareas de menor prioridad influye de forma muy negativa en los tiempos de respuesta que se obtienen en la política basada en la extracción de holgura, ya que la activación de una tarea incluye el cálculo de su holgura.

- La variabilidad en el instante de activación causada por las secciones en las cuales el sistema MaRTE OS se ejecuta con interrupciones inhibidas, también ocasiona un efecto muy negativo cuando se aplica la política basada en la extracción de holgura.

- La activación de tareas de menor prioridad y la variabilidad en el instante de activación son factores relacionados entre sí, por lo que considerarlos por separado introduce un pesimismo innecesario a la hora de calcular los tiempos de respuesta de las tareas. Desafortunadamente, no parece posible, al menos de forma sencilla, formular una ecuación de cálculo de tiempos de respuesta que refleje la relación que existe entre estos factores.

Dado que el tiempo de respuesta de las tareas se ve muy afectado por la activación de tareas de menor prioridad y por la variabilidad en la activación causada por las secciones en las que el sistema se ejecuta con interrupciones inhibidas, se ha propuesto modificar el sistema MaRTE OS para gestionar de forma diferida la interrupciones del temporizador hardware, ya que esta gestión puede reducir de forma considerable el efecto de estos factores.

El comportamiento de esta versión modificada de MaRTE OS ha sido analizado y se ha modificado la ecuación del cálculo de tiempos de respuesta. La sobrecarga del sistema ha sido medida y se han calculado los tiempos de respuesta para el conjunto de tareas del caso de estudio.

Al aplicar esta gestión se observa que los tiempos de respuesta de las tareas se reducen de forma notable para las tres políticas estudiadas, especialmente para las tareas más prioritarias. También se observa que la política basada en la extracción de holgura se beneficia notablemente de la aplicación de esta técnica, ya que los resultados que se obtienen para

esta política se aproximan mucho más a los resultados obtenidos con la política basada en la asignación dual de prioridades.

En este capítulo se ha propuesto también una optimización sencilla y efectiva de la política basada en la extracción de holgura, la cual consiste en cancelar las operaciones de cálculo de la holgura y su actualización cuando la holgura del sistema es igual a cero. Esta optimización permite que se obtengan unos tiempos de respuesta muy similares a los obtenidos con la política basada en prioridades fijas con desalojo. La importancia de esta optimización reside en que, cuando se agota la holgura del sistema y, en consecuencia, las tareas aperiódicas dejan de tener un trato preferente, el sistema vuelve lo antes posible a disponer de holgura para poder seguir atendiendo con preferencia las tareas aperiódicas.

La eficiencia de las políticas objeto de estudio se ha analizado también respecto a la sobrecarga que introducen en el sistema en un intervalo de funcionamiento normal, no en un instante crítico. En este caso, el conjunto de tareas utilizado en el estudio anterior ha sido ejecutado durante un hiperperiodo, y se ha medido el tiempo que el procesador dedica a ejecutar tareas aperiódicas y a estar ocioso. A partir de los datos obtenidos, se ha determinado la utilización que el sistema hace del procesador. Las principales conclusiones obtenidas a partir de este estudio han sido las siguientes:

- La gestión diferida de las interrupciones introduce una sobrecarga poco significativa, por lo que podría decirse que su aplicación es beneficiosa, puesto que contribuye notablemente a reducir los tiempos de respuesta de las tareas.

- Cuando la carga aperiódica es elevada, la extracción de holgura introduce una sobrecarga similar a la asignación dual de prioridades. Este hecho plantea una interesante duda respecto a la eficiencia relativa de ambas políticas objeto de estudio, duda que debería resolverse realizando un estudio más completo, ya que son muchos los factores que pueden influir en el comportamiento de estas políticas. En cualquier caso, se han realizado varios experimentos con otros conjuntos de tareas y los resultados han sido bastante similares a los descritos.

Finalmente, se ha evaluado la eficiencia de la utilización de las secciones de cómputo opcional en el lenguaje Ada. La sobrecarga que introducen se ha medido para dos casos diferentes: cuando la parte opcional se ejecuta en el contexto de una tarea crítica y cuando se ejecuta en el contexto de una tarea aperiódica. En ambos casos se observa que los tiempos que se obtienen son muy elevados, con independencia de la política que se utilice. La causa más destacable de los resultados obtenidos es la complejidad de la implementación de la sentencia de transferencia asíncrona de control. Dado este inconveniente, el uso de las secciones de cómputo adicional quedaría limitado a sistemas en los que la sobrecarga que introducen sea aceptable. Tal como se comentó en la sección 3.10, las dificultades

encontradas han impedido realizar esta evaluación para la interfaz POSIX. En cualquier caso, abortar una sección de código en POSIX requiere de una programación relativamente compleja, por lo que se estima que los tiempos que se hubiesen obtenido también serían bastante elevados.

