ESCUELA TÉCNICA SUPERIOR DE INGENIEROS
INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA

Trabajo Fin de Carrera

CARACTERIZACIÓN TEMPORAL DE SISTEMAS OPERATIVOS DE TIEMPO REAL

Para acceder al Titulo de

INGENIERO DE TELECOMUNICACIÓN

Autor: Marta Alonso Domínguez

Marzo - 2005

Agradecimientos

A Michael González Harbour por darme la oportunidad de hacer este proyecto.

A todos los componentes del Grupo Computadores y Tiempo Real, por su inestimable ayuda y buen ambiente de trabajo.

A mi jefe, y a mis compañeros de trabajo, por su amistad, y por la flexibilidad permitida en el horario laboral, imprescindible para compaginar la actividad laboral con la realización del proyecto.

A mi familia, por apoyarme, comprenderme, animarme cuando me hacía falta, y aguantar tantas horas sin verme.

A muchos de mis compañeros de clase, que son ante todo amigos.

A mis amigos de siempre, por conseguir despejarme la cabeza y ayudarme (o intentarlo) a ordenar mis ideas.

Resumen

Las aplicaciones de tiempo real están adquiriendo cada día mayor importancia en nuestra vida diaria. Hoy en día, podemos encontrarlas en varios entornos distintos, como sistemas de control en aviones, automóviles o trenes, telecomunicaciones, teléfonos móviles, procesos industriales, equipamiento médico, etc.

La complejidad de muchos de estos sistemas hace aconsejable el uso de un sistema operativo, que al formar parte integral del sistema debe tener servicios con tiempos de respuesta acotados. Los requisitos y características que han de reunir estos sistemas, se encuentran definidos en un perfil concreto dentro del estándar POSIX de interfaces de sistemas operativos portables. Este perfil, denominado "Sistema de Tiempo Real Mínimo" (PSE51), reúne las funcionalidades básicas que han de ser implementadas con el fin de lograr un sistema pequeño y eficiente con tiempos de respuesta predecibles.

Este perfil constituye el objeto de estudio del presente Proyecto de Fin de Carrera. Por un lado, se estudia la caracterización teórica, composición de servicios y funcionamiento que componen este tipo de sistemas. Y por otro lado se realiza el diseño y desarrollo de una herramienta que permita evaluar la respuesta temporal de algunos de los servicios que componen dicho perfil. Esta caracterización temporal es de importante utilidad en dos fases muy diferenciadas. Por un lado, constituye una información que facilita el análisis y la evaluación del sistema operativo en su fase de desarrollo, lo que permite comprobar el funcionamiento de los servicios que realiza, así como el rendimiento que presenta para diferentes implementaciones, y detectar posibles "puntos débiles" en el diseño. Por otro lado, esa evaluación del funcionamiento y del rendimiento facilita al usuario un conjunto de datos con los que puede realizar comparaciones entre diferentes sistemas operativos que ofrezca el mercado, o simplemente comprobar si uno en concreto cumple los requisitos temporales necesarios para el sistema en que va a ser implementado. En tercer lugar, facilita el análisis del tiempo de respuesta de las aplicaciones que corren bajo ese determinado sistema.

La herramienta que se describe en esta memoria permite conocer los tiempos característicos de varios de los servicios que ofrece el sistema descrito en el perfil mínimo del estándar. El motivo por el que han sido seleccionados esos servicios en concreto es consecuencia de la experiencia del grupo de Computadores y Tiempo Real en el diseño de controladores para robots industriales. Los servicios analizados en esta herramienta son los que mostramos a continuación:

- *Mecanismos de sincronización: mutexes y variables condicionales.*
- *Gestión del tiempo: relojes, temporizadores y retrasos de alta resolución.*
- *Señales de tiempo real.*
- *Gestión de memoria: reserva y liberación.*
- *Sobrecarga producida en el sistema por interrupciones.*

A partir de esta herramienta se ha llevado a cabo la caracterización temporal sobre dos sistemas operativos con aplicaciones diferentes. Por un lado, se han ejecutado estos experimentos sobre MaRTE. Este es un sistema operativo de tiempo real mínimo para aplicaciones empotradas, diseñado según las especificaciones recogidas en el perfil mínimo del estándar. Por otro lado, se ha llevado a cabo la realización de las medidas sobre un sistema de tiempo compartido Linux, sin funcionalidades específicas de tiempo real.

A pesar de que las prestaciones equipo sobre el que se han llevado a cabo las pruebas del sistema Linux son bastante superiores a las del sistema MaRTE hemos podido comprobar que las diferencias son bastante notables, y acordes con lo que era previsible para cada uno de estos tipos de sistemas. Para empezar, se ha podido comprobar que MaRTE ofrece unos tiempos de respuesta predecibles, en el sentido de que hay poca diferencia entre los casos peor, mejor y promedio, mientras que en las medidas realizadas sobre Linux hemos podido observar que en ocasiones se producen saltos en la ejecución, como consecuencia de ser un sistema de tiempo compartido. Además, y pese a que los tiempos mínimos registrados son mejores en Linux, los tiempos máximos y promedios, que resultan más críticos, habitualmente son bastante peores en Linux.

La herramienta desarrollada es portable a cualquier sistema que cumpla el estándar POSIX de sistema de tiempo real mínimo, y puede ser utilizada por ello para caracterizar temporalmente los servicios de cualquier sistema operativo de este tipo. Ello permite tener una buena estimación de las prestaciones esperables, así como comparar entre diferentes sistemas operativos.

Abstract

Real time applications are acquiring more importance in our daily life. Nowadays we can find them in different environments such as control systems in airplanes, vehicles or trains, telecommunications, mobile phones, industrial processes, medical equipment, etc.

The complexity of most of these systems makes necessary the use of an operating system that, as an integral part of the system, must have services with bounded time responses. The characteristics and requirements that these systems must have are defined in a certain profile in the POSIX standard for Portable Operating System Interfaces. This profile, called "Minimal Real Time System" (PSE51), describes the basic functionalities that these systems must implement, in order to manage a small and efficient system, with deterministic time responses.

This profile constitutes the objective of the present Master Degree Project. On the one hand, this study includes the theoretical characterization, services composition and functionalities that this kind of systems offers. On the other hand, a set of test programs are designed and developed, allowing us to evaluate the time response of some of the services that compose the profile mentioned above. This temporal characterization is very important in two different aspects. Firstly, it constitutes information that facilities the operating system analysis and evaluation in its development stage. This allows us to evaluate the functionality of the services as well as the performance of different implementations, and the detection of potential "weak points" in the design. Besides, this evaluation of the functionality and performance provides a data set to compare different operating systems in the market, or simply to check if one in particular is appropriate for the system in which it is going to be used. Finally, it facilitates an analysis of the application time response when running in a certain system.

The tool described in this document allows us to know the time response of some of the services that the system defined in the profile offers. The rationale for selecting these particular services is a consequence of the "Computers and Real Time" group experience in the design of controllers for industrial robots. The services analyzed in this tool are the following:

- *Synchronization mechanisms: mutexes and condition variables.*
- *Time management: clocks, timers and high resolution delays.*
- *Real Time Signals.*
- *Memory management: reserve and free.*
- *Overhead produced in the system due to interruptions.*

Making use of the designed tool, we have carried out the time characterization of two operating systems, designed with different purposes. On the one hand, we have executed these experiments on MaRTE. This is a minimal real time operating system for embedded applications, designed following the specifications described in the minimal

profile of the standard. On the other hand, we have made the same measurements over a Linux sharing time operating system which has no specific real time functionalities.

In spite of the fact that the Linux computer is much faster than MaRTE's, we have noticed that the differences are very relevant, and they are according to what was predictable for each system. Firsty, we have verified that MaRTE offers deterministic time responses, in the sense that there is a very small difference among the worst, the best and the average cases, while in Linux we have noticed that sometimes there are "jumps" in execution, as a consequence of being a time sharing system. Besides, although the minimum registered times are in Linux, the maximum and the average times - which are more critical - are usually worse in Linux.

The developed tool is portable to any other system which follows the "Minimal Real Time System" profile described in the POSIX standard, and therefore it can be used to make a temporal characterization of the services of any operating system of this kind. This allows us to have a good estimation of possible benefits as well as to compare among different operating systems.

"Caracterización temporal de Sistemas Operativos de Tiempo Real"

0. Índice de tablas y figuras

Página

1. Introducción

2. Sistemas empotrados de tiempo real

3. Estrategia de medida

4. Resultados y evaluación

1.- Introducción

1. Introducción

Las aplicaciones de tiempo real están tomando cada vez más importancia en nuestra vida diaria, y podemos encontrarlas en varios y diferentes entornos como sistemas de control en aviones, vehículos o trenes, telecomunicaciones, teléfonos móviles, procesos industriales, equipamiento médico, video y audio digital, electrodomésticos, etc.

Un sistema de tiempo real se caracteriza principalmente por su fuerte interacción con un entorno que varía con el tiempo. Como consecuencia de ello, los tiempos de respuesta de las acciones que realiza el sistema operativo deben efectuarse dentro de unos intervalos temporales limitados. En estos sistemas, una respuesta tardía es tan inútil y perjudicial como una respuesta incorrecta, de modo que ha de estudiarse a la perfección la capacidad de reacción y el comportamiento del hardware y software que emplean. De hecho, uno de los parámetros más importante es el rendimiento en el peor de los casos posibles, dado que el tiempo medio, o el tiempo esperado, son valores insuficientes.

Un sistema operativo de tiempo real es un sistema operativo que ha sido desarrollado específicamente para realizar tareas de tiempo real. Los criterios de diseño de estos sistemas operativos difieren de los de los sistemas de tiempo compartido habituales, ya que la capacidad de planificación es prioritaria sobre la potencia de cálculo, no es suficiente garantizar tiempos promedios, y no se busca un reparto equitativo de tareas sino que se mantenga en todo momento la estabilidad del sistema. Por estos motivos, las características principales de un sistema operativo de tiempo real son las que se enuncian a continuación:

- Ser predecible.
- Garantizar el peor caso de latencia de interrupciones.
- Garantizar el peor caso de cambio de contexto.
- Garantizar el tiempo de peor caso de los servicios usados en acciones de tiempo real.

Como planificación del sistema entendemos un conjunto de políticas y mecanismos incorporados al sistema operativo, a través de un modulo denominado planificador, que se encarga de decidir qué tareas han de ser realizadas primero y qué orden de ejecución debe seguirse. Su objetivo principal es el máximo aprovechamiento del sistema.

Un sistema operativo de tiempo real es predecible cuando su planificación tiene capacidad para cumplir todos los plazos, es decir, se puede calcular cuál es el máximo

tiempo que tarda en realizarse una llamada cualquiera. Cuando se produce un estado de sobrecarga, el sistema operativo ha de mantenerse estable, y para ello han de cumplir, al menos, los plazos de las tareas críticas.

Las interrupciones son un elemento clave de los sistemas operativos de tiempo real. Por este motivo, es igualmente importante caracterizar el tiempo máximo que transcurre desde que se produce una interrupción hasta que se ejecuta su rutina de atención a la interrupción.

La tercera característica enumerada hace referencia al cambio de contexto entre tareas. El contexto de una tarea guarda el estado del procesador antes de que otra tarea tome el control de este. Habitualmente consiste en un puntero apuntando a la siguiente instrucción a ejecutar, la dirección actual de la parte alta de la pila ("*stack*"), y el contenido de los registros y *flags* de estado. Por lo tanto el cambio de contexto consiste en guardar "el contexto" de la tarea anterior, y cargar el de la nueva tarea que va a ejecutarse.

Por último, también ha de asegurarse que se cumplan los peores tiempos en los servicios de tiempo real y se cumplan las restricciones temporales impuestas para ellos, con el fin de garantizar los tiempos de respuesta establecidos y asegurar el correcto funcionamiento del sistema.

Como ya se ha indicado, las aplicaciones de tiempo real producen una acción o una respuesta ante un evento externo en un tiempo definido y de una manera predecible. No todas estas aplicaciones tienen las mismas restricciones temporales. Mientras muchas de estas aplicaciones requieren gran velocidad de computación, otras pueden no tener restricciones temporales tan críticas, sin embargo en todos los casos han de cumplirle los límites de tiempo que se hayan requerido [GON01a].

Los sistemas operativos de tiempo real se clasifican en dos grupos:

- Sistemas operativos de tiempo real estricto
- Sistemas operativos de tiempo real no estricto

En los sistemas operativos correspondientes al primer grupo es imprescindible que se cumplan los plazos requeridos para cada ejecución, ya que si no el resultado será inválido. Por ejemplo, en el control de los frenos de un automóvil, si el frenado no se realiza en un intervalo de tiempo máximo, puede provocar graves consecuencias. Sin embargo, en un sistema operativo de tiempo real no estricto, estas restricciones no son tan exigentes. El resultado de un determinado cálculo puede ser útil aunque este no sea proporcionado dentro de los plazos temporales especificados. Un ejemplo de este tipo de sistemas es el muestreo de audio. Si un paquete de datos se pierde o retrasa, es obvio que la calidad de audio se degrada, pero no necesariamente dejará de ser audible [OBENL].

Casi todos los sistemas de tiempo real son "sistemas empotrados". Estos sistemas están integrados en un sistema mayor y realizan una función determinada o un grupo de ellas. Las prestaciones de estos sistemas se ven a menudo limitadas por motivos de tamaño, peso, consumo o coste, y por estas razones lo más común es que no

dispongan de un terminal de propósito general que actúe como interfaz al usuario, ni de un sistema de ficheros o almacenamiento magnético, y en muchos casos tanto su procesador como su capacidad de memoria suelen ser muy reducidas [ALD02].

Sin embargo, gracias a los avances tecnológicos, estas limitaciones son menores cada día, y la utilización de estos sistemas se ha ido extendiendo, llegando incluso a la mayoría de los ámbitos cotidianos. Así, podemos encontrar computadores empotrados en televisiones, juguetes, aparatos reproductores de DVD, electrodomésticos, etc. Esta nueva generación de computadores goza de procesadores cada vez más potentes, memorias de menor coste y una mayor capacidad de integración [ALD02].

1.1. Resumen de los requisitos de un sistema operativo de tiempo real

Como consecuencia de la necesidad de responder a eventos externos en un plazo de tiempo acotado, y de controlar los dispositivos externos, los sistemas de tiempo real se implementan habitualmente con funcionalidad para ejecutar múltiples procesos concurrentes, por lo que un sistema operativo de tiempo real ha de tener soporte multiproceso, con el fin de garantizar la concurrencia de tareas [OBENL].

Por otro lado, y debido a que las restricciones temporales han de respetarse en cada tipo de evento, el sistema operativo ha de contar con algún tipo de noción de prioridad, con el fin de poder expresar el grado de urgencia de cada tarea y predecir el comportamiento temporal del sistema en el peor caso. Además, las tareas han de tener capacidad de comunicación entre ellas, de modo que el sistema operativo ha de proporcionar capacidades tanto de sincronización como de comunicación entre tareas [OBENL].

Un sistema operativo de tiempo real ha de soportar también funcionalidades temporales como son temporizadores y relojes de alta resolución. Los temporizadores son empleados principalmente en tareas periódicas, y para detectar errores del sistema producidos por superar el plazo límite. Los relojes son necesarios para llevar la cuenta del tiempo, ya que en las aplicaciones de tiempo real se hace necesario conocer el tiempo con una precisión de milisegundos, e incluso de microsegundos [OBENL].

En relación al funcionamiento, el sistema operativo ha de comportarse de forma predecible, y añadir una sobrecarga a las operaciones lo más pequeña posible. Como ya se ha indicado previamente, el sistema operativo ha de ser predecible. Esto implica que el tiempo requerido para todas las operaciones, incluyendo las funciones propias del sistema operativo, han de ser acotadas. Con objeto de conseguir tiempos de respuesta más rápidos, el planificador ha de ser expulsor, lo que significa que si está procesando una tarea de baja prioridad, deber tener capacidad para dejar de procesar dicha tarea y atender a una tarea de mayor prioridad [OBENL].

1.2. La familia de estándares POSIX

Debido a los avances tecnológicos, constantes durante los últimos años, y el consiguiente incremento en la complejidad de las aplicaciones, se fue haciendo cada vez más necesaria la utilización de lenguajes de alto nivel y sistemas operativos, cuando en al principio se programaban las aplicaciones directamente sobre un lenguaje ensamblador. Además, estos sistemas no trabajan aislados sino que han de interoperar con otros sistemas comerciales, y es muy probable que a lo largo de su ciclo de vida hayan de sufrir modificaciones o ampliaciones de funcionalidad. Por todo ello se hizo imprescindible la definición de un estándar capaz de dotar al software de interoperabilidad y portabilidad, además de propiciar una reducción del coste producido por una posible ampliación de funcionalidad en el futuro [OBENL].

En este contexto nace el estándar POSIX, siglas que corresponden a: "*Portable Operating Systems Inteface*", Interfaz de Sistemas Operativos Portables, con el fin de unificar las diferentes versiones de sistemas operativos UNIX de los distintos fabricantes y usuarios. Fue desarrollado en los años 80 y 90 en el marco de la *Computer Society* del IEEE, con referencia 1003, y constituye un estándar a nivel internacional (ISO/IEC 9945) [OBENL].

El estándar POSIX define la interfaz que los sistemas operativos han de presentar de cara a las aplicaciones, así como la semántica de los servicios que estos ofrezcan, con un doble objetivo [ALD02]:

- Lograr portabilidad a nivel de código fuente.
- Conseguir una adaptación más fácil a nuevos entornos de trabajo.

La familia de estándares POSIX incluía más de 30 estándares individuales, que abarcan desde las especificaciones mínimas de los servicios de un sistema operativo, hasta métodos de comprobación de conformidad con el estándar [OBENL]. Actualmente, muchos de estos estándares se han unificado en un pequeño número de documentos. Podemos clasificarlos en tres grandes grupos [ALD02]:

1. ***Estándares Base****:* Los estándares base están escritos en lenguaje C, y definen la interfaz básica entre el sistema operativo y las aplicaciones, tanto en lo referente a la síntesis como a la semántica. Es decir, definen el conjunto de funciones que permiten a los programas acceder a los servicios que proporciona el sistema operativo.

Estándar	Breve Descripción
1003.1	Servicios básicos del sistema operativo
1003.13	Perfiles de sistemas de tiempo real
1003.3	Métodos para comprobar la conformidad con POSIX
1003.5	Interfaces en lenguaje Ada

Tabla 1.1. Algunos estándares POSIX

El estándar 1003.1 define, por un lado, la interfaz básica para acceder a los servicios del sistema operativo. En él se definen los servicios básicos de un sistema UNIX convencional [GON01a]:

- Gestión de procesos: creación, ejecución, terminación, coordinación entre procesos (señales), y herramientas de temporización de poca resolución.
- Entorno del proceso: identificación, usuarios y grupos, calendario, tiempos de ejecución, variables de configuración, etc.
- Sistema de ficheros y directorios: creación y eliminación, colas FIFO, características, modo y estado de los ficheros.
- Operaciones de entrada y salida: lectura, escritura y operaciones de control. Entrada y salida por terminal.
- Bases de datos del sistema (de usuarios y grupos).

Por otro lado, el estándar 1003.1 en su versión actual modificada define los servicios requeridos para tiempo real. El conjunto de estos servicios puede resumirse como sigue [OBENL]:

- Temporizadores: temporizadores periódicos, la expiración se notifica mediante señales de tiempo real.
- Planificación expulsora por prioridad, con un mínimo de 32 niveles.
- Señales de tiempo real: señales adicionales con múltiples niveles de prioridad.
- Semáforos: especificación y memoria de semáforos contadores.
- Colas de mensajes: especificación de colas de mensajes de memoria.
- Memoria compartida: especificación de zonas de memoria compartida para múltiples procesos.
- Inhibición de memoria virtual: funciones para evitar el intercambio virtual de páginas de memoria física.

Un thread, también llamado *hilo*, o *proceso ligero*, es una unidad básica de utilización de recursos que consiste en un contador de programa, un conjunto de registros y un espacio de pila. Comparte con otros threads una sección de código, una sección de datos y algunos recursos del sistema operativo.

En el estándar POSIX.1 también se definen las cualidades del soporte multithread agrupándolas en las siguientes áreas [OBENL]:

- Control de threads: creación, eliminación y gestión de threads individuales.
- Planificación expulsora por prioridad: extensión que incluye en la planificación por prioridad de procesos, la planificación local de threads de un mismo proceso, o global, de todos los threads de los procesos.
- Mutexes: empleados para proteger secciones críticas de código. Los mutexes incluyen soporte para protocolos de herencia de prioridad y techo de prioridad para prevenir inversiones de prioridad.

- Variables condicionales: en conjunto con los mutexes pueden ser empleados para crear monitores de estructuras de sincronización.
- Señales: capacidad para enviar señales a threads individuales.

2. ***Interfaces para diferentes lenguajes de programación:*** Como se acaba de indicar el estándar básico está escrito en lenguaje C. En este grupo de estándares se define el interfaz a los servicios que proporciona el sistema operativo en otros lenguajes de programación. Los lenguajes para los que se ha definido el interfaz para este estándar hasta el momento son Ada (POSIX.5) y Fortran (POSIX.9).

3. ***Perfiles de entorno de aplicación:*** En ellos se definen subconjuntos POSIX que contienen los servicios necesarios para un ámbito de aplicación determinado, con el objetivo de que los sistemas operativos destinados a áreas muy específicas no hayan de implementar todas las funcionalidades POSIX.

   La finalidad es proporcionar un conjunto de interfaces coherente que facilite a fabricantes y consumidores de software la creación de un entorno de trabajo uniforme para describir y especificar las características del sistema operativo. Esto permite que los programadores creen aplicaciones que puedan ser fácilmente portables a otros sistemas que cumplan con los requisitos del perfil.

   Por estos motivos los perfiles representan un importante mecanismo para definir de forma estándar un conjunto bien definido de implementaciones de sistemas operativos, adecuadas para áreas de aplicación específicas.

Un sistema operativo de tiempo real que implemente completamente el estándar POSIX sería muy grande y complejo, por esta razón, y tal y como se ha mencionado previamente, se han definido en el estándar 1003.13 cuatro subconjuntos de servicios del sistema operativo (perfiles de entorno de aplicación) que se han orientado a cuatro tipo de plataformas comúnmente en las aplicaciones comerciales de tiempo real [ALD02].

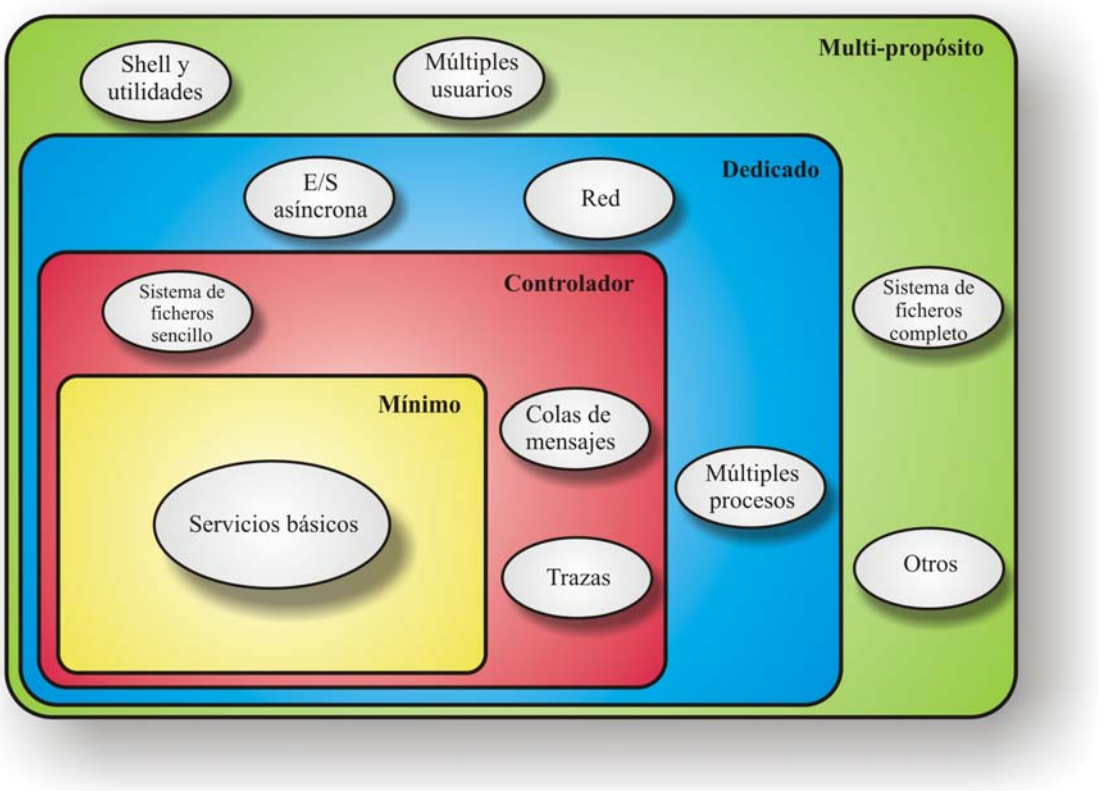

Figura 1.1. Funcionalidad contenida en los perfiles de tiempo real

El mayor de los perfiles, denominado "Sistema de Tiempo Real Multipropósito" (PSE54), dispone de comunicaciones por red, sistema de ficheros jerárquicos en disco, terminales gráficos de interfaz con el usuario, etc. Se trata de una estación de trabajo con características de tiempo real.