Capítulo 6

Conclusiones y Trabajos Futuros

6.1. Conclusiones

En el capítulo 1 se planteó como objetivo de esta tesis el estudio de las políticas de planificación basadas en la extracción de holgura y en la asignación dual de prioridades desde una nueva perspectiva, su aplicabilidad en la práctica. Entendemos como aplicabilidad la posibilidad de utilizar estas políticas en el contexto de los dos entornos más importantes que permiten la programación de sistemas de tiempo real, los servicios POSIX y el lenguaje de programación Ada.

Este estudio ha supuesto la revisión de estas políticas de planificación, su definición en el lenguaje Ada y en los servicios POSIX, su implementación en el sistema operativo MaRTE OS y el análisis de la eficiencia de esta implementación. Adicionalmente, se ha abordado el tema de los servicios de trazas POSIX, una de las incorporaciones más recientes a este estándar. En esta sección se revisan los aspectos más relevantes del trabajo realizado.

Revisión de las Políticas de Planificación basadas en la Extracción de Holgura y en la Asignación Dual de Prioridades

Ambas políticas tienen en esencia el mismo objetivo, ofrecer un trato preferente a las tareas aperiódicas. Su funcionamiento es similar, puesto que en ambos casos las tareas críticas pueden retrasar su ejecución en función de la holgura disponible en el sistema. Difieren en la forma en la que ambas políticas detectan esta holgura. En la extracción de holgura esta detección se realiza de forma directa, calculando la holgura en función del estado actual del sistema. La asignación dual de prioridades detecta esta holgura de forma indirecta, variando la prioridad de las tareas críticas. Este cambio de prioridad se realiza

en los instantes de promoción, los cuales son función del tiempo de respuesta en el peor caso y del plazo de cada tarea, valores que son calculados a priori.

La política basada en la asignación dual de prioridades es mucho más sencilla y eficiente. Es aplicable en un abanico muy amplio de situaciones, como por ejemplo la existencia de recursos compartidos entre tareas críticas y aperiódicas, la presencia de tareas esporádicas en el sistema y la posibilidad de utilizar plazos superiores al periodo. La política basada en la extracción de holgura es más compleja e ineficiente, y admite de forma bastante limitada las situaciones citadas anteriormente. En cambio, tiene la propiedad de detectar más holgura, lo que permite que las tareas aperiódicas obtengan mejores tiempos de respuesta promedios.

La política basada en la extracción de holgura entendemos que ha sido mejorada en dos aspectos. El más relevante ha sido la modificación del método de asignación de prioridades, modificación que ha permitido que las tareas aperiódicas y críticas puedan compartir recursos entre sí usando los mismos protocolos utilizados por las tareas críticas. Otro aspecto mejorado, ha sido la aplicabilidad de esta política en presencia de tareas esporádicas. A raíz de la modificación propuesta, la presencia de tareas esporádicas no requiere ahora de ningún tratamiento particular. Ambas modificaciones han mejorado el ámbito de aplicabilidad de esta política y han reducido su complejidad.

Diseño de Nuevas Interfaces Ada y POSIX

Estas interfaces han sido diseñadas teniendo como principales objetivos los siguientes: sencillez de uso, eficiencia, compatibilidad con el resto de servicios y coherencia entre las interfaces Ada y POSIX.

Para garantizar la consecución de estos objetivos, el diseño de estas interfaces ha sido realizado conjuntamente con su implementación, en nuestro caso, en el sistema operativo MaRTE OS. El proceso de implementación ha servido, fundamentalmente, para valorar distintas alternativas desde los aspectos de sencillez, compatibilidad y eficiencia. También ha sido clave para el desarrollo de las modificaciones realizadas a la política basada en la extracción de holgura.

Otro aspecto relevante del diseño de esta interfaz ha sido la selección de funcionalidad de las políticas objeto de estudio. Ciertas características de estas políticas han sido descartadas como consecuencia de que su complejidad, tanto de definición como de implementación, no justificaban la funcionalidad que aportaban.

Las políticas propuestas permiten al programador diseñar aplicaciones en las que las tareas aperiódicas pueden ser atendidas con preferencia frente a las tareas críticas, garantizando que estas últimas tareas mantienen sus requerimientos temporales. Las tareas aperiódicas de la aplicación pueden utilizar cualquier política de planificación, siempre

que el rango de prioridades que utilicen sea inferior al de las tareas críticas.

En la interfaz diseñada se ha previsto la posibilidad de que la aplicación pueda realizar cambios de modo. Para ello se han aportado servicios que permiten cambiar de forma dinámica los parámetros de planificación de las tareas. Se ha prestado especial atención a que la aplicación de estos cambios de parámetros permitan al programador predecir cuál será el comportamiento del sistema durante un cambio de modo.

Estas interfaces incorporan también la posibilidad de programar tareas críticas con partes opcionales. De esta forma, una tarea puede ampliar su tiempo de ejecución haciendo uso de la holgura disponible. Los servicios diseñados para este fin permiten que estas partes opcionales sean ejecutadas por la misma tarea o por tareas aperiódicas sincronizadas.

Entendemos que los objetivos establecidos para el diseño de estas interfaces han sido cumplidos en su totalidad. No obstante, no podemos dejar de comentar que la política basada en la extracción de holgura, a pesar de haberla simplificado, es relativamente compleja. Su utilización por parte del programador es muy sencilla, pero su definición y, por tanto, su implementación, son complicadas.

Análisis de Eficiencia

El objetivo de esta parte del estudio ha consistido en valorar el coste de introducir en el sistema las políticas de planificación basadas en la extracción de holgura y en la asignación dual de prioridades. Tal como ya se ha comentado, estas políticas han sido implementadas en el sistema MaRTE OS, lo que ha permitido compararlas entre sí y respecto a la política estándar de Ada y POSIX basada en prioridades fijas con desalojo.

En primer lugar, se ha caracterizado la política de planificación estándar basada en prioridades fijas con desalojo implementada en MaRTE OS, obteniendo la ecuación de cálculo de tiempos de respuesta teniendo en cuenta las actividades del sistema operativo. Esta ecuación ha sido posteriormente modificada para las políticas basadas en la extracción de holgura y en la asignación dual de prioridades. El mismo estudio se ha repetido para una versión modificada de MaRTE OS, en el que las interrupciones del temporizador hardware se tratan de forma diferida.

A partir de un ejemplo, programado por una parte utilizando servicios Ada y por otra servicios POSIX, se ha medido la sobrecarga del sistema para todas las políticas objeto de estudio y se han calculado los tiempos de respuesta para el conjunto de tareas del ejemplo. Las conclusiones más relevantes obtenidas a partir de este estudio son las siguientes:

- Los factores que más influyen en el tiempo de respuesta de las tareas, especialmente para las tareas más prioritarias, son la activación de tareas de menor prioridad y la

variación entre instantes de activación causada por las secciones en las que MaRTE OS se ejecuta con interrupciones inhibidas.

- Ambos factores están relacionados entre sí, ya que las secciones con interrupciones inhibidas de mayor duración se producen cuando se activan simultáneamente todas las tareas. Esta relación introduce un pesimismo innecesario en los tiempos de respuesta de las tareas. No obstante, no resulta sencillo plantear una ecuación que trate de forma adecuada esta relación.

- La política basada en la extracción de holgura produce tiempos de respuesta notablemente superiores a la política basada en prioridades fijas con desalojo. La política basada en la asignación dual de prioridades produce tiempos de respuesta sensiblemente mejores, pero presenta un empeoramiento conforme disminuye la prioridad de las tareas. Estos resultados están muy condicionados por dos factores, la activación de tareas menos prioritarias y la variación entre activaciones.

- Cuando se utiliza la gestión diferida de interrupciones los tiempos de respuesta varían de forma significativa. Los tiempos de respuesta de todas las políticas disminuyen, especialmente para las tareas menos prioritarias. Esta disminución se produce porque, con esta técnica, se reduce el efecto de las activaciones de tareas menos prioritarias y la variación entre instantes de activación. La política basada en la extracción de holgura se ve ampliamente beneficiada por el uso de esta técnica, ya que los tiempos de respuesta obtenidos distan mucho menos respecto a los de las otras políticas.