En segundo lugar se encuentran el "Sistema de Tiempo Real Dedicado" (PSE53), un sistema empotrado grande, con un simple sistema de ficheros, capaz de ejecutar una aplicación empotrada compleja que requiera múltiples procesos y comunicaciones por red.

El siguiente perfil corresponde al "Controlador de Tiempo Real" (PSE52), orientado a controladores de propósito muy específico, no contiene unidad de manejo de memoria, pero si un sistema de ficheros simplificado.

Por último, el más sencillo, el "Sistema de Tiempo Real Mínimo" (PSE51), que está pensado para aplicaciones empotradas sencillas. No contiene soporte para múltiples procesos, ni sistema de ficheros jerárquico, con lo que se elimina la parte más compleja del sistema POSIX, y permite implementar un núcleo de sistema operativo pequeño y eficiente.

Figura 1.2. Funcionalidades del perfil mínimo

1.3. Motivaciones y antecedentes

Dado que el factor más crítico en un sistema de tiempo real es el intervalo transcurrido desde que ocurre un evento hasta que se realiza una acción como respuesta, resulta de vital importancia conocer perfectamente su capacidad de reacción y su comportamiento temporal.

El sistema operativo es uno de los elementos que más restringe la capacidad de reacción del sistema global, ya que es quien se encarga de gestionar los recursos de los que dispone el sistema. Por este motivo, es importante caracterizar los servicios que ofrece como forma de analizar el funcionamiento temporal del sistema global.

La caracterización temporal del sistema operativo es de importante utilidad en dos fases muy diferenciadas. Por un lado, constituye una información que facilita el análisis y la evaluación del sistema operativo en su fase de desarrollo, lo que permite comprobar el funcionamiento de los servicios que realiza, así como el rendimiento que presenta para diferentes implementaciones, y detectar posibles "puntos débiles" en el diseño. Por otro lado, esa evaluación del funcionamiento y del rendimiento facilita al usuario un conjunto de datos con los que puede realizar comparaciones entre diferentes sistemas operativos que ofrezca el mercado, o simplemente comprobar si uno en concreto cumple los requisitos temporales necesarios para el sistema en que va a ser implementado. En tercer lugar, facilita el análisis del tiempo de respuesta de las aplicaciones que corren bajo ese determinado sistema.

Durante la década de los ochenta y principios de los noventa, los miembros de PIWG, "*Performance Issues Working Group*", crearon y distribuyeron un grupo de

pruebas, conocido como el *"PIWG Test Suite"*, que consistía en más de 200 archivos, en los que se incluyen pruebas de evaluación de la velocidad del procesador (*Whetstone*), medida de ejecución de sentencias de código por unidad de tiempo (*Dhrystone*), así como otras pruebas sobre varios atributos del lenguaje Ada y sus implementaciones bajo compiladores específicos [PIGW].

Este conjunto de pruebas estaba destinado a Ada 83, y a medida que se fue extendiendo el uso de Ada 95 este grupo de trabajo decidió definir un nuevo conjunto de pruebas de evaluación, que dio lugar al ACES, *"Ada Compiler Evaluation Systems"*.

ACES permite realizar pruebas de rendimiento, análisis de la gestión de software, así como determinar el funcionamiento de los sistemas de compilación y ejecución de Ada. También proporciona herramientas para examinar el sistema de diagnóstico de la implementación, el sistema de gestión de librerías, y un depurador simbólico, así como para medir el tiempo de compilación y de ejecución de la implementación [ACES].

Este grupo de pruebas reúne las mejores características de otros dos sistemas de evaluación: el ACEC, *"Ada Compiler Evaluation Capability"* y el AES, *"Ada Evaluation System"*, está orientada a Ada 95, y está compuesta de aproximadamente 2120 pruebas [ACES].

Hay seis grupos de pruebas predefinidas para analizar el funcionamiento de las siguientes operaciones [ACES]:

- Concurrencia
- Operaciones de coma flotante
- Operaciones con enteros
- Operaciones de punto fijo
- Operaciones con caracteres y cadenas
- Cláusulas de representación y atributos

Los análisis que permite realizar este sistema son:

- Comparar el rendimiento de varias implementaciones.
- Aislar los puntos débiles de un determinado sistema comparándolo con otros ya probados.
- Determinar cambios significativos entre versiones de sistemas compilados.
- Predecir el rendimiento de diferentes formatos de código.
- Determinar si un depurador simbólico soporta unas determinadas características funcionales.
- Determinar si una librería de un programa soporta unas determinadas características funcionales.
- Determinar el tiempo de compilación y ejecución de un determinado sistema de compilación.
- Evaluar la claridad y exactitud de los mensajes de diagnóstico del sistema.

Estas baterías de pruebas están orientadas al lenguaje Ada, y la evaluación del rendimiento de sistemas y compiladores bajo circunstancias muy concretas. Sin embargo no constituyen una herramienta válida para evaluar los servicios descritos en los perfiles POSIX mostrados en el apartado anterior. El propósito del presente trabajo, tal y como se irá exponiendo a lo largo de esta memoria, es la realización de un conjunto de programas que permitan evaluar la respuesta de los servicios que conforman el último de los perfiles del estándar, "Sistema de Tiempo Real Mínimo". Estos servicios serán descritos previamente en el capitulo siguiente para su posterior caracterización y análisis.

La finalidad de este conjunto de pruebas es, por una lado, ofrecer al diseñador del sistema una herramienta que le permita evaluar las respuestas temporales de los servicios que lo componen, permitiendo detectar puntos críticos o problemas de rendimiento, y por otro lado, proporcionar un mecanismo de análisis al posible comprador de un sistema basado en el estándar POSIX que facilite la comparación de varios sistemas disponibles en el mercado, o simplemente determinar si uno en concreto se adecua a un conjunto de necesidades específicas requeridas.

Es preciso indicar que las medidas obtenidas no permiten determinar con garantía absoluta los tiempo de respuesta de peor caso, ya que pueden darse en la práctica situaciones peores a las dadas observadas. La obtención de valores garantizados de tiempo de respuesta de peor caso es muy compleja, y requiere del uso de técnicas especiales [WCET].

1.4. Objetivos del trabajo

El último de los perfiles POSIX presentados en el apartado 1.2, "Sistema de Tiempo Real Mínimo", constituye el objeto de estudio de la presente memoria. Este estudio abarca tanto la caracterización teórica, composición de servicios y funcionamiento, así como el diseño y realización de un conjunto de programas de prueba, que permiten evaluar la respuesta temporal de algunos de los servicios que componen dicho perfil. Además, se aplicará esta herramienta para realizar la caracterización temporal de un sistema operativo concreto denominado MaRTE OS y compararla con la de un sistema de tiempo compartido, no de tiempo real, basado en Linux.

En primer lugar se realizará un análisis teórico de dicho perfil, en el que serán estudiadas las características generales de este tipo de sistemas, así como de los servicios que estos ofrecen. Este estudio es necesario para la etapa siguiente del proyecto, en la que se llevará a cabo el diseño de un conjunto de herramientas que permiten caracterizar el comportamiento temporal de los servicios de los sistemas operativos de tiempo real mínimo.

Esta herramienta de análisis consiste en una serie de experimentos específicos para cada uno de los servicios:

- Mecanismos de sincronización: Para analizar este servicios se realizarán dos experimentos. El primero de ellos caracterizará temporalmente las operaciones que pueden realizarse con los mutexes, mientras que en el segundo se realizará un estudio sobre las variables condicionales.

- Gestión del tiempo: Debido a que un sistema de tiempo real, las restricciones impuestas se refieren siempre a un plazo de tiempo limitado, es muy importante realizar un estudio de los mecanismos de gestión de tiempo que este implementa. Por este motivo, se realizará un experimento que permita determinar tanto la resolución del reloj del sistema, como la de los mecanismos de suspensión (*sleep*). Así mismo, serán objeto de estudio los temporizadores, y los retrasos (*delays*) entre threads de diferentes prioridades.

- Señales de tiempo real: En este experimento se implementarán diferentes operaciones que proporcionan las señales de tiempo real, con objeto de caracterizar temporalmente su funcionamiento.

- Gestión de memoria: La gestión de memoria en tiempo real se realiza mediante dos funciones: *malloc()* y *free()*. De modo que se llevará a cabo un experimento que permita determinar el comportamiento de dichas funciones, con diferentes tamaños de bloques de memoria.

- Interrupciones: Las interrupciones causan una sobrecarga que puede llegar a ser crítica en la estabilidad del sistema. Se realizará un experimento que tendrá como fin determinar la frecuencia de interrupciones en el sistema, así como la duración de cada una de ellas.

En último lugar, se procederá a probar la batería de experimentos diseñada para caracterizar un sistema operativo. Se ha seleccionado el sistema operativo MaRTE OS, ya que ha sido creado siguiendo los requisitos y especificaciones indicados en el perfil mínimo definido en el estándar POSIX. Para ello se hace necesario realizar un estudio previo de este entorno. Para finalizar, se lleva a cabo la caracterización de un sistema Linux, diseñado según especificaciones de tiempo compartido y que no cumple los requisitos de un sistema de tiempo real, para poder llevar a cabo una comparación entre ambos, que ilustre las diferencias entre ambos tipos de sistemas.

1.5. Estructura de la memoria

La presente memoria puede dividirse claramente en tres partes fundamentales. En un primer lugar encontramos una parte teórica, constituida por los dos primeros capítulos, en la cual se realiza un estudio básico del estándar POSIX (contenido en esta introducción), y de un sistema operativo de tiempo real mínimo, con el fin de proceder en una siguiente parte a su caracterización.

La segunda parte de la memoria detalla el diseño de un conjunto de herramientas cuya funcionalidad reside en la caracterización temporal de dicho tipo de sistema, procediendo para ello al análisis temporal de los servicios que ofrece. Este diseño se especifica en los capítulos tercero y cuarto.

La tercera y última parte abarca los resultados recogidos de las pruebas realizadas con la aplicación que se ha diseñado, para dos entornos diferentes, MaRTE OS y Linux, así como la evaluación y comparación de estos resultados.

Finalmente, en el último capítulo se extraen las conclusiones obtenidas tras la realización del proyecto, y se proponen algunas líneas futuras de desarrollo.

2.- Sistemas empotrados de tiempo real

2. Sistemas empotrados de tiempo real

2.1. Sistemas POSIX de Tiempo Real Mínimos (PS51)

Los sistemas de tiempo real mínimos habitualmente son sistemas empotrados en otros sistemas mayores, en el que realizan una o varias funciones específicas. No requieren interacción con el usuario, ni un sistema de ficheros.

El modelo de programación consiste en un único proceso (que corresponde con el espacio de direcciones del procesador hardware) que contiene uno o varios threads de control.

El modelo hardware para este perfil asume un único procesador con su memoria correspondiente, pero no incluye unidad de gestión de memoria, lo que impide en estos sistemas la implementación de procesos con espacios de direcciones separados [1003.13].

Los AEP (*Application Environment Support*) de POSIX de Tiempo Real describen los entornos de aplicación necesarios para la construcción y ejecución de programas de aplicación en tiempo real portables.

El perfil de entorno de aplicación de un sistema de tiempo real mínimo se define en el estándar PSE51. En él se especifican los servicios necesarios que han de cumplir estos sistemas para conseguir la conformidad correspondiente con el estándar.

Los creadores de este estándar han intentado capturar las funcionalidades predominantes en los sistemas de tiempo real ya existentes, y consideraron características de los núcleos de varios sistemas comerciales. Tras este análisis, concluyeron que el modelo más extendido en ciertas áreas de la industria de sistemas empotrados era el de threads definido en POSIX.1. Así mismo, se consideró suficiente el dispositivo básico de entrada salida (*read, write, open, close, control*) en vez de un sistema completo de ficheros. El soporte multiproceso fue descartado [1003.13].

El conjunto de requerimientos exigidos por el estándar para la definición de este perfil de entorno de aplicación será enumerado a continuación. Por un lado, encontramos los requisitos correspondientes a los servicios básicos del sistema, por otro lado los correspondientes a los servicios propios de la extensión de tiempo real, y por último, los requerimientos relativos a threads que serán tratados en el apartado siguiente.

Primitivas de proceso: Dado que este perfil emplea el modelo de threads definido en POSIX.1 para proporcionar un mecanismo de concurrencia, la mayoría de las primitivas de proceso POSIX.1 no son soportadas, ya que se emplea un único proceso. Los servicios de señalización son necesarios para el manejo de errores y eventos.

Entorno de proceso: Las funciones *sysconfig()* y *uname(),* definidas en POSIX.1, son necesarias para permitir que una determinada aplicación pueda determinar el entorno de sistema.

Ficheros y directorios: La función *open()* es necesaria en las operaciones básicas de entrada – salida, así como para la inicialización del dispositivo. Aunque se hace necesario algún tipo de resolución de nombres, no se requiere específicamente un espacio de ruta completa, ni de directorios.

Primitivas de entrada – salida: Para las operaciones básicas de entrada – salida se especifican las funciones *read(), write()* y *close().*
La entrada – salida asíncrona no es necesaria ya que puede ser fácilmente implementada mediante threads dedicados a la entrada salida.

Base de datos del sistema, usuarios y grupos: No son necesarias implementaciones para más de un usuario, ni identificador de grupo, debido a que no hay múltiples usuarios ni grupos. Tampoco se requieren funciones para la base de datos del sistema.

Funciones de dispositivo y clase: Las funciones específicas de dispositivo y clase definidas en el estándar POSIX.1 no son necesarias, debido a que los sistemas empotrados pequeños habitualmente no requieren interfaces para terminar de propósito general.

Señales de tiempo real: Las señales son un mecanismo básico en los sistemas basados en el estándar POSIX, y son imprescindibles para el manejo de errores y eventos. Se asume que los sistemas de tiempo real ejecutarán varios elementos software concurrentemente. Cada entidad deberá responder a ciertos estímulos, periódicos o esporádicos, a menudo en un intervalo de tiempo crítico. Aunque los modelos puramente síncronos pueden suplir dicha funcionalidad mediante procesos y threads, en la práctica el tiempo real ofrece gran rendimiento y baja latencia para notificaciones asíncronas de eventos por exceso de tiempo, llegada de mensajes, o interrupciones de hardware. Las señales de tiempo real proporcionan el mecanismo de alto rendimiento y fiable para dar soporte a dichas notificaciones.

El mínimo número de señales de tiempo real que la implementación ha de soportar es 16, ya que hay muchas aplicaciones en las que se distinguen más de 8 tipos diferentes de eventos.

Entrada – Salida sincronizada: Se definen también funciones para que las operaciones de entrada y salida se realicen de forma sincronizada. Una

escritura sincronizada es aquella que se completa cuando todos los datos han sido transferidos correctamente, así como la información del sistema de ficheros correspondiente, para que dicha información pueda ser recuperada. La interfaz de entrada – salida sin almacenamiento intermedio es básica en los sistemas mínimos.

Sincronización: Los mutexes y las variables condicionales se definen como parte del modelo de concurrencia proporcionado mediante threads, ya que consituyen el mecanismo por el que pueden cooperar.

Por compatibilidad con sistemas anteriores actualmente en uso en la industria, se mantiene dentro del perfil el mecanismo de sincronización que implementan los semáforos, aunque actualmente la tendencia es emplear tanto mutexes como variables condicionales. Ha de tenerse en cuenta que los semáforos POSIX no incorporan ningún mecanismo capaz de acotar el efecto de inversión de prioridad en casos de acceso mutuamente exclusivo a recursos compartidos. Sin embargo, para evitar este fenómeno, puede recurrirse a mutexes con un protocolo correcto de herencia de prioridad o de protección de prioridad.

Planificación por prioridad: Las aplicaciones de tiempo real hacen imprescindible una planificación por prioridad para threads. Así mismo, la planificación de servidor esporádico es necesaria para mejorar el soporte de aplicaciones con requisitos temporales aperiódicos.

Bloqueo de memoria de proceso: El bloqueo de memoria del proceso es inherente en los sistemas que siguen este perfil, y es necesario para lograr portabilidad.

Memoria compartida: Puede implementarse el mapeo de memoria entrada – salida empleando la funcionalidad de memoria compartida. Dicha implementación debe facilitar la creación de objetos de memoria compartida que representen rangos de memoria física que dispongan un dispositivo de control, y registros de estado o buffers. Estas facilidades fortalecen el desarrollo de aplicaciones portables.

Relojes y temporizadores: Los temporizadores de alta resolución se emplean en la mayoría de los sistemas de tiempo real para implementar operaciones de gestión de tiempo, como por ejemplo las activaciones periódicas. Las funciones definidas en el estándar POSIX.1, *sleep()* y *alarm()* tienen una resolución de un segundo, por lo que se hace necesario esta funcionalidad para sistemas que requieran una mayor precisión.

Threads: En este perfil se asume que el sistema consiste en un único proceso con múltiples thread, de modo que se requieren todos los servicios básicos para thread, excluyendo aquellos que sean específicos para múltiples procesos.

2.2. Definición de servicios objeto de caracterización

Tras realizar el repaso del estándar POSIX, con las especificaciones y recomendaciones correspondientes para el perfil de sistema de tiempo real mínimo que nos ocupa, se determinan los servicios que van a ser objeto de la caracterización temporal realizada en el presente proyecto. A continuación pasan a describirse brevemente estos servicios, antes de proceder en el siguiente capítulo a describir el proceso llevado a cabo para realizar dicha caracterización.

2.2.1. Threads

Previo paso a la descripción de los servicios y mecanismos que se describen en el perfil de sistema mínimo de tiempo real, se hace necesaria una breve definición y explicación del funcionamiento de los threads, ya que es el mecanismo de concurrencia en estos sistemas.

Un thread, también llamado *hilo*, o *proceso ligero*, es una unidad básica de utilización de recursos que consiste en un contador de programa, un conjunto de registros y un espacio de pila. Un thread comparte con otros threads una sección de código, una sección de datos y algunos recursos del sistema operativo. De esta forma se comparten los recursos y las localizaciones de memoria manteniendo distintos flujos de control en un mismo proceso.

Para crear un thread, es necesario primero definir sus atributos en un objeto especialmente definido para ello. Este objeto de atributos ha de crearse antes de usarlo, y permite modificar o consultar atributos concretos, aunque lo parámetros del thread no pueden modificarse una vez haya sido creado. Los atributos definidos son:

- Tamaño de stack mínimo: la especificación de este parámetro es opcional.
- Dirección del stack: este parámetro es opcional también, y suele ser peligroso tratar de manejarlo, ya que no conocemos el mapa de memoria.
- Control de devolución de recursos al sistema tras su finalización.

Además, hay otros atributos relacionados con la planificación que se verán más adelante. Los threads tienen dos estados posibles para controlar la devolución de recursos al sistema cuando termina su ejecución [GON01a]:

- Independiente (“*detached*”): cuando el thread termina, devuelve al sistema automáticamente todos los recursos, pero no podemos esperar mediante sincronización a que este thread termine.
- Sincronizado (“*joinable*”): cuando el thread termina su ejecución, mantiene sus recursos hasta que otro thread hace la llamada *pthread_join()*. Si el otro thread hace esta llamada sin que el primero haya terminado, se quedará esperando su finalización.

Podemos hacer que un thread de estado sincronizado pase a ser independiente, pero no a la inversa.

Cuando el thread termina se ejecutan las rutinas de cancelación correspondientes. Una vez que ha terminado, ya no se puede acceder a sus variables locales.

Planificación de threads

Un thread, una vez creado, y antes de terminar su ejecución puede estar en tres estados diferentes:

1. En ejecución: si está siendo ejecutado por un procesador.
2. Activo: si está listo para ser ejecutado, pero está en espera de que un procesador esté libre.
3. Bloqueado o suspendido: si está esperando una condición para poder continuar ejecutando.

Figura 2.1 Estados de un thread

La prioridad de un thread viene determinada por un número entero positivo asociado, y es empleada por la política de planificación. Una política de planificación es un conjunto de reglas que determinan en que orden han de ejecutarse los threads. Esta política es la que decide en cada momento el estado de los threads, y en función a ella estos se suspenden, activan, son expulsados o llaman a una función que cambia la prioridad o la política [GON01a].

Las políticas de planificación definidas en el estándar POSIX .1 para threads son estas [GON01a]:

- SCHED_FIFO: esta política de planificación es expulsora por orden de prioridad, y FIFO (*"First in first out"*) para threads con el mismo orden de prioridad. Está es la política recomendada en sistemas de tiempo real estricto.
- SCHED_RR: esta política de planificación también es expulsora según el orden de prioridad. Para un mismo orden de prioridad sigue un orden rotatorio (*"Round Robin"*), con un intervalo definido fijo.

- SCHED_SPORADIC: esta política de planificación de servidor esporádico es necesaria en aplicaciones de tiempo real para mejorar el soporte de aplicaciones que tengan requisitos temporales aperiódicos. Se basa en dos principalmente en dos parámetros: el periodo de relleno y la capacidad de ejecución disponible. Garantiza y limita la anchura de banda al gasto de un tiempo de CPU igual a la capacidad de procesado, por cada periodo de relleno
- SCHED_OTHER: esta política de planificación ha de ser definida por la implementación.

Uno de los requisitos comunes en los sistemas de tiempo real es que han de ser capaces de ejecutar threads con requisitos de tiempo real junto con threads sin estos requisitos. Una buena forma de conseguir esta planificación es asignar la política SCHED_FIFO a los threads con requisitos de tiempo real, y la política SCHED_RR a los threads sin estos requisitos con unas prioridades menores, para repartir equitativamente entre ellos el tiempo del procesador. Así mismo, se asignaría la política SCHED_SPORADIC a los threads aperiódicos con requisitos temporales.

Así mismo, el estándar requiere que haya como mínimo 32 niveles diferentes de prioridad, de forma que sea posible asegurar altos niveles de utilización incluso cuando el número de threads sea elevado [GODOU].

Los atributos de prioridad y política de planificación forman parte del objeto de atributos de cada thread, y es posible modificarlos o consultarlos dinámicamente incluso después de que el thread haya sido creado [GODOU].

Los atributos de planificación definidos en el estándar y que son útiles en aplicaciones con un solo proceso, son las siguientes:

- Política de planificación (*schedpolicy*): como se ha indicado en los párrafos superiores, para tiempo real se recomienda FIFO (SCHED_FIFO), Round Robin (SCHED_RR) o Servidor Esporádico (SCHED_SPORADIC).

- Parámetros de planificación (*scheparam*): es una estructura de parámetros en la que se definen los valores característicos de la planificación: *sched_priority.* En el caso de que hayan sido definida la política por Servidor Esporádico, está estructura ha de contener además los siguientes campos:
  - sched_ss_low_priority: entero que fija el umbral de mínima prioridad.
  - sched_ss_repl_period: estructura timespec que indica el tiempo de relleno.
  - sched_ss_init_budget: estructura timespec que indica la capacidad inicial.
  - sched_ss_max_repl: entero que indica el máximo de rellenos pendientes.

- Herencia de atributos de planificación (*inheritsched*): mediante este parámetro se especifica si el thread que se va a crear ha de heredar los atributos del thread padre (PTHREAD_INHERIT_SCHED) o si por el contrario son los definidos en el objeto de atributos de thread, y por lo tanto diferentes de los del padre (PTHREAD_EXPLICIT_SCHED).

2.2.2. Sincronización entre threads

Como ya fue indicado en el apartado anterior, los semáforos han sido descartados como mecanismo para realizar la sincronización entre threads, y actualmente la tendencia es emplear mutexes y variables condicionales. Por este motivo, serán estos dos servicios los estudiados para desempeñar esta función.

Un mutex es un objeto de sincronización que permite que múltiples threads puedan acceder de forma mutuamente exclusiva a un recurso de memoria compartida. Permite realizar operaciones de bloqueo y liberación de este recurso. El modo en que el que operan puede describirse sencillamente: si el mutex está libre, el thread correspondiente lo toma y se convierte en su propietario. Si está ocupado, el thread se suspende y se añade a una cola en espera de poder acceder a él. Cuando el mutex se libera, si hay varios threads esperando en la cola, se le asigna el mutex al primer elemento de esas cola, que se convierte en el nuevo propietario. Solo este thread puede liberarlo una vez termine de usarlo, para que otros threads puedan acceder a él [GON01a].