La eficiencia de estas políticas ha sido valorada también respecto a la sobrecarga que el sistema introduce en un intervalo de funcionamiento normal. En este caso se ha medido la utilización del procesador que realiza el sistema bajo distintos supuestos de carga aperiódica. El resultado más destacable de este análisis estriba en que las políticas de extracción de holgura y asignación dual de prioridades ofrecen resultados muy similares cuando la carga aperiódica utiliza todo el tiempo de procesamiento disponible. Este resultado plantea un interesante interrogante respecto a cuál de las dos políticas objeto de estudio es más eficiente, puesto que la extracción de holgura consume más tiempo de procesamiento generalmente, pero en estos casos se generan periodos ociosos en el procesador. Cuando los periodos ociosos desaparecen, la sobrecarga de la asignación dual de prioridades iguala a la de la extracción de holgura. No obstante, los resultados han sido obtenidos a partir de un caso concreto, y por tanto es necesaria una experimentación mucho más exhaustiva para poder confirmar estos resultados. En cualquier caso, se han realizado varios experimentos con otros conjuntos de tareas y los resultados han sido bastante similares a los descritos.

Respecto a la política basada en la extracción de holgura, se ha desarrollado una optimización de su implementación, que consiste en cancelar los cálculos de la holgura cuando la holgura del sistema se hace igual a cero. Esta optimización es sencilla y bastante efectiva. Los tiempos de respuesta de las tareas en el peor caso se aproximan a los de la planificación basada en prioridades fijas con desalojo. Los resultados obtenidos respecto a la utilización del procesador por parte del sistema son similares a los de la versión no optimizada, aunque son mejores cuando la carga aperiódica utiliza todo el tiempo de procesamiento disponible.

Finalmente, se ha evaluado la eficiencia de la utilización de las secciones de cómputo opcional en el lenguaje Ada. La sobrecarga que genera el uso de este servicio es bastante elevada, especialmente debido a la complejidad de la implementación de la sentencia de transferencia asíncrona de control. Dado este inconveniente, el uso de las secciones de cómputo adicional quedaría limitado a sistemas en los que la sobrecarga que introducen sea aceptable.

Servicios de Trazas POSIX

Estos servicios han sido implementados en MaRTE OS con el fin de disponer de una herramienta de análisis del comportamiento del sistema, así como de un instrumento de medida de tiempos. Las trazas obtenidas de MaRTE OS pueden ser representadas gráficamente, en el ordenador de desarrollo, utilizando para ello la aplicación de representación de cronogramas Quivi.

Se han desarrollado los siguientes trabajos relacionados con estos servicios. Se ha definido una interfaz POSIX/Ada para los servicios de trazas, siguiendo las directrices establecidas por el estándar POSIX.5. La adecuación de estos servicios a los sistemas de tiempo real ha sido valorada, y se estima que presentan algunas deficiencias, principalmente debidas a que su definición es muy extensa. Por último, se ha diseñado una interfaz que permite obtener métricas temporales del sistema a partir de trazas POSIX, con el fin de ofrecer al programador un método sencillo para obtener este tipo de información.

6.2. Trabajos Futuros

Tras la realización de esta tesis, se han identificado varias líneas de trabajos futuros que se describen a continuación.

Planificación de Tareas Aperiódicas

Las tareas aperiódicas que se ejecutan en tiempo de holgura pueden utilizar, en principio, cualquier política de planificación. Entre estas tareas se incluyen también las tareas

aperiódicas que ejecutan partes opcionales de tareas críticas. Un aspecto interesante a tratar sería el diseño de políticas adecuadas para este tipo de tareas, volviendo a aplicar los mismos objetivos de sencillez, coherencia y eficiencia. La interfaz para la definición de políticas de planificación a nivel de aplicación, propuesta en [5, 6], podría ser utilizada para diseñar políticas específicas para este tipo de tareas, con la ventaja añadida de que las propias tareas planificadoras podrían ser también tareas aperiódicas. De esta forma, la planificación de tareas aperiódicas no restaría tiempo de cómputo a la ejecución de las actividades críticas del sistema.

Este enfoque ha sido propuesto y probado con éxito en los sistemas ARTIS [25] y Flexible RT-Linux [67]. En estos sistemas se utiliza una interfaz propia, implementada sobre RT-Linux, poco compatible con los servicios estándar definidos por POSIX o Ada y orientada a un modelo de tareas muy específico. Nuestro objetivo es, por contra, diseñar servicios orientados a la planificación de tareas aperiódicas de uso más general y coherentes con los servicios estándar existentes en Ada y POSIX.

Comparación Exhaustiva entre Políticas

A raíz de los análisis de la eficiencia realizados, se ha concluido que se hace necesaria una experimentación mucho más exhaustiva para analizar en profundidad la sobrecarga que introduce el sistema cuando se aplican las políticas estudiadas. Tal como se ha descrito, esta sobrecarga varía notablemente en función de la carga aperiódica que interviene en el sistema, aunque otros factores también pueden influir, tales como la distribución de la carga aperiódica o la relación existente entre los periodos de las tareas críticas de la aplicación.