El perfil de sistema de tiempo real mínimo requiere que se de soporte tanto al protocolo de herencia de prioridad como al de protección de prioridad para los mutex. La inversión de prioridad ocurre cuando un thread de prioridad alta tiene que esperar a que un thread de baja prioridad termine de hacer uno de un recurso determinado. Estos protocolos evitan el efecto de inversión de prioridad no acotada, que produce largos retrasos en el tiempo de respuesta de threads de muy alta prioridad [GODOU].

Los atributos de creación del mutex que son útiles en aplicaciones de un solo proceso son [GODOU]:

- Protocolo *(protocol)*: puede ser uno de los siguientes:
  - PTHREAD_NONE, para sistemas sin requisitos de tiempo real.
  - PTHREAD_PRIO_INHERITANCE, para los protocolos de herencia de prioridad.
  - PTHREAD_PRIO_PROTECT, para los protocolos con protección de prioridad.

- Techo de prioridad *(priority ceiling)*: en el caso concreto del protocolo de protección de prioridad, el techo de prioridad es el valor de prioridad heredado por el propietario del mutex. Este valor puede ser cambiado dinámicamente después de que el mutex haya sido creado.

Una variable condicional también actúa como herramienta de sincronización. Permite que un thread se suspenda hasta que otro thread lo reactive haciendo que se cumpla una condición lógica. Mientras esta condición no se cumpla, el thread suspendido seguirá a la espera para ser reactivado. También permite realizar operaciones de broadcast con una condición, de forma que se reactiven todos los threads que estén suspendidos en espera de esa condición [GON01a].

Las variables condicionales se usan siempre con un mutex que se emplea para proteger la evaluación del predicado lógico que actúa como condición para decidir si el thread ha de esperar o no. La operación de espera se realiza con el mutex bloqueado por el thread que invoca la operación, y mientras éste queda suspendido en espera, el mutex se libera para que el thread correspondiente pueda realizar las modificaciones necesarias en las variables protegidas, y por tanto modificar el resultado del predicado lógico. Cuando la variable condicional se señaliza, el thread que estaba esperando, toma el mutex y evalúa de nuevo el predicado lógico.

2.2.3. Gestión del tiempo

Un reloj es un objeto que mide el paso del tiempo. Su resolución es el intervalo de tiempo más pequeño que dicho objeto es capaz de discriminar. Requisitos generales en los sistemas de tiempo real son poder medir tiempos y realizar acciones determinadas, ya sea periódicamente, en un momento determinado, o al cabo de un intervalo de tiempo. Para ello se definen los siguientes servicios en el estándar POSIX.1b.

- **Relojes**: En los sistemas de tiempo real mínimo podemos hacer uso de tres relojes:

  - Reloj del sistema: este reloj mide los segundos transcurridos desde la Época. Se emplea para registrar el momento de creación de ficheros, etc. Es global al Sistema.

  - Reloj de tiempo real: este reloj también mide los segundos y nanosegundos transcurridos desde la Época. No coincide necesariamente con el reloj del sistema y se emplea para calcular los timeouts y para crear temporizadores. También es global al sistema, y tiene una resolución mucho más precisa que la anterior, mínima 1 nanosegundo, máxima 20 milisegundos.
    Para definir el tiempo con alta resolución, se define el tipo timespec, que sigue esta estructura:

```
struct timespec {
     time_t tv_sec; /*segundos*/
     long tv_nsec; /*nanosegundos*/
}
```

    De forma que el tiempo en nanosegundos queda expresado como:

    $$tiempo = tv\_sec * 10^9 + tv\_nsec$$

    La Época corresponde a las 0 horas, 0 minutos, 0 segundos del día 1 de Enero de 1970.

  - Reloj monotónico: es necesario en aplicaciones de tiempo real para asegurar que los saltos del reloj no afectan a los plazos límite y los requerimientos temporales impuestos.

- **Temporizadores**: Un temporizador es un objeto que permite notificar a un proceso u otro thread si ha transcurrido un determinado intervalo de tiempo (en modo relativo), o se ha alcanzado una hora determinada (en modo absoluto). Es necesario asociarlo a uno de los dos tipos de relojes definidos.

- **Retrasos de alta resolución:** Además de caracterizar estos dos servicios, será tomado en cuenta también el intervalo de tiempo que tarda en producirse el cambio de contexto entre threads de diferentes prioridades a la hora de realizar retrasos de alta resolución, con las funciones *nanosleep y clock_nanosleep*.

2.2.4. Señales de tiempo real

Las señales proporcionan un mecanismo mediante el cual se puede informar a un determinado proceso de un evento que se produce en el sistema. Estos eventos pueden ser excepciones detectadas por hardware, finalización de operaciones de entrada – salida, llegada de mensajes, alarmas y expiración de temporizadores, etc.

Se distinguen dos tipos de señales: las no fiables, y las fiables, o de tiempo real. Las primeras no son exclusivas de los sistemas de tiempo real, y se identifican mediante números enteros, o mediante constantes predefinidas que dan nombre a las señales. Al no ser fiables se pueden perder si ya hay una señal del mismo número pendiente.

En cambio, las señales de tiempo real no se pierden, sino que se encolan si están pendientes ya que se aceptan por orden de prioridad (la prioridad viene determinada por el número de la señal). Existen un mínimo de ocho señales, y están comprendidos entre los números SIGRTMIN y SIGRTMAX. Además contienen un campo adicional de información en el que proporcionan datos del evento que notifican.

Una señal está en estado “pendiente” desde que se crea hasta que es aceptada. Sólo podemos observar su estado, cuando está bloqueada, o enmascarada. Cuando ocurre un nuevo evento de una señal que está en este estado:

- Si se trata de una señal “no fiable”, dicho evento puede perderse.
- Si se trata de una señal de tiempo real, se encola, y se pone a la espera.

Hay tres tipos de acciones que se pueden asociar a una señal: la acción por defecto habitualmente consiste en terminar el proceso, aunque también puede ignorarse la señal. La tercera opción consiste en capturar la señal y ejecutar un manejador. Este manejador se señal se ejecuta cuando se entrega la señal, y cuando finaliza se continúa el proceso desde el lugar en que había sido interrumpido.

En la figura que se expone a continuación se detallan los estados en que pueden encontrarse las señales, así como las opciones que se ofrecen:

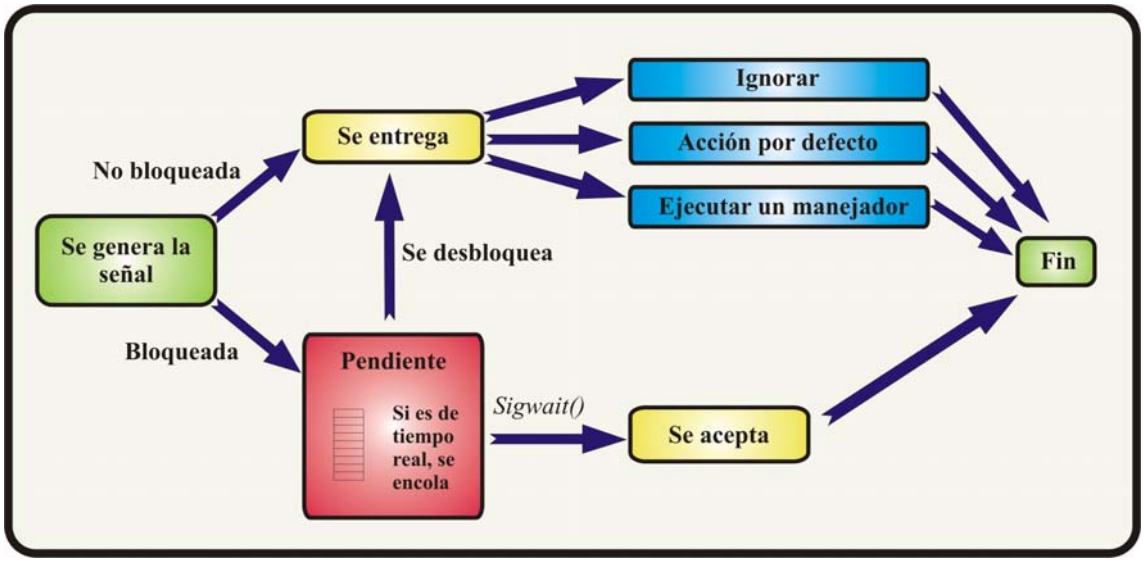

Figura 2.2. Diagrama de funcionamiento de señales de tiempo real

En esta figura se puede observar las diferentes opciones que se presentan para el manejo de señales de tiempo real. En aplicaciones de tiempo real, la opción recomendada consiste en aceptar la señal y ejecutar el manejador correspondiente, para luego proseguir la ejecución normal del proceso en el punto en el que se había suspendido.

2.2.5. Gestión de memoria dinámica

La gestión de memoria en tiempo real se realiza mediante dos funciones: *malloc()* y *free()*. La función *malloc()* permite reservar dinámicamente un bloque de memoria del tamaño suministrado como parámetro. Devuelve un puntero al bloque de memoria del tamaño solicitado. Por el contrario, la función *free()* libera dicho espacio de memoria.

2.2.6. Sobrecarga producida por interrupciones

La sobrecarga que se produce debida a las interrupciones en un sistema operativo de tiempo real puede llegar a ser un factor crítico en su estabilidad. Cuando se produce un estado de sobrecarga, o se produce una interrupción, el sistema ha de mantenerse estable, y para ello ha de cumplir los intervalos requeridos al menos para las tareas más críticas. Por este motivo no solo es importante caracterizar el tiempo que tardan ejecutarse los servicios que ofrece el sistema, sino que es necesario conocer el periodo de tiempo que el sistema deja de "estar disponible" para la ejecución de las tareas, así como la frecuencia con la que esta falta de disponibilidad ocurre.

2.3. Relación de servicios caracterizados

Se han elegido para su caracterización los servicios usuales en aplicaciones de tiempo real de acuerdo con la experiencia del grupo de Computadores y Tiempo Real de la Universidad de Cantabria en el desarrollo de controladores de robots industriales.

2.3.1. Mutex

En total se realizan un total de dieciocho medidas diferentes analizando los distintos aspectos del mecanismo de sincronización, atendiendo a los parámetros de configuración indicados al principio del apartado. Esta es la lista de medidas realizadas:

Independientemente del protocolo:

Inicialización de una variable mutex: ***pthread_mutex_init()***
Fijación del techo de prioridad: ***pthread_mutex_setprioceiling()***
Obtención del techo de prioridad: ***pthread_mutex_getprioceiling()***
Destrucción de una variable mutex: ***pthread_mutex_destroy()***

Para los protocolos PTHREAD_PRIO_INHERIT y PTHREAD_PRIO_PROTECT:

Bloqueo de un mutex: ***pthread_mutex_lock()***
Bloqueo condicional de un mutex: ***pthread_mutex_trylock()***

La diferencia entre ambas llamadas es que en la primera (*lock()*) si el mutex referenciado está ya bloqueado la función se suspende hasta que pueda tomarse la variable. Por el contrario, en la segunda (*trylock()*) la función retorna automáticamente, en ese caso. Para ambas se mide el tiempo de bloqueo cuando el mutex está libre.

Desbloqueo de un mutex: ***pthread_mutex_unlock()***

En un thread de alta prioridad:
- Cuando no hay ningún thread esperando acceso al mutex.

En un thread de baja prioridad + activación del otro thread de mayor prioridad que está esperando el mutex:
- Cuando hay un thread esperando acceso al mutex.
- Cuando hay “th_wait” threads esperando acceso al mutex.

“th_wait” es un parámetro configurable que fija el número de threads que esperan el acceso al mutex.

2.3.2. Variables Condicionales

En total se realizan 11 medidas, tal y como se enumera a continuación en una lista simplificada de las operaciones analizadas:

Inicialización de una variable condicional: ***pthread_cond_init()***
Destrucción de una variable condicional: ***pthread_cond_destroy()***

Señalización: Envío de una señal para desbloquear un thread que hay sido bloqueado por una variable condicional.

pthread_cond_signal() desbloquea al menos uno de los threads que haya sido bloqueado por la variable condicional. Se evalúa:

pthread_cond_signal() (con un thread esperando)
pthread_cond_signal() (con **"th_wait"** threads esperando)
pthread_cond_signal() + activación del thread que espera (entre dos threads)
pthread_cond_signal() + activación del thread que espera (con **"th_wait"** threads esperando)

pthread_cond_broadcast() desbloquea todos los threads que hayan sido bloqueados por esa variable. Se evalúa:

pthread_cond_broadcast() (con un thread esperando)
pthread_cond_broadcast() (con **"th_wait"** threads esperando)
pthread_cond_broadcast() + activación del thread que espera (entre dos threads)
pthread_cond_broadcast() + activación del thread que espera (con **"th_wait"** esperando)

Espera limitada a una variable condicional: ***pthread_cond_timedwait().***
Se evalúa el caso en el se sobrepasa el intervalo de espera.

"th_wait" es un parámetro configurable que fija el número de threads que esperan el acceso al mutex.

2.3.3. Gestión del tiempo

En este experimento se realizan 33 medidas. Este total de medidas incluye la resolución del reloj, el análisis de las funciones de suspensión de threads y los temporizadores. La lista detallada se enumera a continuación:

Resolución del reloj: ***clock_getres()*** para los relojes:
CLOCK_REALTIME
CLOCK_MONOTONIC

Retraso de alta resolución. Se evalúa el tiempo real de suspensión para:

***nanosleep()*:** con cinco intervalos, que abarcan desde valores extremadamente pequeños, hasta valores habituales en sistemas de Tiempo Real:

1 μsg
10 μsg
100 μsg
1000 μsg
10000 μsg

clock_nanosleep() en modo absoluto para los relojes:

CLOCK_REALTIME
CLOCK_MONOTONIC

Con cinco intervalos escogidos de forma que se de tiempo suficiente al experimento para volver a su estado inicial antes de la siguiente realización:

Hora inicial + 0.1 sg + 0 μsg
Hora inicial + 0.2 sg + 10 μsg
Hora inicial + 0.3 sg + 100 μsg
Hora inicial + 0.4 sg + 1000 μsg
Hora inicial + 0.5 sg + 10000 μsg

Activación de un temporizador en modo simple relativo: ***timer_settime()*** + activación del thread tras la expiración: para los relojes:

CLOCK_REALTIME
CLOCK_MONOTONIC

Con los mismos intervalos que los indicados para *nanosleep()*.

Intercambio de ejecución de threads con un *"delay"* de alta prioridad: mide la ejecución del delay más la activación de un thread de menor prioridad:

nanosleep()
clock_nanosleep(): CLOCK_REALTIME, CLOCK_MONOTONIC

Intercambio de ejecución de threads con un *"delay"* de baja prioridad: mide el tiempo que el thread de baja prioridad no puede leer el reloj. Funciones:

nanosleep()
clock_nanosleep(): CLOCK_REALTIME, CLOCK_MONOTONIC

2.3.4. Señales de tiempo real

En este experimento se realizan un total de ocho medidas con objeto de caracterizar el servicio de las señales de tiempo real. Para ello han sido analizadas las siguientes operaciones:

Envío de una señal a un thread:

pthread_kill() (con un thread esperando con ***sigwait()***)

pthread_kill() (con el número de threads indicado en "th_wait" esperando con ***sigwait()*).**
pthread_kill() + tiempo de activación del thread que recibe la señal (con un solo thread esperando con ***sigwait()***).
pthread_kill() + tiempo de activación del thread que recibe la señal (con el número de threads indicado en "th_wait" esperando con ***sigwait()***).

Envío de una señal de tiempo real a un thread. La diferencia con la llamada anterior es que, en el primer caso si la señal estaba bloqueada se pierde, y en este caso se encola:

sigqueue() (con un solo thread esperando con ***sigwaitinfo()***)
sigqueue() (con el número de threads indicado en "th_wait" esperando con ***sigwaitinfo()***).
sigqueue() + tiempo de activación del thread que recibe la señal (con un solo thread esperando con ***sigwaitinfo()***).
sigqueue() + tiempo de activación del thread que recibe la señal (con el número de threads indicado en "th_wait" esperando con ***sigwaitinfo()***)

Las llamadas de espera ***sigwait()*** y ***sigwaitinfo()*** proceden de la misma forma, con la diferencia de que la segunda almacena información sobre la señal.

"th_wait" es un parámetro configurable que fija el número de threads que esperan el acceso al mutex.

2.3.5. Gestión de memoria

Las medidas que se toman durante en el proceso, corresponden a la operación de reserva (*malloc()*) y de liberación (*free()*) para cada uno de los tamaños de bloques especificados, tal y como se resume en la siguiente lista:

En el experimento: **pta_experiment_memory**

	Tamaño del bloque en bytes		
malloc()	**100**	**1000**	**10000**
free()	**100**	**1000**	**10000**

En el experimento: **pta_experiment_memory_x2**

	Tamaño del bloque en bytes		
malloc()	**200**	**2000**	**20000**
free()	**200**	**2000**	**20000**

2.3.6. Sobrecarga producida por interrupciones

Se estima como umbral para la detección de una interrupción aquel intervalo temporal igual o mayor que el doble del tiempo de respuesta de la función *clock_gettime()* en el cual el proceso en ejecución no recoge dicha información temporal.

2.3.7. Servicios omitidos en la caracterización

El resto de servicios que conforman el perfil de un "Sistema de Tiempo Real Mínimo" descrito en el perfil del estándar POSIX que nos ocupa han sido omitidos en la caracterización realizada por no considerarse tan relevantes en las operaciones de tiempo real que habitualmente se emplean en aplicaciones reales. Para realizar esta selección nos hemos basado en la experiencia del grupo de Computadores y Tiempo Real de la Universidad de Cantabria en la implementación de controladores para robots industriales. Los servicios omitidos se listan en las líneas siguientes:

- Funciones de salto a un contexto diferente (*longjmp()* y *setjmp()*). Estas funciones están relacionadas con los manejadores de señal tradicionales que no se recomiendan en sistemas de tiempo real.
- Funciones de la librería de lenguaje C: no son específicas de sistemas de tiempo real.
- Primitivas de Entrada / Salida: no son caracterizadas ya que su tiempo de ejecución es fuertemente dependiente del dispositivo concreto empleado.
- Semáforos: para sincronización se recomienda el uso de mutex y variables condicionales.
- Funciones de entorno de proceso: no son específicas de sistemas de tiempo real.
- Bloqueo de memoria de proceso: no es objeto de caracterización dado que su principal funcionalidad es la de proporcionar portabilidad en las aplicaciones.
- Memoria compartida: no es objeto de caracterización por el mismo motivo que el caso anterior.

2.4. Infraestructura para las medidas

La caracterización de la plataforma en que se lleva a cabo un grupo de medidas es de vital importancia de cara a completar la validez de los datos que de ellas se obtienen. Necesitamos conocer el ambiente y las condiciones en que se llevan a cabo los experimentos para, una vez realizados, poder interpretar los resultados arrojados y obtener las conclusiones correspondientes.

2.4.1. MaRTE OS

Las siglas de MaRTE OS hacen referencia a *"Minimal Real Time Operating System for Embedded Applications"*. Se trata de un sistema operativo de tiempo real para aplicaciones embebidas que sigue las especificaciones establecidas en el estándar POSIX.13 (PSE51) desarrollado por el grupo "Computadores y Tiempo Real" el Departamento de Electrónica de Comunicaciones de la Universidad de Cantabria.

MaRTE OS no consta únicamente de un núcleo que implementa las funcionalidades descritas en el subconjunto mínimo de POSIX, sino que además incluye una serie de aplicaciones y servicios destinados a facilitar la creación, carga y depuración de aplicaciones. La mayoría de su código está escrito en Ada, aunque incluye algunas partes del ensamblador escritas en C, lo que permite el desarrollo cruzado de aplicaciones de software en Ada y C, empleando los compiladores GNU Gnat y Gcc. El lenguaje C es empleado para el código de inicialización del sistema, la salida de caracteres por consola, y las rutinas de cambio de contexto [GCALB].

Dado que MaRTE OS está pensado para sistemas empotrados pequeños, y no es habitual que estos sistemas dispongan de terminal de interfaz con el usuario, no incorpora ningún intérprete de comandos, ya que este aumentaría notablemente su tamaño.

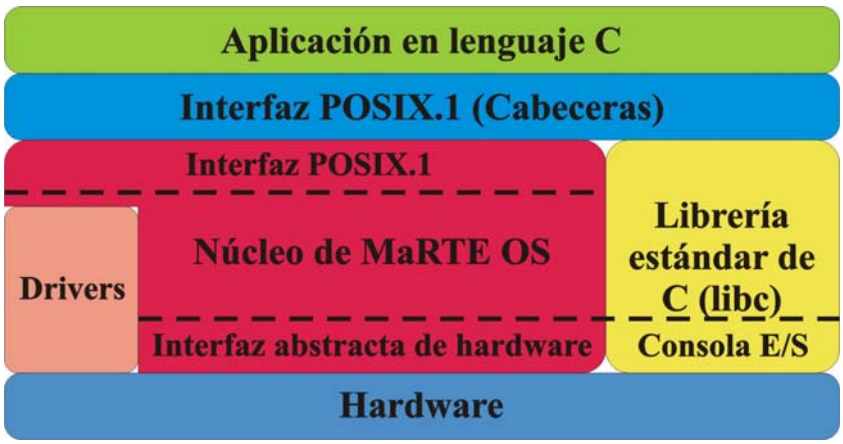

Figura 2.3. Arquitectura de MaRTE OS para lenguaje C

El núcleo del sistema operativo dispone de una interfaz abstracta de bajo nivel con el hardware que permite que su funcionamiento sea independiente del hardware sobre el que se ejecuta. Mediante esta interfaz el núcleo puede realizar una serie de operaciones por las que accede al reloj y al temporizador del sistema, permite instalar manejadores de señal y realizar el cambio de contexto entre tareas [GCALB]:

- ***Temporizador-hardware:*** este dispositivo puede ser programado para provocar una interrupción cuando transcurra un determinado periodo de tiempo.

- ***Reloj:*** permite leer y modificar la hora actual. Debe tener capacidad para almacenar el tiempo desde "la Época", aunque no se exige que está sea asignada

en la inicialización. En caso de que el sistema carezca de reloj, MaRTE OS realiza la programación periódica del temporizador.

- ***Operaciones de conversión de tiempos:*** se proporciona una función de conversión de cuentas del reloj a unidades de tiempo y viceversa. MaRTE OS utiliza internamente el formato usado por el reloj para representar el tiempo. Es preferible emplear las unidades del reloj en lugar de las del temporizador puesto que las operaciones matemáticas en las que interviene el valor actual del reloj son muy frecuentes con lo que se evitan innecesarias conversiones de tipo.

- ***Interrupciones de hardware:*** Como mínimo ha de tener la fuente de interrupción correspondiente al temporizador, aunque pueden darse otras. Se facilita una operación que permita asociar un procedimiento a una interrupción para que sea ejecutado en cuanto se produce la interrupción. A cada interrupción se le asigna un manejador que, además de finalizar la aplicación, informa al usuario de la interrupción producida.

- ***Dispositivo controlador de las interrupciones:*** que permite habilitar o deshabilitar las interrupciones, ya sea de forma individual o conjunta. Inicialmente, todas las interrupciones están deshabilitadas por defecto.

- ***Cambio de contexto entre tareas:*** esta operación tiene como argumentos la parte alta de la pila correspondiente a cada una de las tareas. El estado de los registros del procesador de la tarea saliente se almacena en su pila, y el de la entrante se restaura.

- ***Registros del procesador:*** se proporcionan operaciones para leer y escribir el registro de estado del procesador, así como habilitar y deshabilitar todas las interrupciones modificando uno o varios bits del registro.

Entorno de desarrollo

El entorno de MaRTE está formado por un computador que corre bajo Linux realizando la función de “sistema de desarrollo” y un ordenador 386 que hace las funciones de “plataforma de ejecución” conectados por una red de área local Ethernet (para la inicialización de aplicaciones) y un puerto serie (para el depurador remoto):

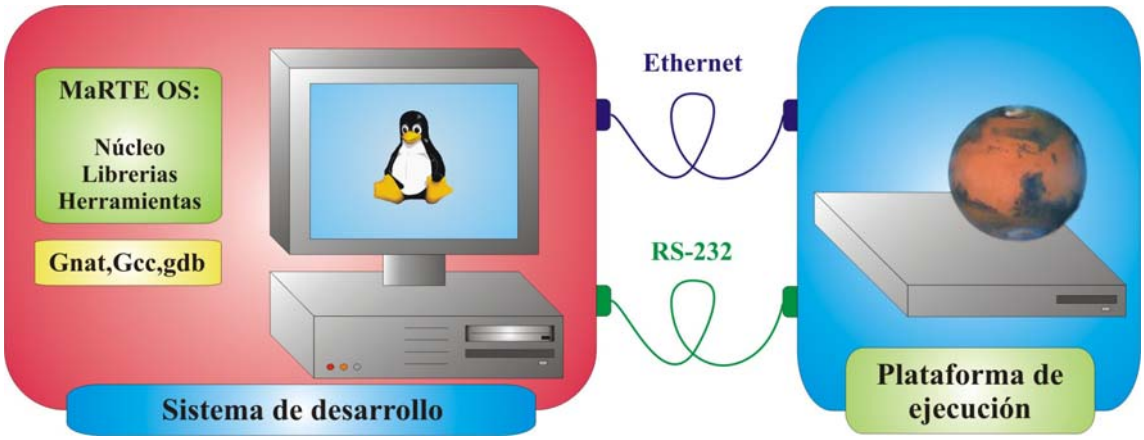

Figura 2.4. Entorno de desarrollo de MaRTE OS

Las fases de desarrollo requeridas para el desarrollo de una aplicación son las siguientes [MaRTE]:

1. El "plataforma de ejecución" se inicializa con un disquete de arranque por red generado durante la instalación de MaRTE OS.
2. La aplicación se compila y enlaza en el "sistema de desarrollo" mediante las ordenes **mgnatmake** (para códigos en lenguaje Ada) y **mgcc** (para códigos en lenguaje C).
3. El programa de arranque en red en el "plataforma de ejecución" descarga la aplicación desde el "sistema de desarrollo" a través de la red *Ethernet* y lo ejecuta.
4. La aplicación se ejecuta o se depura en remoto desde el "sistema de desarrollo".
5. Cuando termina la ejecución de la aplicación, el programa de arranque en red reinicia el *"plataforma de ejecución"* y empieza de nuevo en la fase 2.

Funcionalidades implementadas

MaRTE OS implementa las funcionalidades que pasan a enumerarse a continuación [MaRTE]:

- Gestión de threads: creación, finalización, atributos, etc.
- Planificación por prioridad: FIFO, round robin, política de planificación de servidor esporádico.
- Mutexes y variables condicionales.
- Señales.
- Relojes monotónico y de tiempo real, y temporizadores.
- Reloj y temporizadores de CPU.
- Consola de entrada – salida.
- Servicios de gestión de tiempo: suspensión absoluta y relativa, y de pthreads.
- Gestión dinámica de memoria.

Además de las funcionalidades descritas, MaRTE OS incluye algunos servicios que no se contemplan actualmente en el perfil mínimo, pero que pueden resultar de utilidad en las aplicaciones de tiempo real [ALD02]:

- Gestión de interrupciones a nivel de aplicación: proporciona una interfaz que define las operaciones básicas de habilitación y deshabilitación de interrupciones, así como de la instalación de procedimientos para ser ejecutados a la más alta prioridad justo después de producirse la interrupción. Esta intefaz también incluye una operación para bloquear una tarea a la espera de la generación de una interrupción de hardware.

- Interfaz para la definición de políticas de planificación y protocolos de sincronización: incluye una interfaz de aplicación que permite a las aplicaciones emplear algoritmos de planificación y sincronización compatibles con los definidos en el estándar.

- Añadidos a POSIX: incorpora una adaptación a Ada para relojes de CPU y el acceso a datos específicos de un thread diferente al propietario.

El algoritmo de gestión de memoria dinámica empleado en el sistema operativo que va a ser caracterizado en los capítulos siguientes, MaRTE OS, es el TLSF (*Two Level Segregated Fit*), el cual pasa a ser detallado a continuación.

TLSF: Two Level Segregated Fit

Entre los expertos en materia de tiempo real, el empleo de técnicas de memoria dinámica suele evitarse, debido a que los peores casos, tanto espaciales como temporales, para las operaciones de asignación y liberación de memoria, no estaban lo suficientemente acotados.

Actualmente, se han desarrollado algoritmos de asignación dinámica de memoria con un comportamiento temporal aceptable y limitado. Uno de estos algoritmos es el denominado TLSF (*Two Level Segregated Fit*), que ha sido desarrollado específicamente para sistemas operativos de tiempo real. [MRCVa]

El algoritmo TLSF (*Two Level Segregated Fit*) ha sido diseñado teniendo en cuenta que los sistemas de tiempo real habitualmente no disponen de mucha capacidad de memoria, ni de hardware especifico que soporte memoria virtual (MMU), con el objetivo de cubrir la mayoría de los requisitos de las aplicaciones de tiempo real:

- Tiempo de respuesta acotado
- Tiempo de respuesta rápido
- Fragmentación acotada
- Bajo nivel de fragmentación

2.3.2. Entorno de trabajo

La plataforma de trabajo sobre la que han sido llevados a cabo los experimentos consiste en dos ordenadores que funcionan como *"plataforma de ejecución"* de un sistema. Los dos forman parte de un proyecto en el que se están desarrollando controladores distribuidos para robots industriales.

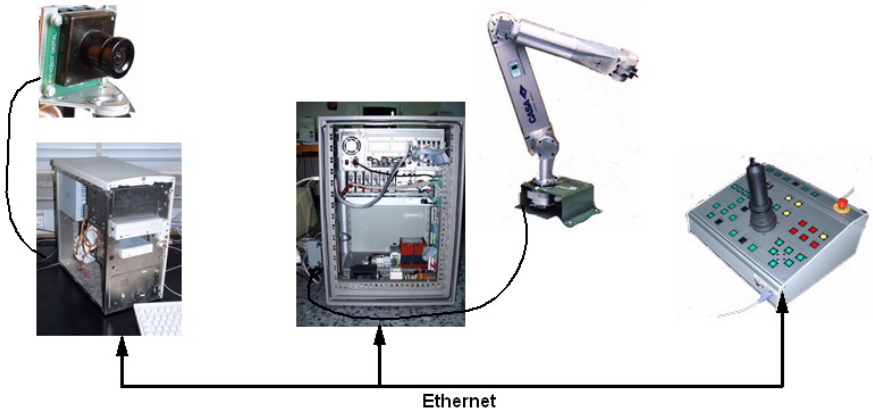

Figura 2.5. Entorno de trabajo

La primera "plataforma de ejecución" es un panel de botones que junto con una palanca de mando controla los movimientos de un brazo articulado. Corre bajo MaRTE OS en un Pentium III a 750 MHz, y se conecta por el puerto COM1 al "sistema de desarrollo" para depurar aplicaciones.

La segunda "plataforma de ejecución" es un controlador del sistema. También es un Pentium III a 750 MHz corriendo bajo MaRTE OS, y se conecta al "sistema de desarrollo" por el puerto COM2 para tareas de depuración de aplicaciones.

Figura 2.6. Plataformas de ejecución

La versión de sistema operativo empleada es MaRTE OS es 1.56, y el compilador gcc 3.2.3, y gnat GAP 1.0.

3.- Estrategia de medida

3. Estrategia de medida

3.1. Estructura común para todos los servicios

El conjunto de experimentos que se han programado para llevar a cabo la caracterización de los sistemas operativos de tiempo real mínimo tiene la estructura que se muestra en la siguiente gráfica:

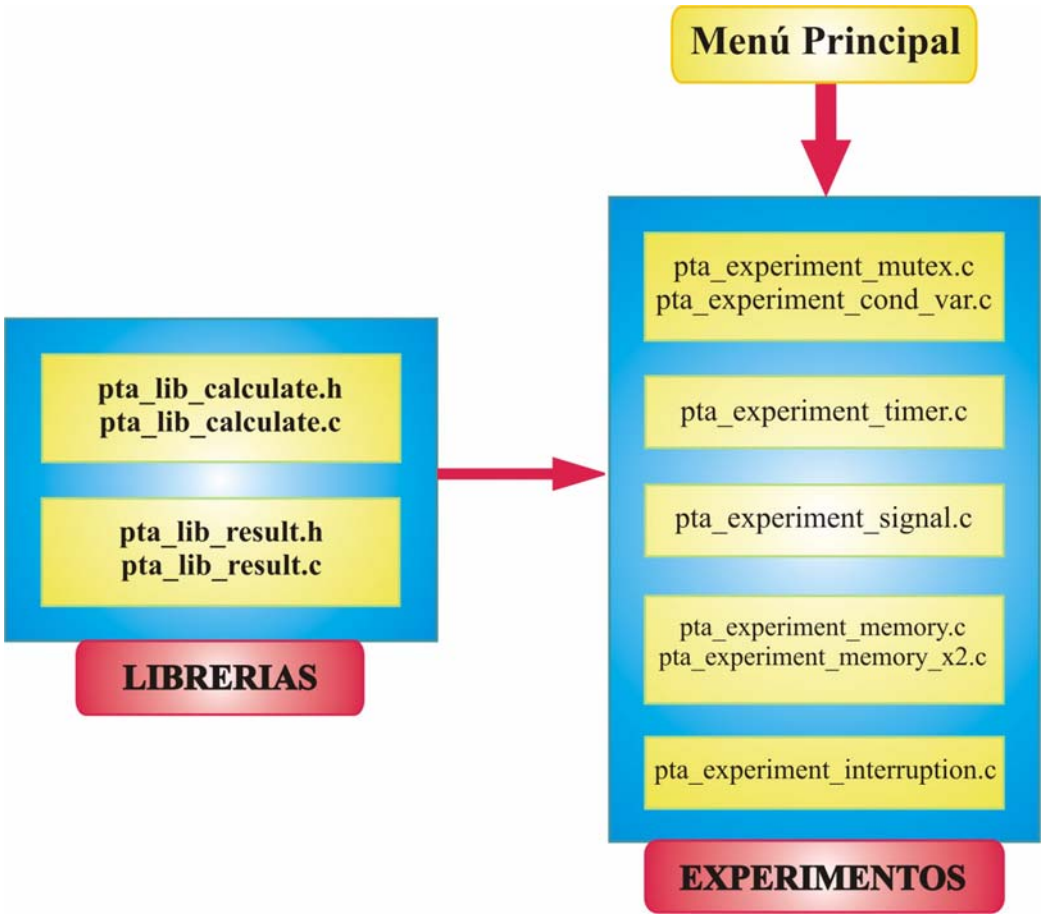

Figura 3.1 Esquema general de los experimentos

Por un lado se encuentran las librerías. La primera de ellas, **pta_lib_calculate** contiene las funciones necesarias para realizar los cálculos de las medidas temporales. Mientras que la segunda, **pta_lib_result**, facilita las funciones necesarias para la representación de los resultados obtenidos de las medidas realizadas.

Por otro lado disponemos de un programa principal que nos permite acceder a cada uno de los experimentos que evalúan cada uno de los servicios. En primer lugar, los mecanismos que facilitan la sincronización, mutex y variables condicionales. Para ello, disponemos de ***pta_experiment_mutex*** y ***pta_experiment_cond_var***. En segundo lugar encontramos un experimento destinado a evaluar los servicios de gestión de tiempo: resolución del reloj, temporizadores, retrasos de alta resolución, etc., en ***pta_experiment_timer***. El tercer grupo de experimentos lo componen las señales de tiempo real, en ***pta_experiment_signal***. La gestión de memoria se realiza mediante dos experimentos, ***pta_experiment_memory*** y ***pta_experiment_memory_x2***, en el que se varían los bloques de tamaño que se reservan, para evaluar las repercusiones que esto tiene sobre los tiempos de ejecución de las sentencias de reserva y liberación (*malloc* y *free* respectivamente). Por último, se realiza un experimento con el fin de determinar la sobrecarga que se produce en el sistema debida a las interrupciones. Para ello, ***pta_experiment_interruption*** registrará el número de interrupciones que se producen, así como su duración correspondiente, durante un determinado intervalo de tiempo.

Desde el programa principal se puede acceder a cada uno de los experimentos y especificar el número de iteraciones que han de realizarse para tomar cada una de las medidas.

3.2. Librerías

3.2.1. Librería de cálculos: pta_lib_calculate

En el archivo cabecera, ***pta_lib_calculate.h***, se describen, por un lado, una estructura de datos que se empleará para las medidas, y por otro, las interfaces de las funciones que serán empleadas para realizar dichas medidas. El archivo ***pta_lib_calculate.c*** contiene el desarrollo de cada una de estas funciones.

La estructura de datos, **pta_time_data**, almacenará los resultados de cada una las medidas realizadas para los distintos servicios, y contiene los siguientes campos:

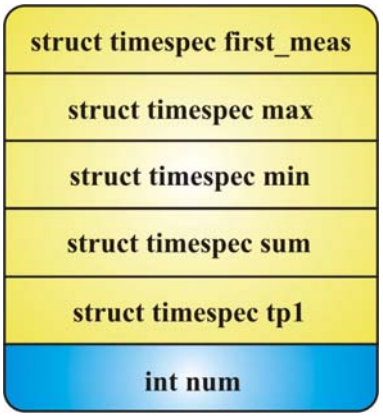

Figura 3.2 Estructura de datos para las medidas: pta_time_data

En los cinco primeros campos de la estructura, almacenaremos datos temporales en formato **struct timespec** (definido en la librería <*time.h*>), y en la última variable el número de iteraciones realizadas para esa medida:

- **first_meas**: almacena la primera medida ya que previsiblemente será la mayor de todas las registradas, debido que la función a caracterizar temporalmente ha de ser cargada en caché cuando va a ser usada por primera vez.
- **max**: almacena el tiempo máximo registrado para el número de iteraciones indicado en **num**. Se excluye la primera medida.
- **min:** almacena el tiempo mínimo registrado para el número de iteraciones indicado en num.
- **sum:** almacena el tiempo correspondiente la suma de medidas en el número de iteraciones indicado en **num** para posteriormente calcular la media del total de medidas.
- **tp1:** guarda temporalmente el tiempo en el que se inicia la medida.
- **num:** es una variable de tipo entero que contiene el número de iteraciones que se han realizado para una medida concreta.

Para el experimento en el que se realiza la caracterización de la gestión de tiempo, se hace necesario definir una estructura que almacene la resolución del reloj. Esta estructura, **pta_timer**, tiene la forma que se muestra a continuación:

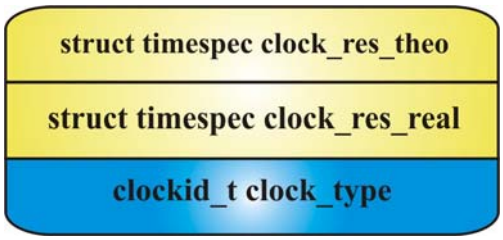

Figura 3.3 Estructura de datos para medidas de resolución: pta_timer

Las estructuras temporales (**clock_res_theo** y **clock_res_real**) almacenan la resolución del reloj teórica (***clock_getres()***) y la real obtenida mediante la lectura del reloj en instantes sucesivos de tiempo. Así mismo, se almacena el tipo de reloj para el cual se realizan las medidas en **clock_type.**

Las funciones que realizarán los cálculos correspondientes a las medidas temporales son las que se muestran en el siguiente fragmento del archivo cabecera ***pta_lib_calculate.h***:

```
struct timespec pta_substract_timespec
   (struct timespec tp1, struct timespec tp2);
struct timespec pta_add_timespec
   (struct timespec tp1, struct timespec tp2);
void pta_eval_time();
```

```
void pta_initiate_measurement
   (pta_time_data *measurement);
void pta_finalize_measurement
   (pta_time_data *measurement);
void pta_initiate_measurement_delay
   (pta_time_data *measurement,struct timespec time1);
void pta_finalize_measurement_delay
   (pta_time_data *measurement,struct timespec time2);
```

En primer lugar, dos funciones cuya finalidad es realizar la suma y resta de estructuras temporales en el formato *struct timespec*. Los argumentos de entrada de estas funciones son los operandos del cálculo. En el caso de la operación resta el argumento *tp1* corresponde al tiempo inicial, y *tp2* al tiempo final. Ambas funciones devuelven una variable del mismo formato que los argumentos a introducir, con el resultado de la operación.

En segundo lugar, y dada la forma en la que se va a llevar a cabo el procedimiento de medida, se hace necesario tener caracterizada temporalmente la función *clock_gettime()*, que devuelve la hora actual del sistema. Para ello ha sido definida la función **pta_eval_time()**. Esta función no requiere argumentos de entrada, mide el tiempo que tarda en ejecutarse dicha función, y devuelve el mínimo valor registrado en 100 iteraciones con el reloj de tiempo real. Su resultado se almacena en una variable global (definida en ***pta_lib_calculate.c***). Esta función deberá ser ejecutada antes de realizar los experimentos de medida.

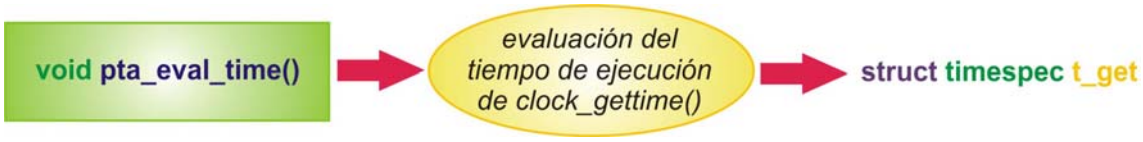

Figura 3.4 pta_eval_time()

Las funciones restantes se emplean para inicializar y finalizar los procesos de medida en los experimentos. La función **pta_initiate_measurement()** recibe como argumento de entrada un puntero a una estructura de datos **pta_time_data**. Inicializa las variables de dicha estructura y obtiene el tiempo en que se inicia la medida.

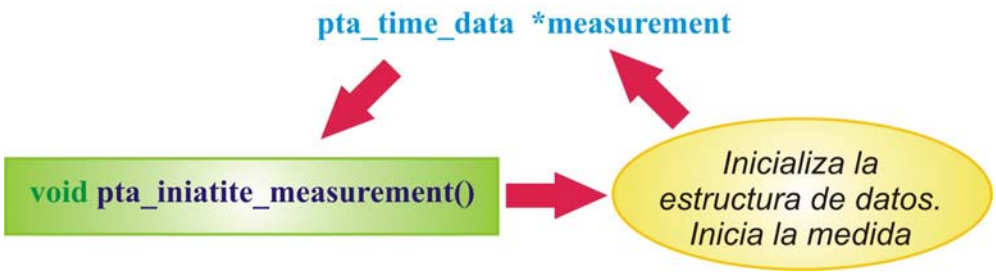

Figura 3.5 pta_initiate_measurement()

La función **pta_finalize_measurement** obtiene el tiempo en que termina la medida, y calcula el tiempo de ejecución del experimento y actualiza los valores de todas las componentes de la citada estructura de datos. Los argumentos de entrada y salida son los mismos que en la función de inicialización.

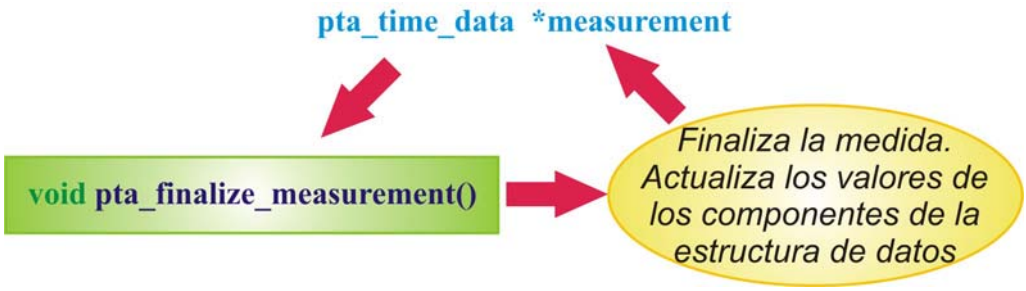

Figura 3.6 pta_finalize_measurement()

De modo que, el tiempo de ejecución de una función o sentencia concreta se calculará siguiendo el siguiente proceso:

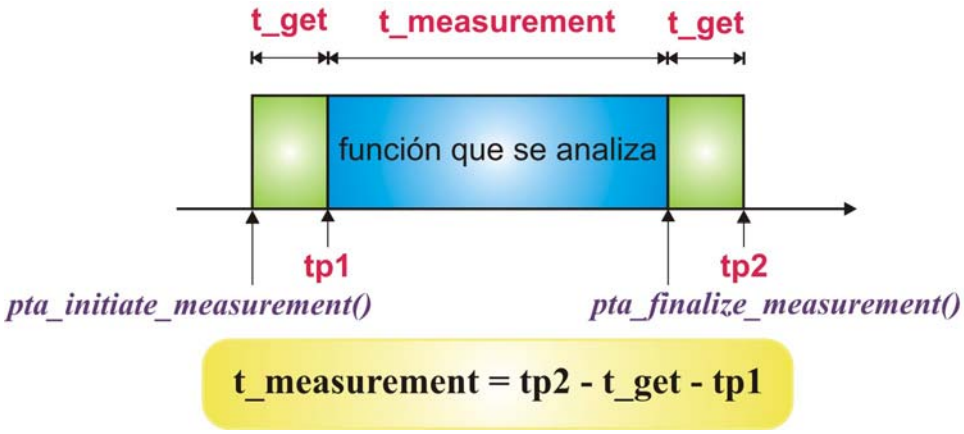

Figura 3.7 Proceso de medida

Para las medidas de los retrasos de alta resolución, la toma de los tiempos se tomará en el propio thread. Por este mismo motivo, se define una variante de las funciones de inicialización y finalización de medida, en las que solo se realizan los cálculos y se actualizan los valores de los componentes correspondientes de la estructura de datos de cada una de las medida. Estas funciones son ***pta_inititate_measurement_delay()*** y ***pta_finalize_measurement_delay()***, y tendrán como argumentos de entrada, la estructura de datos en las que se almacena los valores de la medida que se esté realizando y los tiempos de inicio y final de la ejecución de la función que se esté evaluando.

3.2.2. Librería de representación de resultados: pta_lib_result

En esta librería se definen las funciones para representar por pantalla los resultados obtenidos en los experimentos. Se trata de cuatro funciones cuya finalidad es tratar los diferentes tipos de datos para mostrarlos correctamente al usuario. El fragmento del archivo cabecera, pta_lib_result.h, en el cual se presenta el interfaz de estas funciones es el que se muestra a continuación:

```
pta_write_timespec (struct timespec arg);
pta_write_time_data(char *name[], pta_time_data_meas);
pta_write_experiment_header (char *name);
pta_write_resolution (char *name[], pta_timer res);
```

La función **pta_write_timespec()**: recibe como argumento de entrada un dato del tipo *struct timespec*, y lo representa por pantalla.

La función **pta_write_time_data()**: recibe como argumentos de entrada el nombre del experimento y una estructura de datos del tipo **pta_time_data**, y realiza su representación por pantalla. Para ello hace uso de la función anterior.

La función **pta_write_experiment_header()**: recibe como argumento de entrada el nombre del servicio que se ha analizado, y muestra lo muestra por pantalla, junto con la fecha y hora en la que se realizan las medidas.

La función **pta_write_resolution()**: recibe como argumento de entrada el nombre de la función de la cual se ha calculado la resolución, así como la estructura de datos que alberga dicha resolución, y lo muestra por pantalla.

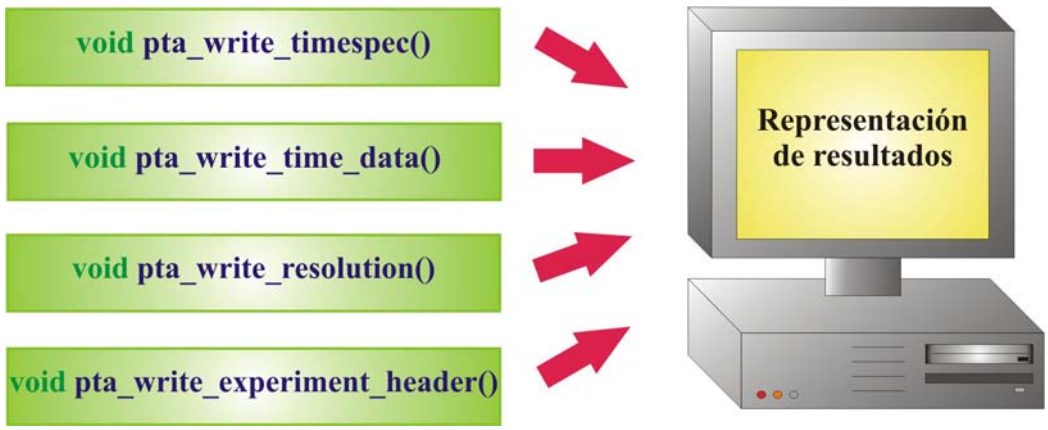

Figura 3.8 pta_lib_result

3.2.3. Librerías específicas para el entorno de desarrollo MaRTE OS

Para la aplicación práctica de este conjunto de experimentos sobre el entorno MaRTE OS, con el fin de poder almacenar en un fichero los resultados obtenidos, y teniendo en cuenta que esta acción no puede llevarse a cabo sobre la plataforma de ejecución, puesto que no consta con un sistema de almacenamiento, se hace uso de un procedimiento mediante el cual se envían los datos por la red Ethernet al sistema de desarrollo, donde sí pueden ser almacenados. Para ello, se hace uso de un protocolo de red en tiempo real para sistemas distribuidos actualmente en desarrollo en el departamento en el cual ha sido realizado este trabajo. Las características más representativas de las librerías que realizan esta función son las que pasan a mostrarse a continuación:

pta_send_result:

En esta pequeña librería se describe dos funciones cuya finalidad es establecer una conexión mediante un *socket TCP/IP* entre el sistema de desarrollo y la plataforma de ejecución. Se ejecuta desde la plataforma de desarrollo y su misión es enviar hacia el sistema de desarrollo los resultados de las medidas obtenidas. La interfaz de esta librería, definida en **pta_send_result.h** es:

```
void pta_tx_data
   (int meas_num, pta_time_data data_res[meas_num]);
void pta_tx_data_res
   (int meas_num, pta_timer[meas_num]);
```

Los argumentos de entrada indican por un lado el número de medidas que han de ser transmitidas, así como el conjunto de valores de las medidas que se van a enviar.

pta_catch_result.c:

Este archivo es un ejecutable, que corresponde a la otra parte de la comunicación. Se ejecuta en el sistema de desarrollo y tiene como función la de recibir los datos enviados desde la plataforma de ejecución y almacenarlos en un fichero. Para escribir los datos recibidos en un fichero, se ha realizado una pequeña variación de la librería **pta_lib_result** ya explicada, en la que en vez de representar visualmente los datos, estos son almacenados en un fichero. Esta librería recibe el nombre de **pta_lib_result_file**, y el interfaz de las funciones que contiene son las que se muestran a continuación:

```
pta_write_timespec_file
   (FILE *file_desc, struct timespec arg);
pta_write_time_data_file
   (FILE *file_desc,char*name[], pta_time_data_meas);
pta_write_experiment_header_file
   (FILE *file_desc, char *name);
pta_write_resolution_file
   (FILE *file_desc, char *name[], pta_timer res);
```

Como se puede observar son las mismas funciones que las ya descritas, en las que se incluye como argumento el identificador del fichero en el que van a ser escritos los datos recibidos.