Con este fin se pretende programar un sistema de experimentación automático, capaz de generar conjuntos de tareas con distintas características y de obtener, por sí solo, los resultados necesarios. Esta experimentación incluiría también la política basada en el servidor esporádico definida por POSIX. En este caso el sistema de experimentación debería ser capaz de asignar los parámetros de planificación a las tareas aperiódicas que utilicen esta política.

Esta comparación podría extenderse al análisis del tiempo medio de respuesta de las tareas aperiódicas en cada política, con el fin de contrastar los resultados obtenidos por simulaciones con los que se obtendrían en un sistema real, en el cual la sobrecarga del sistema puede influir de forma muy significativa.

Servicios Reducidos de Trazas POSIX

Los servicios de trazas POSIX posiblemente sean demasiado ambiciosos para sistemas de tiempo real pequeños. En particular, no se están considerando para ser incluidos

como parte del perfil de sistema de tiempo real mínimo. Estos servicios tampoco tienen en cuenta la división de funciones que existe entre el sistema de ejecución y el de desarrollo.

A este respecto, se pretende estudiar si es posible plantear una división de estos servicios en unidades de menor tamaño, así como introducir algunas restricciones. De esta forma, un sistema operativo de tiempo real podría seleccionar las opciones que más se adecuen a sus necesidades, atendiendo a criterios de funcionalidad y eficiencia.

Caracterizar la Sobrecarga del Sistema de Trazas

El sistema de trazas introduce una sobrecarga en el sistema que puede llegar a ser considerable, especialmente si el número de eventos generados por el sistema es muy elevado. Cuando los servicios de trazas se utilizan como instrumento de medida, es importante caracterizar en detalle esta sobrecarga. En este aspecto se han realizado ya algunos avances [44], en los que se han evaluado distintos métodos de medición de tiempos, entre ellos los servicios de trazas. Como objetivo, en este trabajo se persigue poder conocer en detalle los tiempos mínimo y máximo que cuesta generar un evento de traza, así como el tiempo medio y su distribución, con el fin de poder estimar correctamente el efecto del instrumento de medida, en este caso los servicios de trazas, en las mediciones realizadas.

Ampliar los Servicios de Obtención de Métricas

Ofrecer una interfaz de obtención de métricas al usuario parece una idea interesante. La interfaz propuesta en esta tesis es una primera aproximación a este problema, y se pretende ampliar su funcionalidad en distintos aspectos. En principio, los servicios diseñados miden la actividad del sistema en relación con las tareas, pero no sirven para medir la utilización de otros elementos del sistema tales como mutex, variables condición, señales, etc. También sería interesante definir cuál debería ser el conjunto de métricas que todo sistema de tiempo real debería ofrecer, con el fin de que el programador pudiese evaluar el rendimiento del sistema en el cual se ejecutan sus aplicaciones.

Actualmente se dispone también, para RT-Linux, de un sistema de trazas POSIX [66] y una aplicación de obtención de propiedades temporales a partir de trazas POSIX [68]. Los trabajos ya realizados en este sistema y en MaRTE OS serán ampliados para ofrecer un entorno completo de análisis de prestaciones para ambos sistemas.

Bibliografía

[1] (1983) “Reference Manual for the Ada Programming Language”. ANSI/MIL-STD 1815A.

[2] (1985) “International Standard ISO/IEC 8652:1995(E): Information Technology - Programming Languages - Ada. Ada Reference Manual”.

[3] Aldea, M. y González, M. (2001) “MaRTE OS: An Ada Kernel for Real-Time Embedded Applications”. International Conference on Reliable Software Technologies, Ada-Europe-2001.

[4] Aldea, M. y González, M. (2001) “Extending Ada’s Real-Time System Annex with the POSIX Scheduling Services”. 10th International Real-Time Ada Workshop, Las Navas del Marqués (Ávila), Spain, ACM Ada Letters.

[5] Aldea, M. y González, M. (2002) “A POSIX-Ada Interface for Application-Defined Scheduling”. International Conference on Reliable Software Technologies, Ada-Europe-2002. Lecture Notes in Computer Science No. 2361, pp.136-150.