3.3. Experimentos para caracterizar los mecanismos de sincronización

3.3.1. Mutex

3.3.1.1. Datos

En primer lugar se define el número de grupos de medidas que van a realizarse sobre este mecanismo de sincronización, para posteriormente definir un array con el número de estructuras de datos, **pta_time_data**, necesario para llevarlas a cabo. Así mismo, se definen otras variables necesarias para la realización de este experimento, como son el número de threads que se ejecutarán simultáneamente, la estructura de datos que comparten estos threads, o el nombre del experimento para su posterior representación en pantalla. Estos toman los valores que siguen a continuación:

- Número de grupos de medidas: **num_meas_mutex**:
  - En MaRTE OS: 14; ya que se lleva a cabo el análisis de 7 posibles situaciones para cada uno de los protocolos de prioridad del mutex, por herencia de prioridad, y con prioridad de prioridad.
  - En LINUX: 9; ya que se analizan esos 7 casos para un único protocolo, ya que Linux no cuenta con esos atributos para la planificación, junto con la inicialización y destrucción de la variable compartida.

  La estrategia de medida seguida para realizar estos análisis se detallará a continuación.

- Número de threads que se ejecutan simultáneamente: th_wait: 10; debido a que es el número habitual de tareas simultaneas en un robot implementado con un sistema de las características propuestas en este trabajo.

La estructura de datos que comparten los threads para la realización del experimento es:

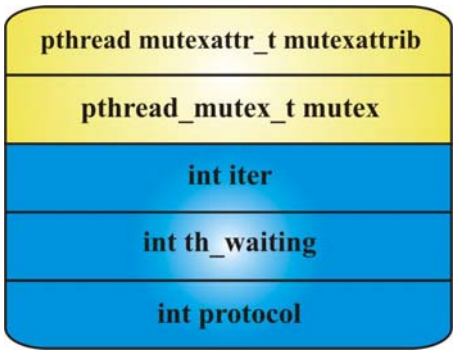

Figura 3.9 Mutex - Estructura de datos para los threads

3.3.1.2. Procedimiento de medida

Dado que hay varios casos que tomar en cuenta, en función de la configuración de varios parámetros (protocolos, prioridades, número de threads, etc.) se clasifica el análisis por casos. Aunque, en primer lugar, y sin atender a los parámetros de configuración se analizan los tiempos de creación y destrucción del mutex, así como de las funciones de configuración del techo de prioridad *pthread_mutex_setprioceiling()* y *pthread_mutex_getprioceiling()*. El resto del análisis se ha dividido en casos, los cuales se explican a continuación. Para todos ellos se sigue el siguiente procedimiento común:

- La política de planificación es SCHED_FIFO, porque es la más comúnmente implementada en los sistemas de tiempo real.
- El thread principal, **main()** es el de mayor prioridad, para garantizar que se creen a tiempo los threads necesarios para los diferentes casos.
- Los threads creados para la realización de los experimentos se crean con los siguientes atributos:
    - PTHREAD_EXPLICIT_SCHED: para poder aplicar diferentes atributos a los threads, y que estos no los hereden del thread principal.
    - PTHREAD_CREATE_JOINABLE: para la espera sincronizada en la terminación de los threads desde el thread principal.

Para cada uno de los análisis se crean los threads con su prioridad correspondiente, que realizan una función específica para cada análisis, y el thread principal espera su terminación sincronizada, mediante la función *pthread_join()*.

Análisis 1: Bloqueo y desbloqueo de mutex, sin que haya otro thread esperándolo

En él se analizan los tiempos de bloqueo y desbloqueo del mutex (*pthread_mutex_lock(), pthread_mutex_unlock() y pthread_mutex_trylock()*). Para ello se crea un thread que accederá a tomar y liberar este objeto de sincronización, sin que haya ningún otro thread esperando para acceder a él, es decir, el acceso al mutex esté libre. Se toman en cuenta los las protocolos por herencia de prioridad y protección de prioridad: PTHREAD_PRIO_INHERIT y PTHREAD_PRIO_PROTECT.

Análisis 2: Desbloqueo de un mutex en un thread de baja prioridad más tiempo de activación de un thread de mayor prioridad que estaba esperándolo

Dos threads de diferentes prioridades intentan acceder al mutex. El thread de menor prioridad bloquea el mutex, y antes de liberarlo inicia la medida mientras el thread de mayor prioridad espera a que este lo libere para terminar de realizar la medida. Los protocolos analizados son por herencia de prioridad, PTHREAD_PRIO_INHERIT, y por protección de prioridad, PTHREAD_PRIO_PROTECT. El procedimiento completo que se sigue para realizar la medida es el que se muestra en las figura 3.10 y 3.11 para el protocolo con herencia de prioridad y por protección de prioridad respectivamente.

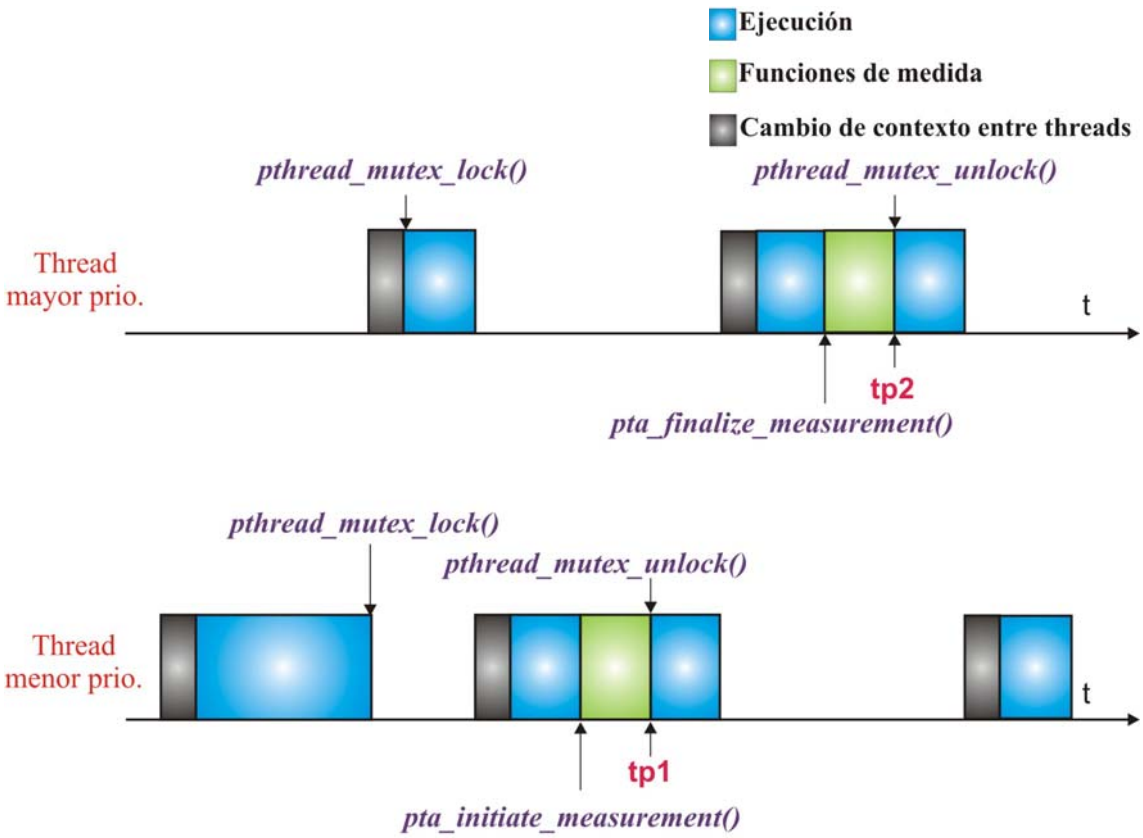

Figura 3.10 Mutex - Desbloqueo de un mutex en un thread de baja prioridad más tiempo de activación de un thread de mayor prioridad que estaba esperándolo para el protocolo por herencia de prioridad

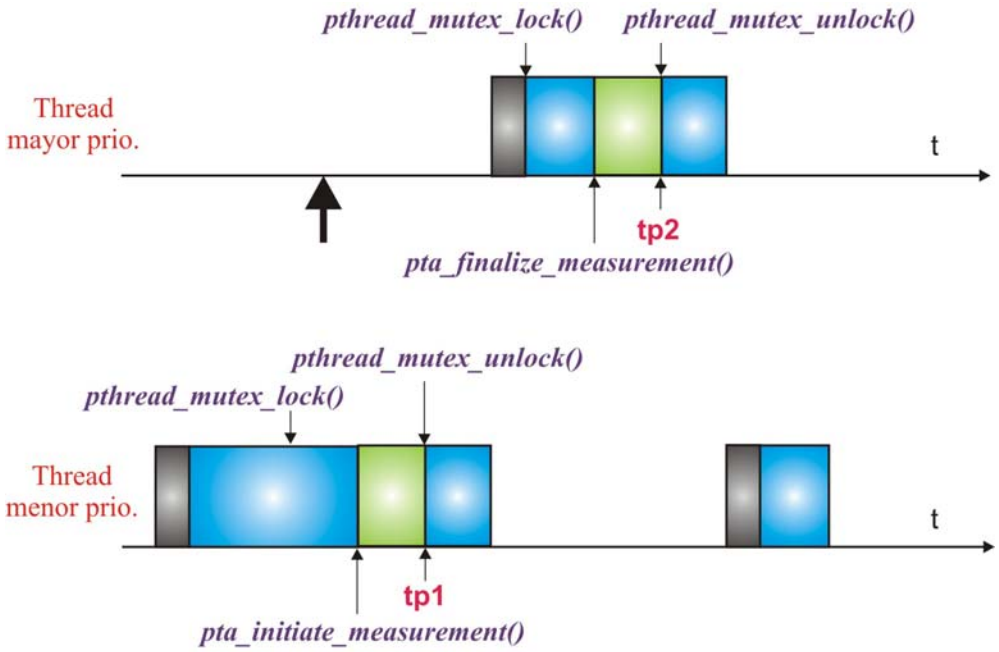

Figura 3.11 Mutex - Desbloqueo de un mutex en un thread de baja prioridad más tiempo de activación de un thread de mayor prioridad que estaba esperándolo para el protocolo de protección de prioridad

Análisis 3: Desbloqueo de un mutex en un thread de baja prioridad más tiempo de activación de un thread de mayor prioridad que estaba esperándolo (varios threads)

Este último caso sigue el mismo procedimiento que el anterior, salvo por la diferencia de que son varios los threads que se están esperando al mutex. El que termina la medida es el de mayor prioridad, mientras que el resto de threads que estaban esperando, lo único que hacen es bloquear y desbloquear el mutex cuando por orden de prioridad les vaya correspondiendo el acceso.

3.3.2 Variables Condicionales

3.3.2.1 Datos

En primer lugar, y al igual que en apartado anterior, se define el número de grupos de medidas que van a realizarse sobre este mecanismo de sincronización, para posteriormente definir un array con el número de estructuras de datos, **pta_time_data**, necesario para llevarlas a cabo. Así mismo, son definidas otras variables necesarias para la realización de este experimento, como son el número de threads que se ejecutarán simultáneamente, la estructura de datos que comparten estos threads, o el nombre del experimento para su posterior representación en pantalla. Estos toman los valores que siguen a continuación:

- Número de grupos de medidas (**num_meas_cond_var**): 11; ya que son estos las diferentes situaciones objeto de estudio para este mecanismo de sincronización.
  Los diferentes casos que componen este experimento serán explicados a continuación.

- Número de threads que se ejecutan simultaneamente (**th_wait**): 10; ya que como se explicó en el apartado de los mutex es el número habitual de tareas que se ejutan simultáneamente en un robot con estas características.

En este experimento, la estructura de datos necesaria para que los threads compartan el objeto de sincronización es la que sigue:

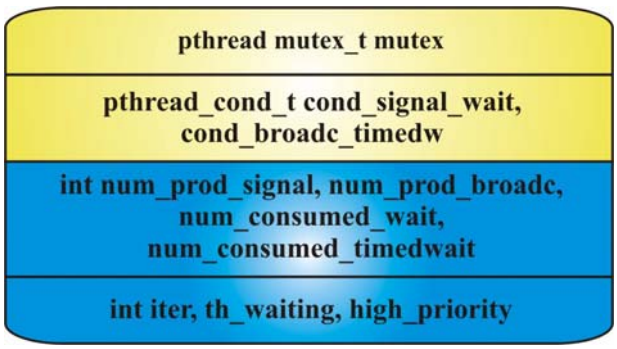

Figura 3.12 Variables Condicionales - Estructura de datos para los threads

3.2.2.2 Procedimiento de medida

Para determinar la prioridad de los threads se hace uso del protocolo de planificación SHED_FIFO, por el mismo motivo que se indicó en el caso de los mutex. Además, se siguen las mismas características que en el experimento anterior:

- El thread principal, **main()**, es el de mayor prioridad, para garantizar que se creen a tiempo los threads necesarios para los diferentes casos.
- Los threads creados para la realización de los experimentos se crean con los siguientes atributos:
    - PTHREAD_EXPLICIT_SCHED: para que poder aplicar diferentes atributos a los threads, y que estos no los hereden del thread principal.
    - PTHREAD_CREATE_JOINABLE: para la espera sincronizada en la terminación de los threads desde el thread principal.

Con el objetivo de simplificar el análisis y seguir un orden, como ya se estableció en el experimento anterior, éste ha sido dividido en casos, los cuales pasarán a ser detallados a continuación. Para cada uno de los análisis se crea los threads con su prioridad correspondiente, que realizan una función específica para cada análisis, y el thread principal espera su terminación sincronizada, mediante *pthread_join()*.

Cabe resaltar, que antes de proceder con el estudio de cada uno de los casos, se realiza la caracterización temporal de las funciones específicas para crear y destruir un objeto de variable condicional: *pthread_cond_init()*, y *pthread_cond_destroy()*.

Análisis 1: Señalización más activación de un thread de mayor prioridad que estaba esperando el acceso a la variable condicional

Dos threads de diferentes prioridades comparten una variable condicional. El de mayor prioridad espera a que el de menor prioridad produzca el dato que él “consume”. Se evalúa el tiempo desde que el thread de menor prioridad produce el dato hasta que se activa el thread de mayor prioridad, una vez se cumple el predicado de la condición.

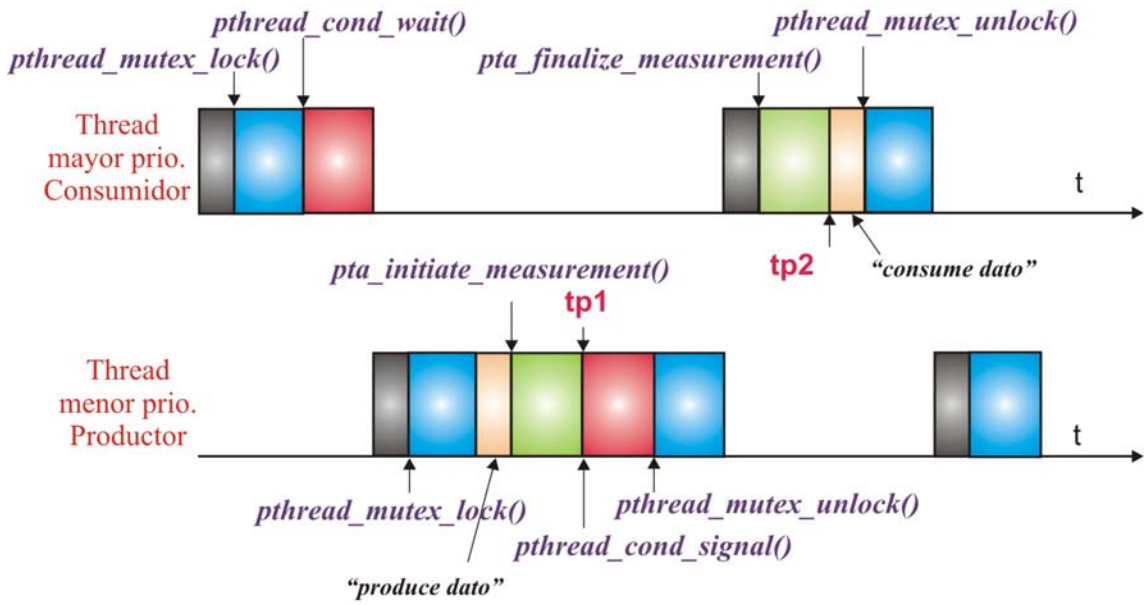

Figura 3.13 Variables Condicionales: Señalización + activación de un thread de mayor prioridad

Análisis 2: Señalización más activación de un thread de mayor prioridad que estaba esperando el acceso a la variable condicional, cuando hay más threads esperando

Se evalúa el proceso caracterizado en el apartado anterior. La diferencia es que en este caso hay "**th_wait**" (que en la realización de estos experimentos es 10, como ya se ha indicado) threads en espera. El thread de menor prioridad es el que realiza las funciones de productor. Del resto de threads, el de mayor prioridad espera a que se cumpla la condición y termina la medida, el resto únicamente bloquearán y desbloquearán el mutex que proporciona el acceso mutuamente exclusivo a la variable, en el orden determinado por su prioridad.

Análisis 3: Señalización mediante broadcast más activación de un thread de mayor prioridad que realiza una espera limitada

Este caso surge también a partir del primero. En él, sustituimos la función *pthread_cond_signal()* por *pthread_cond_broadcast()* y *pthread_cond_wait()* por *pthread_cond_timedwait().* El intervalo de tiempo que se especifica como parte de la condición de espera es lo suficientemente grande como para que no se produzca un error por sobrepasar la imposición de limitación temporal.

Análisis 4: Señalización mediante broadcast más activación de un thread de mayor prioridad que realiza una espera limitada, cuando hay más threads esperando

Este caso surge de la conjunción de los casos 2 y 3. En él se emplean *pthread_cond_broadcast()* y *pthread_cond_timedwait()*, sin que se den errores por exceso de tiempo de espera, con un total de "**th_wait**" (fijado a 10) threads que esperan el acceso al objeto de sincronización. Una vez más, el thread de menor prioridad es que el "produce" los datos, y realiza la función de señalización.

Análisis 5: Señalización en un thread de alta prioridad, con un thread de menor prioridad esperando

En este caso se consideran únicamente dos threads, aunque ahora se invierten las prioridades para realizar cada una de las funciones. Es decir, el thread "productor" es el de mayor prioridad, mientras que el thread "consumidor" que espera a que cumpla la condición tiene menor prioridad. Se evalúa el tiempo necesario para realizar la señalización en el thread de mayor prioridad. El procedimiento que se sigue es este:

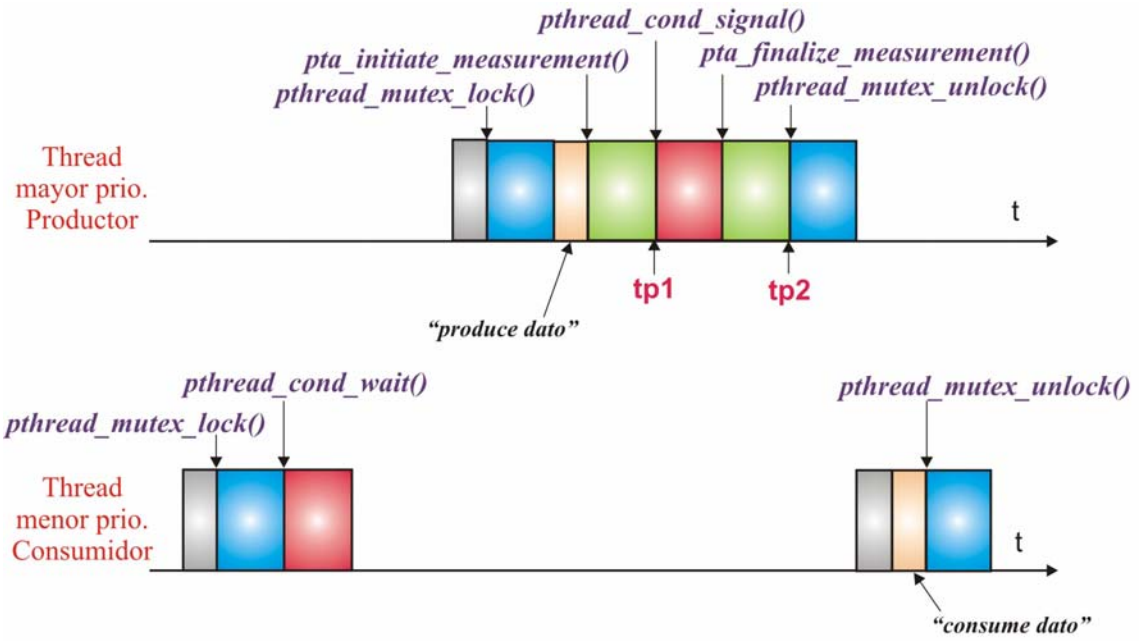

Figura 3.14 Variables Condicionales: Señalización en un thread de alta prioridad

Análisis 6: Señalización en un thread de alta prioridad, cuando varios threads de menor prioridad esperan

En este caso se analiza el mismo procedimiento que en el análisis 5, ampliando el número de threads de baja prioridad que esperan al indicado en "**th_wait**" (fijado a 10). De estos threads, el de mayor prioridad, "consume" los datos, el resto simplemente van tomando y liberando el mutex que proporciona el acceso mutuamente exclusivo según orden determinado por su prioridad.

Análisis 7 y 8: Señalización en un thread de alta prioridad, cuando uno o varios threads de menor prioridad realizan espera limitada

En ellos se analizan los casos 5 y 6 respectivamente, sustituyendo la señalización a un único thread por la señalización broadcast (*pthread_cond_broadcast()*), y la condición de espera, por una espera limitada en el tiempo con *pthread_cond_timedwait()*. Al igual que se hizo en los casos 3 y 4, se configura un intervalo para la espera que garantice que se cumplan los intervalos de tiempo necesarios para que no ocurra un error.

Análisis 9: Espera limitada fallida en un thread de baja prioridad

Por último se analiza la espera a una variable condicional con limitación de tiempo. Para ello se emplean dos threads: el de menor prioridad realiza las funciones de "consumidor" y realiza la espera mediante *pthread_cond_timedwait()*. Para asegurar que esta espera es fallida, marcamos como tiempo limite absoluto la hora a la que ha empezado a ejecutarse dicho thread (mediante *clock_gettime()*), de modo que al ser el tiempo de espera nulo, el tiempo que se analiza corresponde directamente a la operación *pthread_cond_timedwait()*. El thread de mayor prioridad "produce" los datos, y señaliza mediante *pthread_cond_broadcast()*.