[6] Aldea, M. y González, M. (2002) “POSIX-Compatible Application-Defined Scheduling in MaRTE OS”. Proceedings of 14th Euromicro Conference on Real-Time Systems, Vienna, Austria, IEEE Computer Society Press, pp. 67-75.

[7] Audsley, N. C., Burns, A., Richardson, M. F. y Wellings, A. J. (1991) “Hard Real-Time Scheduling: The Deadline Monotonic Approach”. Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software, Atlanta, GA, USA. pp. 127-132.

[8] Audsley, N. C. (1993) Flexible Scheduling of Hard Real-Time Systems”. D.Phil. Thesis, Department of Computer Science, University of York, UK.

[9] Audsley, N., Burns A., Davis, R.I. y Wellings, A.J. (1994) “Integrating Best Effort and Fixed Priority Scheduling”. IFIP Workshop on Real-Time Programming, Lake Konstanz.

[10] Audsley, N., Burns A., Davis, R.I. y Wellings, A.J. (1994) “Appropriate Mechanisms for the Support of Optional Processing in Hard Real-Time Systems”. 11th IEEE Workshop on Real-Time Operating Systems and Software.

[11] Audsley, N. C., Burns, A., Davis, R., Tindell, K. y Wellings, A. (1995) “Fixed Priority Preemptive Scheduling: an Historical Perspective”. Real-Time Systems Vol. 8, pp. 173-198.

[12] Baker, T.P. y Shaw, A. (1988) “The Cyclic Executive Model and Ada”. IEEE Real-Time Systems Symposium, pp. 120-129.

[13] Bernat,G. y Burns, A. (1999) “New Results on Fixed Priority Aperiodic Servers”. 20th IEEE Real-Time Systems Symposium. RTSS’99. Phoenix, USA.

[14] Bernat,G. y Burns, A. (2001) “Implementing a Flexible Scheduler in Ada”. 6th International Conference on Reliable Software Technologies. Ada Europe-2001.

[15] Burns, A. y Wellings, A.J. (1993) “Dual Priority Assignment: A Practical Method for Increasing Processor Utilization”. Proceedings of the 5th Euromicro Workshop on Real-Time Systems, pp. 48-55.

[16] Burns, A., Tindell, K. y Wellings, A.J. (1995) “Effective Analysis for Engineering Real-Time Fixed Priority Schedulers”. IEEE Transactions on Software Engineering, Vol.44, No.5, pp. 100-106.

[17] Burns, A. y Wellings, A.J. (1995) “Engineering a Hard Real-time System: From Theory to Practice”. Software Practice and Experience, Vol. 25, No. 7, pp. 705-726.

[18] Burns, A. y Wellings, A.J. (1996) “Dual Priority Scheduling in Ada 95 and Real-Time POSIX”. Workshop on Real-Time Programming, pp. 45-50.

[19] Burns, A. (1999) “The Ravenscar Profile”. ACM Ada Letters, Vol. 19, No. 4, pp. 49-52.

[20] Davis, R.I., Tindell, K.W. y Burns, A. (1993), “Scheduling Slack Time in Fixed Priority Preemptive Systems”. IEEE Real-Time Systems Symposium, pp. 222-231.

[21] Davis, R.I.(1993), “Approximate Slack Stealing Algorithms for Fixed Priority Preemptive Systems”. Department of Computer Science, University of York. Technical Report YCS217.

[22] Davis. R.I. y Wellings, A.J. (1995) “Dual Priority Scheduling”. Proceedings IEEE Real-Time Systems Symposium, pp. 100-109.

[23] Espinosa, A., Julián, V., Carrascosa, C., Terrasa, A. y García-Fornes, A., "Programming Hard-Real Time Systems with Optional Components in Ada". Lecture Notes in Computer Science, Reliable Software Technologies, AdaEurope-98, Vol. 1411, pp. 102-111.

[24] García-Fornes, A. y Botti, A. (1995) "ARTIS: Una arquitectura para Sistemas Inteligentes de tiempo real". CAEPIA'95, pp. 161-174.

[25] García-Fornes, A. (1996) "ARTIS: Una Arquitectura para Sistemas Inteligentes de Tiempo Real". PhD Thesis.

[26] García-Fornes, A., Terrasa, A., Botti, V. (1997) "Técnicas de Planificación de Tareas Aperiódicas en Sistemas de Tiempo Real Estricto". Novática, No. 129, pp. 22-30.

[27] Gonzalez, M. y Sha, L. (1991) "An Application-Level Implementation of the Sporadic Server". Technical Report CMU/SEI-91-TR-26, ESD-91-TR-26.

[28] Gonzalez, M., Klein, M.H., y Lehoczky, J.P. (1991) "Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority". IEEE Real-Time Systems Symposium, pp. 116-128.

[29] Gonzalez, M., Gutiérrez J.J. y Palencia, J.C. (1997) "Implementing Application-Level Sporadic Server Schedulers in Ada 95.". Lecture Notes in Computer Science , Reliable Software Technologies Ada-Europe'97, Springer Verlag, Vol. 1251 pp. 125-136.

[30] Gonzalez, M. Aldea, M., Gutierrez, J.J. y Palencia, J.C. (1998) "Implementing and Using Execution Time Clocks in Ada Hard Real-Time Applications". Lecture Notes in Computer Science 1411, Reliable Software Technologies Ada-Europe'98, Springer Verlag, pp. 91-101,

[31] Harter, P. K. (1984) "Response Times in Level Structured Systems". Department of Computer Science, University of Colorado, USA. CU-CS-269-94.

[32] Harter, P. K. (1987) "Response Times in Level Structured Systems". ACM Transactions on Computer Systems. Vol. 5, No. 3, pp. 232-248.

[33] Joseph, M. (1985) "On a Problem in Real-Time Computing". Information Processing Letters. Vol. 20, No. 4, pp. 179-207.

[34] Joseph, M. y Pandya, P. (1986) "Finding Response Times in Real-Time Systems". The Computer Journal (British Computer Society). Vol. 29, No. 5, pp. 390-395.

[35] Katcher, D., Arakawa, H. y Stronsider, J. (1993) “Engineering and Analysis of Fixed Priority Schedulers”. IEEE Transactions on Software Engineering, Vol. 19, pp. 920-934.

[36] Lampson, B.W., y Redell, D. (1980) “ Experience with Processes and Monitors in Mesa”. Communications of the ACM, Vol. 23, No. 2, pp 105-117.

[37] Lehoczky, J. P., Sha, L. y Stronsider, J.K. (1987) “Enhanced Aperiodic Responsiveness in Hard Real-Time Environments”. Proceedings IEEE Real-Time Systems Symposium, pp. 261-270.

[38] Lehoczky, J. P., Sha, L. y Ding, Y. (1989) “The Rate-Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior”. Proceedings IEEE Real-Time Systems Symposium, pp. 166-171.

[39] Lehoczky, J. P. y Ramos-Thuel, S. (1992) “An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed Preemptive Systems”. Proceedings IEEE Real-Time Systems Symposium, pp. 110-123.

[40] Leroy, P. “An Invitation to Ada 2005”. Ada User Journal, Vol. 23, No. 4, pp. 230-237.

[41] Leung, J.Y.T. y Whitehead, J. (1982) “On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks”. Performance Evaluation (Netherlands), Vol. 2, No. 4, pp. 237-250.

[42] Liu, C. L. y Layland, J. W. (1973) “Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment”. Journal of the ACM, Vol. 20, No. 1, pp 44-61.

[43] Locke, C.D. (1992) “Software Architecture for Hard Real-Time Applications: Cyclic Executives vs. Fixed Priority Executives”. Real-Time Systems, Vol. 4, No. 1, pp. 37-53.

[44] Lorente V., Espinosa, A., García-Fornes, A. y Crespo, A. (2003) “Measuring Execution time of code by means of Tracing POSIX”. 27th IFAC/IFIP/IEEE Workshop on Real-Time Programming, WRTP’03.

[45] Miranda, J. “A Deatailed Description of the GNU Ada Run Time”. En http://www.iuma.ulpgc.es/users/jmiranda/gnat-rts/main.htm.

[46] The Open Group (2002) “The Single Unix Specification. Version 3”.

[47] IEEE Standard 1003.1b:1993, (1993) “Standard for Information Technology - Portable Operating System Inteface (POSIX) - Part 1: System Application Interface (API)

- Amendment 1: Realtime Extension [C Language]. The Institute of Electrical and Electronics Engineers.

[48] IEEE Standard 1003.1c:1995, (1995) "Standard for Information Technology - Portable Operating System Inteface (POSIX) - Part 1: System Application Interface (API) - Amendment 2: Threads Extension [C Language]. The Institute of Electrical and Electronics Engineers.

[49] IEEE Standard 1003.1:1996, (1996) "Standard for Information Technology - Portable Operating System Inteface (POSIX) - Part 1: System Application Interface (API) [C Language]. The Institute of Electrical and Electronics Engineers.