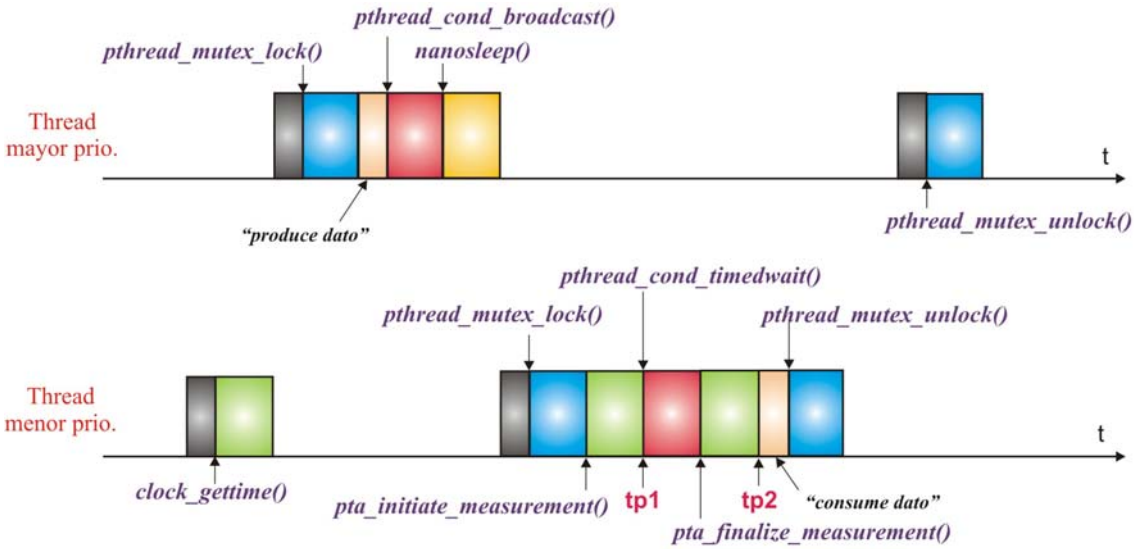

Figura 3.15 Variables Condicionales: Espera limitada fallida

3.4. Experimentos para caracterizar la gestión del tiempo

3.4.1. Datos

Este experimento es en el que más análisis se consideran. Para llevarlo a cabo se han definido un total de 33 medidas diferentes, aunque no todas siguen el formato estándar, **pta_time_data**, definido en la librería de cálculos **pta_lib_calculate**, sino que dos de esas medidas están destinadas a evaluar la resolución de los relojes monotónico y de tiempo real. Para ellas, se emplea la estructura de datos **pta_timer**, definida en la misma librería.

Por otro lado, y siguiendo las pautas establecidas en los experimentos anteriores, para determinar la prioridad de los threads se hace uso del protocolo de planificación SHED_FIFO. Además, igual que anteriormente:

- El thread principal, **main()**, es el de mayor prioridad, para garantizar que se creen a tiempo los threads necesarios para los diferentes casos.
- Los threads creados para la realización de los experimentos se crean con los siguientes atributos:
    - PTHREAD_EXPLICIT_SCHED: para que poder aplicar diferentes atributos a los threads, y que estos no los hereden del thread principal.
    - PTHREAD_CREATE_JOINABLE: para la espera sincronizada en la terminación de los threads.

La estructura de datos que manejarán los threads que realizarán cada una de las funciones de gestión de tiempo es muy sencilla, ya que en este caso los únicos datos que han de compartir son tres variables de tipo entero para especificar las condiciones de medida entre los diferentes threads, y una variable de tipo *clockid_t* que especifique el reloj con el que se está realizando esa medida concreta.

El resto de parámetros específicos de este experimento se enumeran a continuación:

- Número de grupos de medidas (**num_meas_signal**): 33; en las que se incluye la resolución del reloj, las funciones *sleep* de alta resolución en modo absoluto y relativo, con varios intervalos temporales, temporizadores, y *delays* entre threads de distinta prioridad. En la ejecución de este experimento se realizan 27 medidas, siendo descartadas los intercambios entre threads con retrasos de alta resolución.

- Número de threads que se ejecutan simultáneamente (**th_wait**): 10; igual que en los experimentos anteriores ya descritos.

- Número de iteraciones para la resolución del reloj (**res_iter**): 3; para poder identificar y tomar en cuenta el valor obtenido en la primera medida.

- Número de intervalos para *sleep* de alta resolución (**nano_interval**): 5; indica el número de intervalos que serán considerados para la caracterización temporal de las funciones **nanosleep()** y **clock_nanosleep()**.

3.4.2. Procedimiento de medida

En este experimento hay tres tipos diferenciados de medidas, cada uno de los cuales incluye varios casos. En primer lugar se analiza la resolución del reloj, en sus dos tipos. En segundo lugar se analiza la función de suspensión, *sleep*, de alta resolución relativa (*nanosleep()*) y absoluta (*clock_nanosleep()*), así como los temporizadores en modo simple relativo. Para este segundo grupo de medidas se especifican cinco intervalos de tiempo para evaluar cada una de las funciones. Por último, se analiza el retraso que se produce entre threads de diferente prioridad al realizar la función *sleep* de alta resolución. A continuación se describen detalladamente cada uno de los casos:

Análisis 1: Resolución del reloj

En este caso se evalúa la resolución del reloj monotónico, CLOCK_MONOTONIC, y del reloj de tiempo real, CLOCK_REALTIME. Estas medidas emplean el tipo de dato **pta_timer** para almacenar los resultados que se obtienen, como ya se ha indicado.

Análisis 2: Tiempo de ejecución de *nanosleep()*

Realiza una evaluación del tiempo de ejecución de la función *nanosleep()* para diferentes intervalos de tiempo, con el fin de determinar el tiempo total de suspensión. Estos intervalos de tiempo son los que se especifican a continuación:

Figura 3.16 Gestión del tiempo: Intervalos de tiempo para nanosleep()

Análisis 3: Tiempo de ejecución de *clock_nanosleep()*

Este caso tiene como objetivo determinar el tiempo real de suspensión, para la operación de *sleep* de alta resolución en modo absoluto (TIMER_ABSTIME). Este análisis se realiza para los dos relojes definidos, CLOCK_MONOTONIC, y CLOCK_REALTIME. Al igual que en el caso anterior se especifican cinco intervalos, que en esta ocasión son:

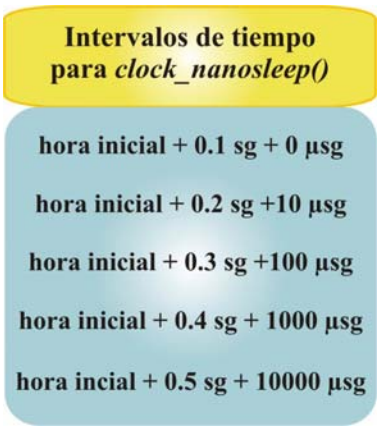

Figura 3.17 Gestión del tiempo: Intervalos de tiempo para clock_nanosleep()

Análisis 4: Temporizador en modo simple relativo

En este caso se evalúa el tiempo de expiración de un temporizador en modo simple relativo. Para ello se crea un thread programado para ejecutar una función cuando dicho temporizador expire. Los intervalos de expiración se definen igual que en el análisis 2 (*nanosleep()*), para los dos tipos de relojes. El tiempo que se mide es el correspondiente a la activación del temporizador y su posterior expiración con la función programada para ello.

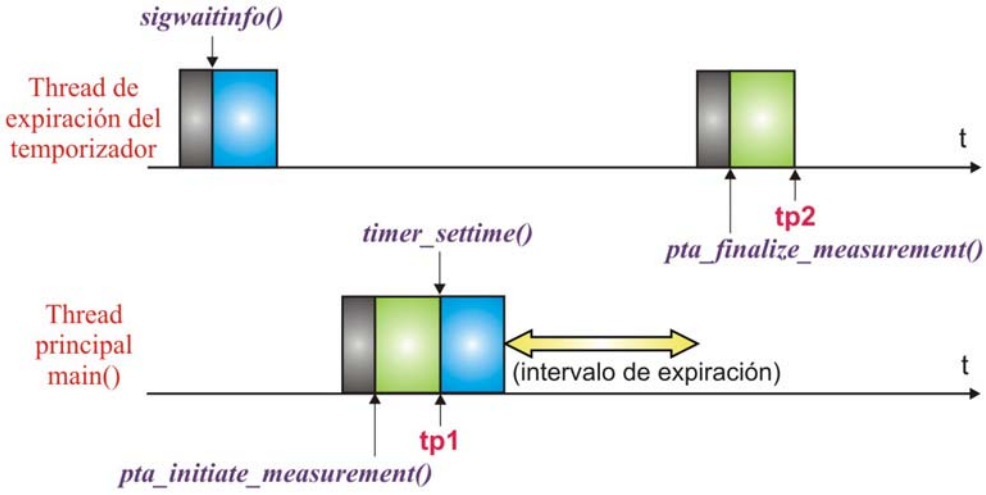

Figura 3.18 Gestión del tiempo: Temporizador en modo simple relativo

Análisis 5 y 6: Intercambio de ejecución de threads con un *"delay"* de alta prioridad

Este caso analiza el tiempo que tarda en realizarse la activación de un thread de baja prioridad, cuando se ejecuta un retraso (*delay*) en un thread de alta prioridad. El tiempo que se mide es el correspondiente a la ejecución de suspensión del thread de mayor prioridad y la activación del thread de menor prioridad, tal y como se muestra en la figura siguiente. Las operaciones de suspensión consideradas son *nanosleep()* y *clock_nanosleep()* (para los dos tipos de reloj). Ver Figura 3.19.

Análisis 7 y 8: Intercambio de ejecución de threads con un *"delay"* de baja prioridad

En este caso se mide únicamente sobre el thread de baja prioridad. Este coge constantemente la hora, mediante *clock_gettime()*. El thread de alta prioridad realiza una suspensión y luego se activa de nuevo. El tiempo que se mide en este caso es justamente el tiempo que está activo el thread de alta prioridad, es decir, el intervalo de tiempo durante el cual el thread de baja prioridad no puede leer el reloj. Este procedimiento se evalúa con las funciones *nanosleep()* y *clock_nanosleep()*, para los dos tipos de reloj. La figura 3.20. explica gráficamente el proceso.

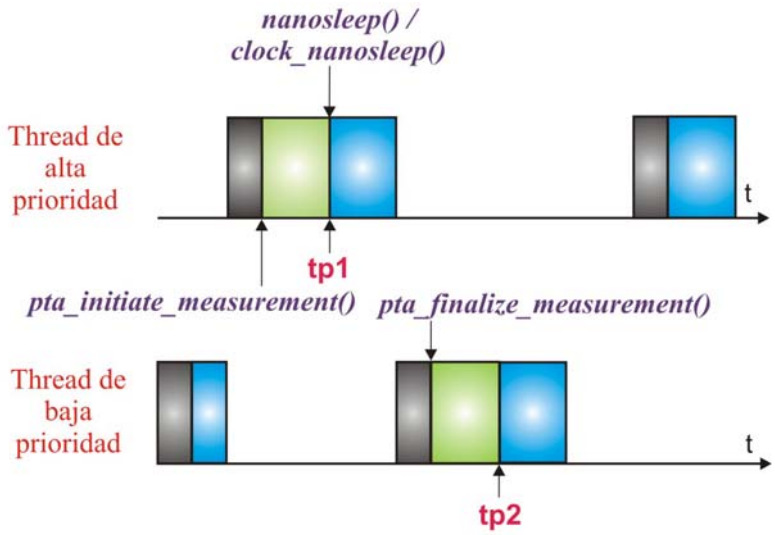

Figura 3.19 Gestión del tiempo: Intercambio de threads con delay de alta prioridad

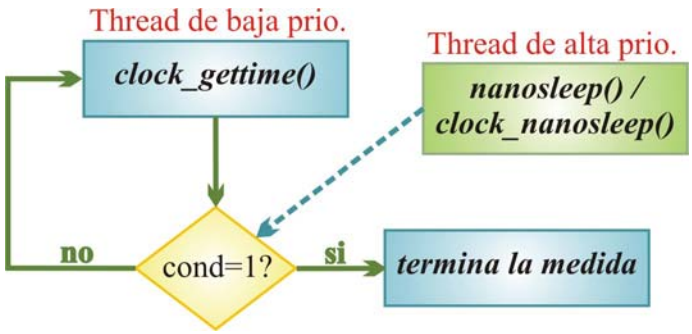

Figura 3.20 Gestión del tiempo: Intercambio de threads con delay de baja prioridad

3.5. Experimentos para caracterizar las señales de tiempo real

3.5.1. Datos

Procediendo de la misma forma que en los experimentos anteriores, en primer lugar se define el número de grupos de medidas que van a realizarse sobre este mecanismo de sincronización, para posteriormente definir un array con el número de estructuras de datos, **pta_time_data**, necesario para llevarlas a cabo. Igualmente, son definidas otras variables necesarias para la realización de este experimento, como son el número de threads que se ejecutarán simultáneamente, la estructura de datos que comparten estos threads, o el nombre del experimento para su posterior representación en pantalla. Estos parámetros toman los valores que siguen a continuación:

- Número de grupos de medidas (**num_meas_signal**): 8; es el número de casos considerados para llevar a cabo este análisis.

- Número de threads que se ejecutan simultaneamente (**th_wait**): 10; por el mismo motivo que en el resto de experimentos ya planteados.

Para determinar la prioridad de los threads se hace uso del protocolo de planificación SHED_FIFO. Además, se siguen las mismas características que en todos los experimentos anteriores:

- El thread principal, **main()**, es el de mayor prioridad, para garantizar que se creen a tiempo los threads necesarios para los diferentes casos.
- Los threads creados para la realización de los experimentos se crean con los siguientes atributos:
    - PTHREAD_EXPLICIT_SCHED: para que poder aplicar diferentes atributos a los threads, y que estos no los hereden del thread principal.
    - PTHREAD_CREATE_JOINABLE: para la espera sincronizada en la terminación de los threads.

Cabe mencionar en este punto que también han sido programados los procedimientos de análisis para la espera de señales con limitación de tiempo (*pthread_signal_timedwait()*), pero dado que este mecanismo no ha sido implementado aún en la plataforma sobre la que se va a llevar a cabo la realización práctica, MaRTE OS, no será detallado en esta memoria, aunque su código está incluido en los programas.

El siguiente paso es determinar la estructura de datos que compartirán los threads en este experimento. Dicha estructura tendrá los campos que a continuación se detallan:

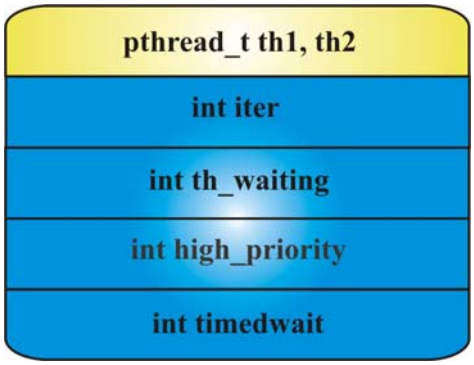

Figura 3.21 Señales: Estructura de datos para los threads

3.5.2. Procedimiento de medida

Este experimento concreto se divide en ocho casos que evalúan diferentes posibilidades que pueden darse en el manejo de señales de tiempo real. A continuación se describe cada uno de ellos:

Análisis 1: Envío de una señal *(kill)* y activación del thread de alta prioridad que la espera

Se consideran dos threads de diferentes prioridades. El de mayor prioridad espera la señal que ha de enviarle el de menor prioridad. Se evalúa el tiempo que tarda en activarse el thread que espera desde que le es enviada la señal.

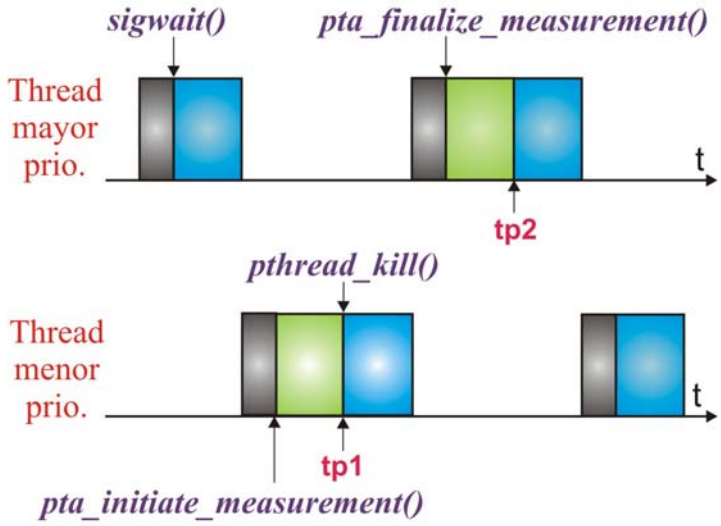

Figura 3.22 Señales: Envío (kill) + activación del thread que espera

Análisis 2: Envío de una señal *(kill)* y activación del thread de alta prioridad que la espera, cuando hay más threads esperando

Ahora en lugar de un thread que espera la señal, se consideran los que hayan sido definidos en **"th_wait"**, que en este experimento son 10. De entre ellos, solo el de mayor prioridad recibirá dicha señal. Al igual que antes, el de menor prioridad es el que realiza el envío.

Análisis 3: Envío de una señal *(sigqueue)* y activación del thread de alta prioridad que espera *(sigwaitinfo)*

Este caso es una variación del caso número 1, en el que se han cambiado las operaciones que se realizan con las señales de tiempo real, aunque se mantiene la misma filosofía de procedimiento. Se opta por la espera a la señal con información adicional con *sigwaitinfo()* y el envío mediante *sigqueue()*.

Análisis 4: Envío de una señal *(sigqueue)* y activación del thread de alta prioridad que espera *(sigwaitinfo)*, cuando hay más threads esperando

Se considera, al igual que en el apartado anterior, el envío y la recepción de señales mediante *sigqueue()* y *sigwaitinfo()*. Aunque ahora, son varios threads los que esperan. De entre ellos, el de mayor prioridad recibirá la señal, que será enviada por el de menor prioridad.

Análisis 5: Envío de una señal *(kill)* con un thread de menor prioridad que espera

Regresamos de nuevo al planteamiento realizado en el primero de los casos. Pero en este caso se invierten las prioridades de los threads que realizan las funciones de envío y recepción, de modo que el que envía tenga mayor prioridad que el que recibe. El tiempo que se evalúa es únicamente el del envío, con pthread_kill().

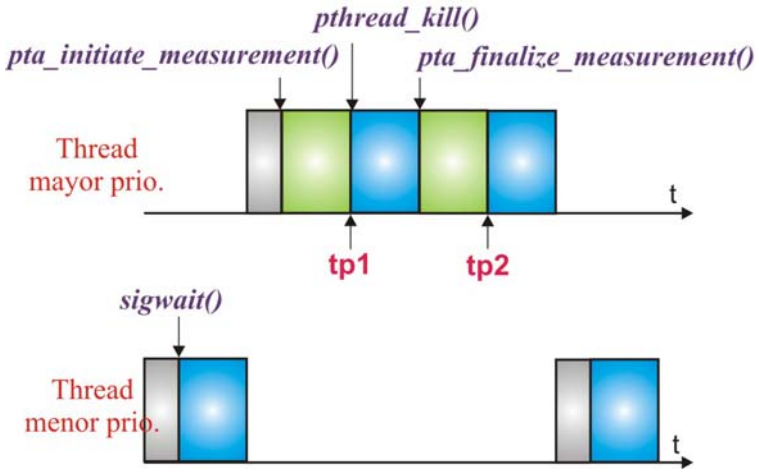

Figura 3.23 Señales: Envío de una señal (kill)

Análisis 6: Envío de una señal *(kill)* con varios varios threads de menor prioridad que esperan

Al igual que se ha ido considerando en el resto de los experimentos, se evalúa el caso anterior ampliando el número de threads implicados en el proceso. Ahora contamos con el número de threads definido en **"th_wait"** (que es 10). El thread de mayor prioridad realiza el envío de la señal. De los restantes, el que goce de mayor prioridad será el que la reciba.

Análisis 7 y 8: Envío de una señal *(sigqueue)* con uno o varios threads de menor prioridad que esperan *(sigwaitinfo)*

Los últimos casos que se evalúan en este experimento consisten en sustituir las operaciones de envío y espera definidos en los dos casos anteriores, por el envío mediante *sigqueue()* y la espera con información adicional con *sigwaitinfo()*. Se analiza el tiempo que toma *sigqueue()* en los dos casos descritos (con un thread o con el número indicado por **"th_wait"**, establecido a 10 para esta serie de experimentos).

3.6. Experimentos para caracterizar la gestión de memoria: *(malloc – free)*

3.6.1. Datos

Este experimento consiste en reservar y liberar bloques de memoria de diferentes tamaños, con el fin de determinar el tiempo que tardan en ejecutarse las funciones *malloc()* y *free()*, así como determinar la máxima capacidad de memoria que puede ser reservada.

Para analizar el efecto sobre el tiempo de trabajar con diferentes tamaños de memoria se realizarán dos experimentos diferentes, en los que la única diferencia es el tamaño de los bloques que se reservan y liberan. A continuación se muestra una tabla con los valores de lo bloques que se tratarán en cada uno de los experimentos:

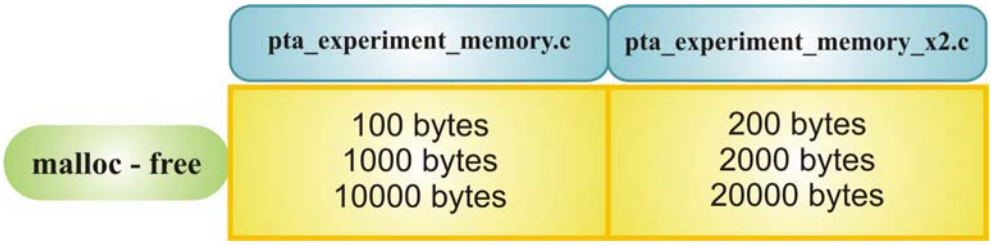

Figura 3.24 Gestión de memoria: Tamaño de los bloques

De este modo, el número de medidas que se realiza en cada uno de los experimentos, así como el resto de las variables necesarias para llevar a cabo el proceso son:

- Número de grupos de medidas (**num_meas_memory**): 6; para las dos funciones en y cada uno de los tamaños mostrados en la tabla anterior.
- Número de tamaños para los bloques (**block_sizes**): 3, tal y como se puede ver en la tabla.

3.6.2. Procedimiento de medida

Una vez establecidos los parámetros necesarios para llevar a cabo el experimento, se detallará a continuación el proceso seguido:

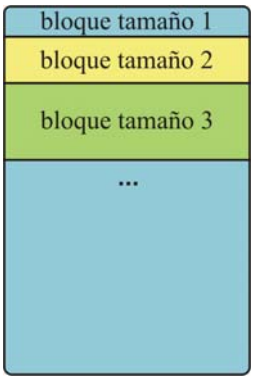

Figura 3.25 Gestión de memoria: Ejemplo de bloques de memoria

En primer lugar se reservan todos los bloques de medida de acuerdo a los tamaños y el número de iteraciones prefijados. Cada uno de los bloques tendrá asignado su puntero correspondiente, para posteriormente realizar en un segundo paso la liberación de la primera mitad del total de memoria reservada. El tercer paso consiste precisamente en volver a reservar esa misma mitad de la memoria, que previamente ha sido liberada. A continuación, se procederá a liberar la segunda parte de la memoria, que hasta ahora seguía reservada, en lo que constituye el paso cuatro. En el anteúltimo paso se volverá a reservar el bloque de memoria liberado en el caso anterior. Y finalmente, en un sexto paso, te liberará toda la memoria que había sido reservada. Ello nos permite comprobar el funcionamiento de los mecanismos de reserva y liberación de memoria bajo diversas circunstancias, y con diferentes grados de fragmentación de la memoria libre. Gráficamente el proceso podría ilustrarse como sigue:

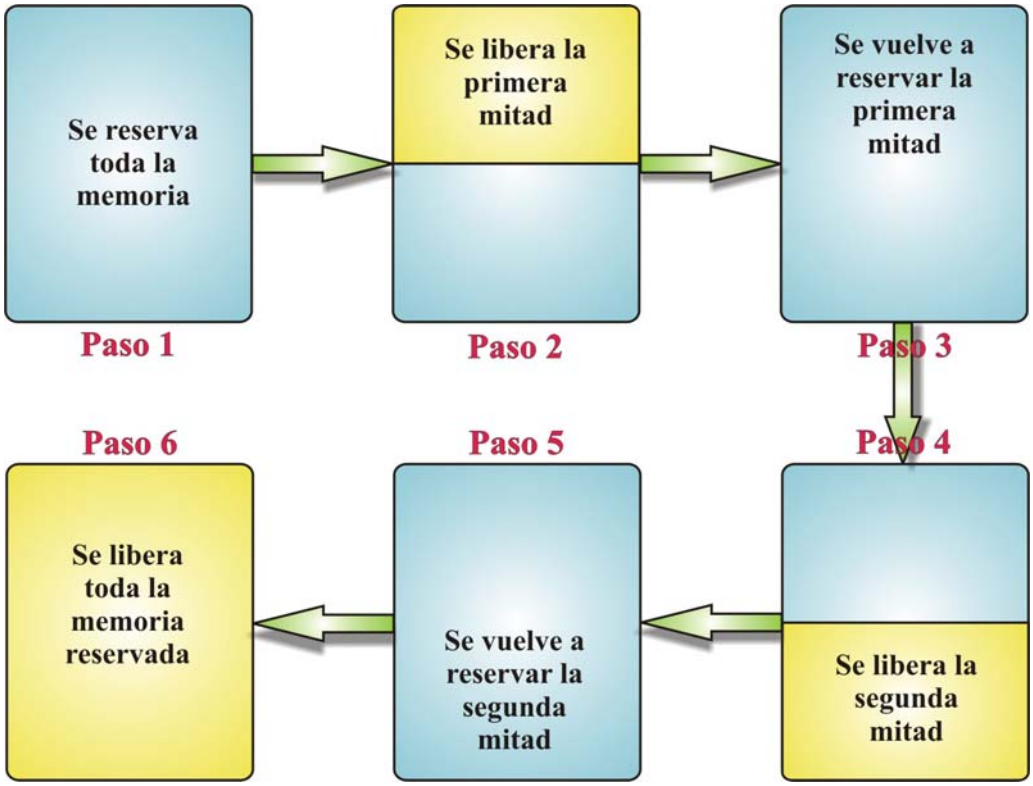

Figura 3.26 Gestión de memoria: Proceso de medida

3.7. Experimento para caracterizar la sobrecarga producida por las interrupciones

3.7.1. Datos

Este experimento consiste en ejecutar un código que detecte las interrupciones producidas durante un intervalo de tiempo, así como su duración. De modo que, son dos los parámetros necesarios que requieren configuración previa a su ejecución. Por un lado, ha de fijarse el intervalo de tiempo de ejecución de dicho código. Por otro, el número máximo de interrupciones que pueden detectarse, con el fin de limitar el tamaño del array que almacene los resultados que se obtienen.

Para cada una de las interrupciones detectadas se almacenará por un lado la fecha y hora a la que se produce dicha interrupción, así como su duración.

3.7.2. Procedimiento de medida

El primer paso en el procedimiento de medida consiste en evaluar el tiempo medio de la operación *clock_gettime()*, ya que será la herramienta empleada para detectar las interrupciones. Este tiempo medio se calcula a partir de 100 medidas de la

ejecución de dicha operación. Una vez caracterizada la herramienta de trabajo, se establece que será considerada como interrupción todo aquel retraso detectado que supere un umbral. Este umbral viene determinado por el doble del tiempo medio calculado de la ejecución de *clock_gettime()*, es decir, cualquier retraso superior al doble del tiempo medio de *clock_gettime()* será considerado una interrupción.

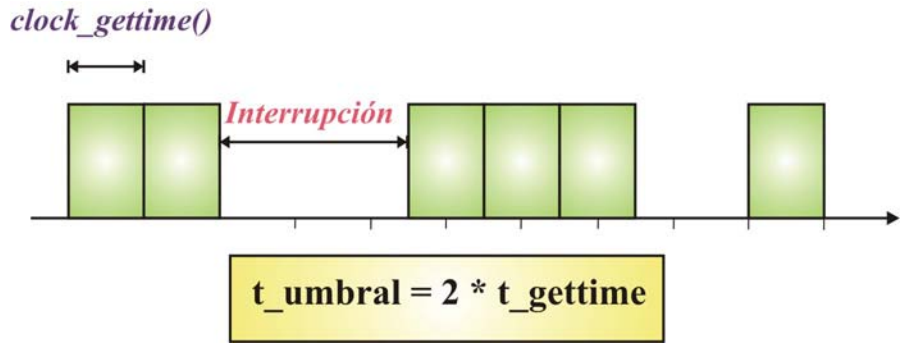

Figura 3.27 Interrupciones: Proceso de medida

El modo de calcular este retraso es medir continuamente la hora mediante *clock_gettime()* y determinar el intervalo de tiempo transcurrido entre el actual y el anterior. Cuando esta diferencia de tiempos sea mayor que el tiempo umbral se considerará interrupción, y se anotará la fecha y hora, y duración correspondiente en la tabla de resultados.

4.- Resultados y evaluación

4. Resultados y evaluación

4.1. Resultados para MaRTE OS

La aplicación práctica de los experimentos descritos en el apartado anterior, se ha llevado a cabo en un equipo del laboratorio que actúa como plataforma de ejecución de un sistema distribuido para controlar robots. Trabaja con un procesador Pentium III a 750 MHz, y está conectado por el puerto serie al equipo anfitrión, o sistema de desarrollo, mediante el puerto serie COM1.

La versión de MaRTE OS empleada para realizar los experimentos es la versión 1.56, junto con los compiladores gcc y Gnat GAP 1.0.

Para la realización de los experimentos, y tal y como se explicó en el apartado anterior, es posible especificar el número de iteraciones que han de realizarse para cada uno de ellos. De este modo, se especificará junto con cada uno de los resultados obtenidos, el número de iteraciones que han sido realizadas para la obtención de cada una de las medidas.

Los resultados para el primero de las plataformas de ejecución, el panel de botones, se mostrarán en la primera parte del capítulo. No se incluyen en esta memoria los resultados para la segunda de las plataformas de ejecución, por considerarse redundante e innecesario.

En primer lugar se muestran los resultados obtenidos para los mecanismos de sincronización. Tanto para los mutex, como para las variables condicionales, el número de iteraciones que se han realizado para tomar cada una de las medidas es 100. Los resultados que se han obtenido son los que se muestran a continuación:

Figura 4.1. Plataformas de ejecución sobre los que se realizan los experimentos

4.1.1 Mecanismos de sincronización

4.1.1.1. Mutex

Número de iteraciones realizadas para cada medida: 100.
Fecha de realización del experimento: 14/03/05.

MUTEX			
Inicialización de un mutex		***Destrucción de un mutex***	
Primera medida	6,679 µs	Primera medida	5,057 µs
Mínimo	0,741 µs	Mínimo	0,205 µs
Máximo	0,778 µs	Máximo	0,237 µs
Media	0,742 µs	Media	0,205 µs
Establecer valor del techo de prioridad		***Obtener valor del techo del prioridad***	
Primera medida	0,242 µs	Primera medida	1,031 µs
Mínimo	0,186 µs	Mínimo	0,053 µs
Máximo	0,213 µs	Máximo	0,054 µs
Media	0,186 µs	Media	0,053 µs
Protocolo con herencia de prioridad		***Protocolo con protección de prioridad***	
Bloqueo de un mutex libre (lock)			
Primera medida	16,302 µs	Primera medida	0,775 µs
Mínimo	0,857 µs	Mínimo	0,239 µs
Máximo	0,923 µs	Máximo	0,240 µs
Media	0,873 µs	Media	0,239 µs
Bloqueo de un mutex libre (trylock)			
Primera medida	5,688 µs	Primera medida	0,328 µs
Mínimo	0,658 µs	Mínimo	0,222 µs
Máximo	0,708 µs	Máximo	0,223 µs
Media	0,667 µs	Media	0,222 µs
Desbloqueo de un mutex en un thread de alta prioridad con cola de espera libre			
Primera medida	24,513 µs	Primera medida	0,421 µs
Mínimo	1,450 µs	Mínimo	0,243 µs
Máximo	1,521 µs	Máximo	0,244 µs
Media	1,460 µs	Media	0,243 µs
Desbloqueo de un mutex en un thread de baja prioridad y activación de un thread de mayor prioridad que está esperando el mutex			
Primera medida	13,235 µs	Primera medida	5,014 µs
Mínimo	4,784 µs	Mínimo	4,944 µs
Máximo	4,866 µs	Máximo	4,984 µs
Media	4,818 µs	Media	4,965 µs
Desbloqueo de un mutex en un thread de baja prioridad y activación de un thread de mayor prioridad que está esperando el mutex (con 10 threads esperándolo)			
Primera medida	4,458 µs	Primera medida	4,282 µs
Mínimo	4,373 µs	Mínimo	4,282 µs

Máximo	4,403 µs	Máximo	4,338 µs
Media	4,387 µs	Media	4,333 µs

Tabla 4.1. "Plataforma de ejecución 1": Resultados obtenidos para los mutex

4.1.1.2. Variables Condicionales

Número de iteraciones realizadas para cada medida: 100.
Fecha de realización del experimento: 14/03/05.

VARIABLES CONDICIONALES			
Inicialización de una variable condicional		***Destrucción de una variable condicional***	
Primera medida	6,380 µs	Primera medida	3,153 µs
Mínimo	1,556 µs	Mínimo	0,213 µs
Máximo	1,682 µs	Máximo	0,261 µs
Media	1,574 µs	Media	0,214 µs
Señalización (signal) en un thread de baja prioridad + activación del thread de alta prioridad que espera la condición (wait)		***Señalización (signal) en un thread de baja prioridad + activación del thread de alta prioridad que espera la condición (wait) (con 10 threads esperando)***	
Primera medida	23,476 µs	Primera medida	7,556 µs
Mínimo	5,320 µs	Mínimo	5,340 µs
Máximo	6,014 µs	Máximo	5,394 µs
Media	5,343 µs	Media	5,366 µs
Señalización (broadcast) en un thread de baja prioridad + activación del thread de alta prioridad que espera la condición (timedwait)		***Señalización (broadcast) en un thread de baja prioridad + activación del thread de alta prioridad que espera la condición (timedwait) (con 10 threads esperando)***	
Primera medida	16,643 µs	Primera medida	8,580 µs
Mínimo	6,395 µs	Mínimo	6,321 µs
Máximo	6,473 µs	Máximo	6,457 µs
Media	6,409 µs	Media	6,375 µs
Señalización (signal) en un thread de alta prioridad cuando un thread de menor prioridad espera la condición (wait)		***Señalización (signal) en un thread de alta prioridad cuando un thread de menor prioridad espera la condición (wait) (con 10 threads esperando)***	
Primera medida	1,848 µs	Primera medida	1,870 µs
Mínimo	0,318 µs	Mínimo	0,318 µs
Máximo	0,541 µs	Máximo	0,560 µs
Media	0,321 µs	Media	0,321 µs

Señalización (broadcast) en un thread de alta prioridad cuando un thread de menor prioridad espera la condición (timedwait)		***Señalización (broadcast) en un thread de alta prioridad cuando un thread de menor prioridad espera la condición (timedwait) (con 10 threads esperando)***	
Primera medida	2,884 µs	Primera medida	3,009 µs
Mínimo	0,277 µs	Mínimo	0,277 µs
Máximo	0,499 µs	Máximo	0,514 µs
Media	0,285 µs	Media	0,286 µs
Espera fallida (timedwait) en un thread de baja prioridad cuando espera la condición de un thread de alta prioridad			
Primera medida	2,274 µs		
Mínimo	1,130 µs		
Máximo	1,142 µs		
Media	1,135 µs		

Tabla 4.2 "Plataforma de ejecución 1": Resultados obtenidos para las variables condicionales

4.1.2. Gestión del tiempo

Número de iteraciones realizadas para cada medida: 100.
Fecha de realización del experimento: 14/03/05.

GESTIÓN DEL TIEMPO			
Resolución del reloj (Tiempo Real)		***Resolución del reloj (Monotónico)***	
Máxima Diferencia	597 ns	Máxima Diferencia	594 ns
Resolución media	1 ns	Resolución media	1 ns
Delay de alta resolución: nanosleep			
Intervalo: 1 µsg		***Intervalo: 10 µsg***	
Primera medida	18,078 µs	Primera medida	1,156 µs
Mínimo	1,144 µs	Mínimo	1,144 µs
Máximo	1,219 µs	Máximo	1,145 µs
Media	1,145 µs	Media	1,144 µs
Intervalo: 100 µsg		***Intervalo: 1000 µsg***	
Primera medida	104,715 µs	Primera medida	1004,134 µs
Mínimo	104,088 µs	Mínimo	1004,090 µs
Máximo	104,705 µs	Máximo	1004,127 µs
Media	104,114 µs	Media	1004,106 µs
Intervalo: 10000 µsg			
Primera medida	10004,089 µs		
Mínimo	10004,060 µs		

Máximo	10004,107 µs		
Media	10004,078 µs		
Delay de alta resolución: clock_nanosleep (en modo absoluto)			
Reloj de tiempo real		***Reloj monotónico***	
Intervalo: 100001 µsg		***Intervalo: 100001 µsg***	
Primera medida	100012,653 µs	Primera medida	100003,910 µs
Mínimo	100003,969 µs	Mínimo	100003,752 µs
Máximo	100004,177 µs	Máximo	100003,946 µs
Media	100004,027 µs	Media	100003,887 µs
Intervalo: 200010 µsg		***Intervalo: 200010 µsg***	
Primera medida	200012,853 µs	Primera medida	200012,670 µs
Mínimo	200012,613 µs	Mínimo	200012,451 µs
Máximo	200012,822 µs	Máximo	200012,694 µs
Media	200012,740 µs	Media	200012,599 µs
Intervalo: 300100 µsg		***Intervalo: 300100 µsg***	
Primera medida	300102,596 µs	Primera medida	300102,433 µs
Mínimo	300102,376 µs	Mínimo	300102,165 µs
Máximo	300102,570 µs	Máximo	300102,416 µs
Media	300102,465 µs	Media	300102,310 µs
Intervalo: 401000 µsg		***Intervalo: 401000 µsg***	
Primera medida	401002,245 µs	Primera medida	401002,103 µs
Mínimo	401002,049 µs	Mínimo	401001,896 µs
Máximo	401002,293 µs	Máximo	401002,137 µs
Media	401002,167 µs	Media	401002,032 µs
Intervalo: 510000 µsg		***Intervalo: 510000 µsg***	
Primera medida	510001,903 µs	Primera medida	510001,860 µs
Mínimo	510001,751 µs	Mínimo	510001,607 µs
Máximo	510001,978 µs	Máximo	510001,836 µs
Media	510001,861 µs	Media	510001,727 µs
Temporizador en modo simple relativo			
Reloj de tiempo real		***Reloj monotónico***	
Intervalo: 1 µsg		***Intervalo: 1 µsg***	
Primera medida	61,662 µs	Primera medida	29,107 µs
Mínimo	29,071 µs	Mínimo	29,107 µs
Máximo	29,079 µs	Máximo	29,126 µs
Media	29,075 µs	Media	29,118 µs
Intervalo: 10 µsg		***Intervalo: 10 µsg***	
Primera medida	30,385 µs	Primera medida	29,086 µs
Mínimo	29,123 µs	Mínimo	29,080 µs
Máximo	29,130 µs	Máximo	29,089 µs
Media	29,125 µs	Media	29,084 µs
Intervalo: 100 µsg		***Intervalo: 100 µsg***	
Primera medida	109,147 µs	Primera medida	109,176 µs

Mínimo	109,137 µs	Mínimo	109,176 µs
Máximo	109,153 µs	Máximo	109,188 µs
Media	109,146 µs	Media	109,184 µs
Intervalo: 1000 µsg		***Intervalo: 1000 µsg***	
Primera medida	1009,186 µs	Primera medida	1009,147 µs
Mínimo	1009,186 µs	Mínimo	1009,136 µs
Máximo	1009,198 µs	Máximo	1009,146 µs
Media	1009,193 µs	Media	1009,141 µs
Intervalo: 10000 µsg		***Intervalo: 10000 µsg***	
Primera medida	10009,123 µs	Primera medida	10009,150 µs
Mínimo	10009,109 µs	Mínimo	10009,147 µs
Máximo	10009,117 µs	Máximo	10009,162 µs
Media	10009,114 µs	Media	10009,155 µs
Delay en un thread de alta prioridad+activación de un thread de menor prioridad (nanosleep)		***Repercusión de un delay en un thread de alta prioridad sobre un thread de baja prioridad (nanosleep)***	
Primera medida	4,210 µs	Primera medida	0,585 µs
Mínimo	3,973 µs	Mínimo	0,585 µs
Máximo	4,049 µs	Máximo	0,586 µs
Media	4,007 µs	Media	0,585 µs
Delay en un thread de alta prioridad+activación de un thread de menor prioridad (clock_nanosleep abs.)			
Reloj de tiempo real		***Reloj monotónico***	
Primera medida	4,664 µs	Primera medida	4,729 µs
Mínimo	4,576 µs	Mínimo	4,327 µs
Máximo	4,622 µs	Máximo	4,369 µs
Media	4,592 µs	Media	4,340 µs
Repercusión de un delay en un thread de alta prioridad sobre un thread de baja prioridad (clock_nanosleep abs.)			
Reloj de tiempo real		***Reloj monotónico***	
Primera medida	0,585 µs	Primera medida	0,585 µs
Mínimo	0,585 µs	Mínimo	0,585 µs
Máximo	0,586 µs	Máximo	0,586 µs
Media	0,585 µs	Media	0,585 µs

Tabla 4.3 "Plataforma de ejecución 1": Resultados obtenidos para la gestión del tiempo

4.1.3. Señales de tiempo real

Número de iteraciones realizadas para cada medida: 100.
Fecha de realización del experimento: 14/03/05.

SEÑALES DE TIEMPO REAL			
Señalización (kill)+recepción de la señal en un thread de mayor prioridad que espera (sigwait)		***Señalización (kill)+recepción de la señal en un thread de mayor prioridad que espera (sigwait) (con 10 threads esperando)***	
Primera medida	22,920 µs	Primera medida	4,755 µs
Mínimo	3,386 µs	Mínimo	3,521 µs
Máximo	3,428 µs	Máximo	3,569 µs
Media	3,409 µs	Media	3,553 µs
Señalización (sigqueue)+recepción de la señal en un thread de mayor prioridad que espera (sigwaitinfo)		***Señalización (sigqueue)+recepción de la señal en un thread de mayor prioridad que espera (sigwaitinfo) (con 10 threads esperando)***	
Primera medida	11,136 µs	Primera medida	3,915 µs
Mínimo	3,748 µs	Mínimo	3,828 µs
Máximo	3,847 µs	Máximo	3,933 µs
Media	3,825 µs	Media	3,856 µs
Señalización (kill) en un thread de alta prioridad cuando espera la señal (sigwait) un thread de menor prioridad		***Señalización (kill) en un thread de alta prioridad cuando espera la señal (sigwait) un thread de menor prioridad (con 10 threads esperando)***	
Primera medida	2,511 µs	Primera medida	2,423 µs
Mínimo	2,349 µs	Mínimo	2,357 µs
Máximo	2,451 µs	Máximo	2,466 µs
Media	2,379 µs	Media	2,402 µs
Señalización (sigqueue) en un thread de alta prioridad cuando espera la señal (sigwaitinfo) un thread de menor prioridad		***Señalización (sigqueue) en un thread de alta prioridad cuando espera la señal (sigwaitinfo) un thread de menor prioridad (con 10 threads esperando)***	
Primera medida	2,796 µs	Primera medida	2,828 µs
Mínimo	2,720 µs	Mínimo	2,824 µs
Máximo	2,740 µs	Máximo	2,870 µs
Media	2,730 µs	Media	2,836 µs

Tabla 4.4 "Plataforma de ejecución 1": Resultados obtenidos para las señales de tiempo real

4.1.4. Gestión de memoria dinámica

Para llevar a cabo la caracterización de la gestión de memoria se procedió a reservar y liberar bloques de diferentes tamaños. Los resultados obtenidos son los que se muestran en la siguiente tabla:

Número de iteraciones realizadas para cada medida: 1000.
Fecha de realización del experimento: 14/03/05.

	GESTIÓN DE MEMORIA			
		100 bytes	1000 bytes	10000 bytes
malloc()	Primera medida	10,070 µs	0,878 µs	1,118 µs
	Mínimo	0,658 µs	0,664 µs	0,637 µs
	Máximo	1,542 µs	1,400 µs	1,418 µs
	Media	0,844 µs	0,850 µs	0,878 µs
free()	Primera medida	5,425 µs	2,514 µs	0,726 µs
	Mínimo	0,414 µs	0,409 µs	0,422 µs
	Máximo	1,062 µs	1,026 µs	1,642 µs
	Media	0,431 µs	0,444 µs	0,467 µs

Tabla 4.5 "Plataforma de ejecución 1": Resultados obtenidos para la gestión de memoria

Número de iteraciones realizadas para cada medida: 500.
Fecha de realización del experimento: 15/03/05.

	GESTIÓN DE MEMORIA x2			
		200 bytes	2000 bytes	20000 bytes
malloc()	Primera medida	11,267 µs	0,692 µs	1,117 µs
	Mínimo	0,676 µs	0,675 µs	0,690 µs
	Máximo	1,497 µs	1,420 µs	1,444 µs
	Media	0,866 µs	0,861 µs	0,863 µs
free()	Primera medida	5,774 µs	2,512 µs	0,710 µs
	Mínimo	0,434 µs	0,438 µs	0,442 µs
	Máximo	0,490 µs	0,704 µs	1,932 µs
	Media	0,465 µs	0,465 µs	0,474 µs

Tabla 4.6 "Plataforma de ejecución 1": Resultados obtenidos para la gestión de memoria 2

4.1.5. Sobrecarga producida por las interrupciones

Fecha de realización del experimento: 14/03/05.

Para la realización de este experimento se varió la configuración del parámetro de tiempo de ejecución, para observar las interrupciones producidas en diferentes intervalos de tiempo. De esta forma los parámetros del experimentos quedan definidos así:

Tiempo de ejecución, en segundos: 1sg – 10 sg – 100 sg
Tamaño máximo de la tabla de resultados: 200 interrupciones

Tal y como se explicó en el apartado anterior, la forma de detectar las interrupciones es calcular los intervalos de tiempo por encima de un determinado umbral (dos veces el tiempo que se tarda en coger el reloj).

La limitación de este experimento vino impuesta por el tiempo de ejecución, ya que en ninguno de los casos se llegó a alcanzar el tamaño máximo de la tabla de resultados. Es más, en los dos primeros caso, 1 sg y 10 sg, no han sido detectadas interrupciones por encima de ese umbral temporal, mientras que para el último caso (100 sg) se detectó únicamente una interrupción de 11,025 μs.

4.3. Resultados para Linux

Las medidas sobre Linux se han realizado sobre un equipo con un procesador Pentium 4, a 2.4 GHz, corriendo bajo Fedora Core 3. La ejecución de los programas de medida se realizado en el terminar mientras se estaban ejecutando al mismo iempo el escritor de OpenOffice, el navegador Mozilla, y el Amsn.

4.3.1. Mecanismos de sincronización

4.3.1.1. Mutex

Número de iteraciones realizadas para cada medida: 100.
Fecha de realización del experimento: 14/03/05.

MUTEX			
Inicialización de un mutex		***Destrucción de un mutex***	
Primera medida	2,000 µs	Primera medida	1,000 µs
Mínimo	0,000 µs	Mínimo	0,000 µs
Máximo	2,000 µs	Máximo	14,000 µs
Media	0,888 µs	Media	1,030 µs
Bloqueo de un mutex libre (lock)		***Bloqueo de un mutex libre (trylock)***	
Primera medida	2,000 µs	Primera medida	2,000 µs
Mínimo	0,000 µs	Mínimo	0,000 µs
Máximo	2,000 µs	Máximo	15,000 µs
Media	0,939 µs	Media	1,080 µs
Desbloqueo de un mutex en un thread de alta prioridad cuando con cola de espera libre			
Primera medida	1,000 µs		
Mínimo	0,000 µs		
Máximo	12,000 µs		
Media	1,040 µs		
Desbloqueo de un mutex en un thread de baja prioridad y activación de un thread de mayor prioridad que está esperando el mutex		***Desbloqueo de un mutex en un thread de baja prioridad y activación de un thread de mayor prioridad que está esperando el mutex (con 10 threads esperándolo)***	
Primera medida	32,000 µs	Primera medida	32,000 µs
Mínimo	30,000 µs	Mínimo	31,000 µs
Máximo	48,000 µs	Máximo	950,000 µs
Media	35,545 µs	Media	42,808 µs

Tabla 4.7 Linux OS: Resultados obtenidos para los mutex

4.3.1.2. Variables Condicionales

Número de iteraciones realizadas para cada medida: 100.
Fecha de realización del experimento: 15/03/05.