[50] IEEE Standard 1003.13:1998, (1998) "Standard for Information Technology - Standardized Application Environment Profile (AEP)- POSIX Realtime Application Support ". The Institute of Electrical and Electronics Engineers.

[51] IEEE Standard 1003.5c:1998, (1998) "IEEE Standard for Information technology-POSIX® Ada Language Interfaces- Part 1: Binding for System Application Program Interface (API)". The Institute of Electrical and Electronics Engineers.

[52] IEEE Standard 1003.1d:1999, (1999) "Standard for Information Technology - Portable Operating System Inteface (POSIX) - Part 1: System Application Interface (API) - Amendment d: Additional Realtime Extensions [C Language]. The Institute of Electrical and Electronics Engineers.

[53] IEEE Standard 1003.1j:1999, (1999) "Standard for Information Technology - Portable Operating System Inteface (POSIX) - Part 1: System Application Interface (API) - Amendment j: Advanced Realtime Extensions [C Language]. The Institute of Electrical and Electronics Engineers.

[54] IEEE Standard 1003.1q:2000, (2000) "Standard for Information Technology - Portable Operating System Inteface (POSIX) - Part 1: System Application Interface (API) - Tracing [C Language]. The Institute of Electrical and Electronics Engineers.

[55] IEEE Standard 1003.1:2001, (2001) "Standard for Information Technology - Portable Operating System Inteface (POSIX) - Part 1: System Application Interface (API) [C Language]. The Institute of Electrical and Electronics Engineers.

[56] IEEE Standard 1003.13:2002, (2002) "Draft Standard for Information Technology - Standardized Application Environment Profile (AEP)- POSIX Realtime Application Support". The Institute of Electrical and Electronics Engineers.

[57] Rajkumar, R. (1989) "Task Synchronization in Real-Time Systems". PhD thesis, Carnegie Mellon University.

[58] Real, J. (2000) “Protocolos de Cambio de Modo para Sistemas de Tiempo Real”. PhD thesis, Universidad Politécnica de Valencia.

[59] Schonberg, E. y Banner, B. (1994) “The GNAT Project: A GNU-Ada 9X Compiler”. TRI-Ada 1994 pp. 48-57

[60] Sha, L., Lehoczky, J.P y Rajkumar, R. (1986) “Solutions for Some Practical Problems in Prioritised Preemptive Scheduling”. Proceedings IEEE Real-Time Systems Symposium. pp. 181-191.

[61] Sha, L., Rajkumar, R. y Lehoczky, J.P. (1990) “Priority Inheritance Protocols: An Approach to Real-Time Synchronization”. IEEE Transactions on Computers. Vol. 39, No. 9, pp. 1175-1185.

[62] Sprunt, B., Sha, L. y Lehoczky, J.P. (1989) “Aperiodic Task Scheduling for Hard-Real-Time Systems”. Vol. 1, No. 1, pp. 27-60.

[63] Stankovic, A., Spuri, M., Ramamritham, K. y Buttazzo, G. ‘(1998) “Deadline Scheduling for Real-Time Systems EDF and Related Algorithms”. Kluwer Academic Publishers.

[64] Taft, S.L., Duff, A., Brukardt, R.L. y Ploedereder, E. (2000) “Consolidated Ada Reference Manual. Language and Standard Libraries”. Springer.

[65] Terrasa A.M. (2000) “Flexible Real-Time Linux. A New Environment for Flexible Hard Real-Time Systems”. PhD Thesis.

[66] Terrasa, A.M., Pachés, I. y García-Fornes, A. (2001) “An evaluation of POSIX Trace Standard implemented in RT-Linux”. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2001), pp. 30-37.

[67] Terrasa, A.M., García-Fornes, A. y Botti, V. (2002) “Flexible real-time linux”. Real-Time Systems Journal, pp. 149-170.

[68] Terrasa, A.M. y Bernat, G. (2003) “Extracting Temporal Properties of Real-Time Systems by Automatic Tracing Analysis”. Lecture Notes in Computer Science (pendiente de publicación).

[69] Tindell, K. (1992) ”Fixed Priority Scheduling for Hard Real-Time Systems”. PhD thesis.

[70] Tindell, K., Burns, A. y Wellings, A. J. (1994) “ An Extendible Approach for Analysing Fixed Priority Hard Real-Time Tasks”. Real-Time Systems, Vol. 6, No. 2, pp. 133-151.

[71] Yodaiken, V. (1999) “An RT-Linux Manifesto”. Proceedings of the 5th Linux Expo, Raleigh, North Carolina, USA.