VARIABLES CONDICIONALES			
Inicialización de una variable condicional		***Destrucción de una variable condicional***	
Primera medida	2,000 µs	Primera medida	1,000 µs
Mínimo	0,000 µs	Mínimo	0,000 µs
Máximo	2,000 µs	Máximo	2,000 µs
Media	0,878 µs	Media	0,888 µs
Señalización (signal) en un thread de baja prioridad + activación del thread de alta prioridad que espera la condición (wait)		***Señalización (signal) en un thread de baja prioridad + activación del thread de alta prioridad que espera la condición (wait) (con 10 threads esperando)***	
Primera medida	15,000 µs	Primera medida	14,000 µs
Mínimo	10,000 µs	Mínimo	10,000 µs
Máximo	23,000 µs	Máximo	11,000 µs
Media	10,636 µs	Media	10,727 µs
Señalización (broadcast) en un thread de baja prioridad + activación del thread de alta prioridad que espera la condición (timedwait)		***Señalización (broadcast) en un thread de baja prioridad + activación del thread de alta prioridad que espera la condición (timedwait) (con 10 threads esperando)***	
Primera medida	11,000 µs	Primera medida	12,000 µs
Mínimo	6,000 µs	Mínimo	6,000 µs
Máximo	7,000 µs	Máximo	7,000 µs
Media	6,868 µs	Media	6,818 µs
Señalización (signal) en un thread de alta prioridad cuando un thread de menor prioridad espera la condición (wait)		***Señalización (signal) en un thread de alta prioridad cuando un thread de menor prioridad espera la condición (wait) (con 10 threads esperando)***	
Primera medida	3,000 µs	Primera medida	2,000 µs
Mínimo	0,000 µs	Mínimo	0,000 µs
Máximo	19,000 µs	Máximo	2,000 µs
Media	1,101 µs	Media	0,909 µs
Señalización (broadcast) en un thread de alta prioridad cuando un thread de menor prioridad espera la condición (timedwait)		***Señalización (broadcast) en un thread de alta prioridad cuando un thread de menor prioridad espera la condición (timedwait) (con 10 threads esperando)***	
Primera medida	3,000 µs	Primera medida	2,000 µs
Mínimo	0,000 µs	Mínimo	0,000 µs
Máximo	19,000 µs	Máximo	16,000 µs
Media	1,161 µs	Media	1,111 µs

Espera fallida (timedwait) en un thread de baja prioridad cuando espera la condición de un thread de alta prioridad	
Primera medida	4,000 µs
Mínimo	3,000 µs
Máximo	15,000 µs
Media	4,090 µs

Tabla 4.8 Linux OS: Resultados obtenidos para las variables condicionales

4.3.2. Gestión del tiempo

Número de iteraciones realizadas para cada medida: 100.
Fecha de realización del experimento: 14/03/04.

GESTIÓN DEL TIEMPO			
Resolución del reloj (Tiempo Real)		***Resolución del reloj (Monotónico)***	
Máxima Diferencia	3,000 µs	Máxima Diferencia	3,000 µs
Resolución media	999,848 µs	Resolución media	999,848 µs
Delay de alta resolución: nanosleep			
Intervalo: 1 µsg		***Intervalo: 10 µsg***	
Primera medida	1931,000µs	Primera medida	1995,000 µs
Mínimo	1701,000 µs	Mínimo	1578,000 µs
Máximo	2306,000 µs	Máximo	2415,000 µs
Media	1995,222 µs	Media	1995,282 µs
Intervalo: 100 µsg		***Intervalo: 1000 µsg***	
Primera medida	1996,000 µs	Primera medida	2978,000 µs
Mínimo	1852,000 µs	Mínimo	2949,000 µs
Máximo	2104,000 µs	Máximo	3048,000 µs
Media	1995,313 µs	Media	2994,838 µs
Intervalo: 10000 µsg			
Primera medida	11992,000 µs		
Mínimo	11852,000 µs		
Máximo	12135,000 µs		
Media	11993,595 µs		
Delay de alta resolución: clock_nanosleep (en modo absoluto)			
Reloj de tiempo real		***Reloj monotónico***	
Intervalo: 100001 µsg		***Intervalo: 100001 µsg***	
Primera medida	100983,000 µs	Primera medida	101972,000 µs
Mínimo	100983,000 µs	Mínimo	101971,000 µs
Máximo	101997,000 µs	Máximo	101979,000 µs
Media	101975,343 µs	Media	101975,282 µs

Intervalo: 200010 µsg		***Intervalo: 200010 µsg***	
Primera medida	201959,000 µs	Primera medida	201961,000 µs
Mínimo	201950,000 µs	Mínimo	201919,000 µs
Máximo	201971,000 µs	Máximo	201984,000 µs
Media	201960,151 µs	Media	201959,909 µs
Intervalo: 300100 µsg		***Intervalo: 300100 µsg***	
Primera medida	301943,000 µs	Primera medida	301945,000 µs
Mínimo	301845,000 µs	Mínimo	301825,000 µs
Máximo	302045,000 µs	Máximo	317033,000 µs
Media	301944,959 µs	Media	302287,393 µs
Intervalo: 401000 µsg		***Intervalo: 401000 µsg***	
Primera medida	402936,000 µs	Primera medida	402925,000 µs
Mínimo	402920,000 µs	Mínimo	402107,000 µs
Máximo	403940,000 µs	Máximo	406291,000 µs
Media	402929,575 µs	Media	402958,606 µs
Intervalo: 510000 µsg		***Intervalo: 510000 µsg***	
Primera medida	511912,000 µs	Primera medida	511911,000 µs
Mínimo	511904,000 µs	Mínimo	511849,000 µs
Máximo	511923,000 µs	Máximo	511976,000 µs
Media	511912,959 µs	Media	511912,858 µs
Temporizador en modo simple relativo			
Reloj de tiempo real		***Reloj monotónico***	
Intervalo: 1 µsg		***Intervalo: 1 µsg***	
Primera medida	716,000 µs	Primera medida	960,000 µs
Mínimo	716,000 µs	Mínimo	802,000 µs
Máximo	982,000 µs	Máximo	981,000 µs
Media	932,040 µs	Media	933,909 µs
Intervalo: 10 µsg		***Intervalo: 10 µsg***	
Primera medida	672,000 µs	Primera medida	959,000 µs
Mínimo	672,000 µs	Mínimo	863,000 µs
Máximo	963,000 µs	Máximo	982,000 µs
Media	954,060 µs	Media	953,141 µs
Intervalo: 100 µsg		***Intervalo: 100 µsg***	
Primera medida	953,000 µs	Primera medida	960,000 µs
Mínimo	826,000 µs	Mínimo	775,000 µs
Máximo	1057,000 µs	Máximo	1057,000 µs
Media	956,606 µs	Media	956,090 µs
Intervalo: 1000 µsg		***Intervalo: 1000 µsg***	
Primera medida	1959,000 µs	Primera medida	1959,000 µs
Mínimo	1858,000 µs	Mínimo	1334,000 µs
Máximo	1983,000 µs	Máximo	1983,000 µs
Media	1950,575 µs	Media	1941,848 µs

Intervalo: 10000 μsg		*Intervalo: 10000 μsg*	
Primera medida	10954,000 μs	Primera medida	10972,000 μs
Mínimo	10854,000 μs	Mínimo	10809,000 μs
Máximo	10979,000 μs	Máximo	10985,000 μs
Media	10953,131 μs	Media	10951,474 μs

Tabla 4.9 Linux OS: Resultados obtenidos para la gestión del tiempo

4.3.3. Señales de tiempo real

Número de iteraciones realizadas para cada medida: 100.
Fecha de realización del experimento: 14/03/05.

SEÑALES DE TIEMPO REAL			
Señalización (kill)+recepción de la señal en un thread de mayor prioridad que espera (sigwait)		***Señalización (kill)+recepción de la señal en un thread de mayor prioridad que espera (sigwait) (con 10 threads esperando)***	
Primera medida	10,000 μs	Primera medida	10,000 μs
Mínimo	3,000 μs	Mínimo	4,000 μs
Máximo	37,000 μs	Máximo	22,000 μs
Media	4,699 μs	Media	6,202 μs
Señalización (sigqueue)+recepción de la señal en un thread de mayor prioridad que espera (sigwaitinfo)		***Señalización (sigqueue)+recepción de la señal en un thread de mayor prioridad que espera (sigwaitinfo) (con 10 threads esperando)***	
Primera medida	37,000 μs	Primera medida	7,000 μs
Mínimo	5,000 μs	Mínimo	6,000 μs
Máximo	27,000 μs	Máximo	114,000 μs
Media	5,717 μs	Media	9,383 μs
Señalización (kill) en un thread de alta prioridad cuando espera la señal (sigwait) un thread de menor prioridad		***Señalización (kill) en un thread de alta prioridad cuando espera la señal (sigwait) un thread de menor prioridad (con 10 threads esperando)***	
Primera medida	4,000 μs	Primera medida	4,000 μs
Mínimo	2,000 μs	Mínimo	3,000 μs
Máximo	18,000 μs	Máximo	10,000 μs
Media	2,989 μs	Media	3,676 μs
Señalización (sigqueue) en un thread de alta prioridad cuando espera la señal (sigwaitinfo) un thread de menor prioridad		***Señalización (sigqueue) en un thread de alta prioridad cuando espera la señal (sigwaitinfo) un thread de menor prioridad (con 10 threads esperando)***	
Primera medida	5,000 μs	Primera medida	7,000 μs
Mínimo	3,000 μs	Mínimo	4,000 μs

Máximo	12,000 µs	Máximo	12,000 µs
Media	4,282 µs	Media	5,909 µs

Tabla 4.10 Linux OS: Resultados obtenidos para las señales de tiempo real

4.3.4. Gestión de memoria

Tal y como se detalló en el apartado anterior, para llevar a cabo la caracterización de la gestión de memoria se procedió a reservar y liberar bloques de diferentes tamaños. Los resultados obtenidos son los que se muestran en la siguiente tabla:

Número de iteraciones realizadas para cada medida: 1000.
Fecha de realización de los experimentos: 14/03/05.

		GESTIÓN DE MEMORIA		
		100 bytes	1000 bytes	10000 bytes
malloc()	First measurement	104,000 µs	1,000 µs	5,000 µs
	Minimum	0,000 µs	0,000 µs	0,000 µs
	Maximum	17,000 µs	23,000 µs	24,000 µs
	Average	1,170 µs	2,380 µs	6,386 µs
free()	First measurement	15,000 µs	1,000 µs	1,000 µs
	Minimum	0,000 µs	0,000 µs	0,000 µs
	Maximum	91,000 µs	22,000 µs	259,000 µs
	Average	1,202 µs	1,111 µs	1,426 µs

Tabla 4.11 Linux OS: Resultados obtenidos para la gestión de memoria

Número de iteraciones realizadas para cada medida: 500.

		GESTIÓN DE MEMORIA x2		
		200 bytes	2000 bytes	20000 bytes
malloc()	First measurement	51,000 µs	1,000 µs	7,000 µs
	Minimum	0,000 µs	0,000 µs	0,000 µs
	Maximum	17,000 µs	99,000 µs	26668,000 µs
	Average	1,274 µs	3,759 µs	59,579 µs
free()	First measurement	15,000 µs	1,000 µs	1,000 µs
	Minimum	0,000 µs	0,000 µs	0,000 µs
	Maximum	2,000 µs	16,000 µs	169,000 µs
	Average	0,977 µs	1,066 µs	1,517 µs

Tabla 4.12 Linux OS: Resultados obtenidos para la gestión de memoria 2

4.3.5. Sobrecarga producida por las interrupciones

Fecha de realización del experimento: 15/03/05.

La configuración de parámetros para la realización de este experimento se realizó tal y como se muestra a continuación:

Tiempo de ejecución, en segundos: 1sg
Tamaño máximo de la tabla de resultados: 200 interrupciones

Los resultados obtenidos, para este tiempo de ejecución, son los que se muestran a continuación en escala logarítimica, para una mejor visualización:

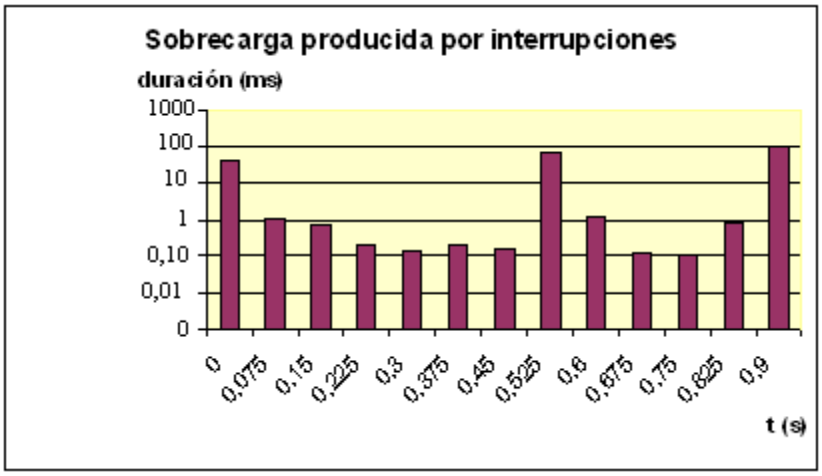

Figura 4.2. Linux: Resultados obtenidos para las interrupciones

4.4. Evaluación de los resultados obtenidos

La aplicación práctica de los experimentos se ha llevado a cabo sobre dos plataformas de desarrollo diferentes:

- Sistema de Tiempo Real Mínimo basado en el estándar descrito, con un procesador Pentium III @ 750 MHz, corriendo bajo MaRTE OS 1.56.

- Sistema de Tiempo Compartido en Linux, con un procesador Pentium IV @ 2.4 GHz corriendo bajo Fedora Core 3.

Como puede verse, la plataforma de ejecución de Linux es mucho más rápida que la empleada para MaRTE, dato que habrá de ser tenido en cuenta a la hora de comparar los tiempos.

Evaluación general:

- Como era previsible, los tiempos de la primera medida son bastante superiores al resto. Esto se debe a los efectos de carga de la caché y de la *"pipeline"* del procesador.

- Los tiempos mínimos de ejecución son menores en Linux, pero los tiempos máximos y medios habitualmente se mantienen por debajo en MaRTE.

- En general, las diferencias entre los tiempos mínimos y máximos son mucho menores en el sistema diseñado siguiendo los requerimientos de tiempo real. En Linux, en ocasiones, los tiempos se disparan. Por ejemplo, en una ocasión el desbloqueo de un mutex tardó casi un 1 ms, cuando la media es de 42 μs. En otra ocasión, la reserva de un bloque de memoria llevó casi 26 ms, cuando la media es de 59 μs. Como comparación, el mismo experimento en MaRTE, tiene un tiempo máximo de 1,44 μs y un promedio de 0,86 μs.

- En los dos sistemas, el tiempo de respuesta prácticamente no varía en función del número de threads que estén siendo ejecutados.

Para cada uno de los servicios evaluados:

- Mutexes: En MaRTE los tiempos de ejecución con el protocolo por protección de prioridad son claramente menores que con el protocolo de herencia de prioridad. Linux no tiene protocolos de prioridad.

    - En MaRTE, los tiempos promedio de toma y liberación de mutexes oscilan, según el caso, entre 0,23 μs y 5 μs.
    - En Linux estos tiempos son de 1 μs y 42 μs.

- Variables Condicionales: Los tiempos por señalización individual y por broadcast son semejantes, en ambos sistemas operativos.

  - En MaRTE, los tiempos promedio de señalización y espera de variables condicionales oscilan, según el caso, entre 0,23 μs y 5 μs.
  - En Linux estos tiempos son de 0,8 μs y 10,7 μs.

- Gestión del tiempo: En MaRTE la resolución media teórica de los relojes es de 1ns, mientras en Linux este valor alcanza 1 ms. Pero los valores reales son algo superiores, en realidad, para MaRTE OS se rondan los 600 ns mientras que para Linux se alcanzan los 3 ms.
  En relación a la ejecución de un temporizador en modo simple relativo, los tiempos promedio que se han conseguido son:
  - En MaRTE, para intervalos de suspensión menores de 20 μs (valor que es configurable) no se llega a realizar la función, sino que se ejecuta un *"yield"* en su lugar, por este motivo, se registra un valor de aproximadamente 29 μs.
  - En Linux estos tiempos de ejecución son muy superiores, alcanzándose valores de unos 900 μs cuando el intervalo de expiración es de tan solo 1 μs.

  Esto mismo sucede con los retrasos de alta resolución:
  - En MaRTE, dado que el intervalo mínimo de suspensión son 20 μs, no se realiza la suspensión para intervalos temporales menores. En tiempos promedios, la ejecución del *nanosleep()* ronda los 4 μs mientras que para *clock_nanosleep()* en modo absoluto este valor es cercano a los 2 μs.

- Señales de Tiempo Real: El tiempo para la señalización con *sigqueue()* es ligeramente superior a *pthread_kill()*. Esto mismo ocurre con los mecanismos de recepción, *sigwait()* y *sigwaitinfo()*.
  - En MaRTE, los tiempos de ejecución promedio oscilan entre 2,3 y 3,5 μs.
  - En Linux, estos tiempos sin embargo alcanzan valores de entre 3 y 9 μs.

- Gestión de memoria: En MaRTE los tiempos de reserva de bloques son independientes del tamaño solicitado. Los tiempos para la liberación de bloques son mucho menores.
  - En MaRTE, el valor promedio para reservar un bloque de memoria oscila entorno a los 0,86 μs, mientras que el valor promedio para la liberación de un bloque de memoria es 0,45 μs.
  - En Linux, estos valores si dependen del tamaño de memoria solicitado. Con bloques desde 100 a 20000 bytes, en reserva de memoria se obtienen tiempos promedio de 1 a 6 μs, y 1 a 1,5 μs para liberación.

- Sobrecarga producida por interrupciones: En MaRTE no ocurrían nunca prácticamente. En Linux la detección de interrupciones es continuada, ya que se trata de un sistema de tiempo compartido.

5.- Conclusiones y líneas futuras de trabajo

5. Conclusiones y líneas futuras de trabajo

El presente trabajo nos ha permitido estudiar las aplicaciones de tiempo real, que están tomando cada vez más relevancia en nuestra vida diaria, y podemos encontrarlas en muchos y variados entornos, desde aviones, vehículos, o trenes, procesos industriales, o hasta en nuestras casas, en electrodomésticos, reproductores de video y audio, o incluso en teléfonos móviles.

En particular el estudio se ha centrado en el "Sistema de Tiempo Real Mínimo" definido en el estándar POSIX.13, y que es el empleado habitualmente en aplicaciones empotradas como parte de un sistema más grande.

La caracterización temporal de los servicios que componen un sistema operativo de tiempo real es de importante utilidad en dos fases muy diferenciadas. Por un lado, constituye una información que facilita el análisis y la evaluación del sistema operativo en su fase de desarrollo, lo que permite comprobar el funcionamiento de los servicios que realiza, así como el rendimiento que presenta para diferentes implementaciones, y detectar posibles "puntos débiles" en el diseño. Por otro lado, esa evaluación del funcionamiento y del rendimiento facilita al usuario un conjunto de datos con los que puede realizar comparaciones entre diferentes sistemas operativos que ofrezca el mercado, o simplemente comprobar si uno en concreto cumple los requisitos temporales necesarios para el sistema en que va a ser implementado. En tercer lugar, facilita el análisis del tiempo de respuesta de las aplicaciones que corren bajo ese determinado sistema. Por todos estos motivos, se ha desarrollado un herramienta que permite conocer el comportamiento temporal de un determinado sistema.

La herramienta diseñada y desarrollada descrita en esta memoria permite conocer los tiempos característicos de varios de los servicios que ofrece el sistema descrito en el perfil mínimo del estándar. El motivo por el que han sido seleccionados esos servicios en concreto es consecuencia de la experiencia del grupo de Computadores y Tiempo Real en el diseño de controladores para robots industriales. Sin embargo, esta herramienta no garantiza las medidas de tiempos en los peores casos, ya que caracterizar con precisión estos tiempos, en la práctica resulta muy complejo, y se hace necesario el uso de técnicas especiales. Los servicios analizados son mostrados a continuación:

- Mecanismos de sincronización: Para analizar este servicios se realizan dos experimentos. El primero de ellos caracteriza temporalmente las operaciones que pueden realizarse con los mutexes, teniendo en cuenta las posibles políticas de planificación, mientras que en el segundo se realiza un estudio sobre las variables condicionales.

- Gestión del tiempo: Debido a que un sistema de tiempo real, las restricciones impuestas se refieren siempre a un plazo de tiempo limitado, es muy importante realizar un estudio de los mecanismos de gestión de tiempo que este implementa. Por este motivo, nuestra herramienta realiza un experimento que permite determinar tanto la resolución del reloj del sistema, como la de los mecanismos de suspensión (*sleep*). Asimismo, son objeto de estudio los temporizadores, y los retrasos (*delays*) entre threads de diferentes prioridades.

- Señales de tiempo real: En este experimento se implementan diferentes operaciones que proporcionan las señales de tiempo real, con objeto de caracterizar temporalmente su funcionamiento.

- Gestión de memoria: La gestión de memoria en tiempo real se realiza mediante dos funciones: *malloc()* y *free()*, de modo que se lleva a cabo un experimento que permita determinar el comportamiento de dichas funciones, con diferentes tamaños de bloques de memoria.

- Interrupciones: Las interrupciones causan una sobrecarga que puede llegar a ser crítica en la estabilidad del sistema. Se realiza un experimento que tiene como fin determinar la frecuencia de interrupciones en el sistema, así como la duración de cada una de ellas.

A partir de esta herramienta se ha llevado a cabo la caracterización temporal sobre dos sistemas operativos con aplicaciones diferentes. Por un lado, se han ejecutado estos experimentos sobre MaRTE. Este es un sistema operativo de tiempo real mínimo para aplicaciones empotradas, diseñado según las especificaciones recogidas en el perfil mínimo del estándar. Por otro lado, se ha llevado a cabo la realización de las medidas sobre un sistema de tiempo compartido Linux, sin funcionalidades específicas de tiempo real.

A pesar de que las prestaciones equipo sobre el que se han llevado a cabo las pruebas del sistema Linux son bastante superiores a las del sistema MaRTE hemos podido comprobar que las diferencias son bastante notables. Para empezar, se ha podido comprobar que MaRTE ofrece unos tiempos de respuesta predecibles, en el sentido de que hay poca diferencia entre los casos peor, mejor y promedio, mientras que en las medidas realizadas sobre Linux hemos podido observar que en ocasiones se producen saltos en la ejecución, como consecuencia de ser un sistema de tiempo compartido. Además, y pese a que los tiempos mínimos registrados son mejores en Linux, los tiempos máximos y promedios, que resultan más críticos, habitualmente son bastante peores en Linux. Por otro lado, y refiriéndonos brevemente a cada uno de los servicios:

- Mutexes: En la ejecución sobre MaRTE, hemos podido observar que los tiempos para el protocolo por protección de prioridad son considerablemente inferiores a los medidos con el protocolo con herencia de prioridad. Linux, no tiene estos protocolos de prioridad.

- Variables Condicionales: En ambos sistemas se concluye que el tiempo requerido para la señalización de una variable condicional de forma individual y conjunta es semejante.
- Gestión del tiempo: En este experimento se aprecia una gran diferencia entre los resultados temporales proporcionados por los dos sistemas operativos, siendo muy superiores los reportados por Linux, que no ha sido diseñado para cumplir plazos de tiempo críticos.
- Gestión de memoria: En MaRTE hemos podido comprobar que el tiempo necesario para reservar un bloque de memoria es independiente del tamaño de este. Sin embargo, esto no ocurre en Linux, donde claramente se disparan los tiempos cuando el tamaño de los bloques es elevado. En ambos sistemas, se obtienen valores promedios inferiores para la función de liberación de bloques.
- Sobrecarga producida por interrupciones: en MaRTE, no se han detectado prácticamente interrupciones. En Linux , debido a que se trata de un sistema de tiempo compartido, con más de un usuario, se producen continuamente.

La herramienta desarrollada es portable a cualquier sistema que cumpla el estándar POSIX de sistema de tiempo real mínimo, y puede ser utilizada por ello para caracterizar temporalmente los servicios de cualquier sistema operativo de este tipo. Ello permite tener una buena estimación de las prestaciones esperables, así como comparar entre diferentes sistemas operativos.

En el futuro sería deseable completar la herramienta añadiendo el resto de los servicios POSIX que por ser menos importantes y por motivos de tiempo de desarrollo, no se han incluido en el presente trabajo. También sería aconsejable implementar un conjunto similar de experimentos para caracterizar las prestaciones de los servicios del sistema operativo cuando se utilizan desde aplicaciones en lenguaje Ada, que también está especificado en el estándar POSIX.

6. Bibliografía

6. Bibliografía

[ACES] Ada Compilers Test Suites
http://www.adaic.org/compilers/articles/benchmrk.html

[ALD02] *"Planificación de Tareas en Sistemas de Tiempo Real Estricto para Aplicaciones Empotradas" (Tesis Doctoral)*
Mario Aldea Rivas
Universidad de Cantabria, Noviembre 2002.

[ALNET] *"An introduction to real time concepts"*
Staffan Nilsson
http://www.algonet.se/~staffann/developer/realtimeintro.htm

[ALDGOa] *"Evalutation of New POSIX Real Time Operating Systems for Small Embedded Platforms" (Artículo)*
Mario Aldea Rivas, Michael González Harbour
Proceedings of the 15th Euromicro Conference on Real-Time Systems, ECRTS, Porto, Portugal, Julio 2003

[ALDGOb] *"POSIX-Compatible Application Defined-Scheduling in MaRTE OS" (Artículo)*
Mario Aldea Rivas, Michael González Harbour
Proceedings of 14th Euromicro Conference on Real-Time Systems, Vienna, Austria, IEEE Computer Society Press, Junio 2002

[GCALB] *"Migración de un sistema operative de tiempo real, MaRTE OS, a un microcontrolador" (Proyecto Fin de Carrera)*
Alberto Gutiérrez Castro
Universidad de Cantabria, Octubre 2003

[GODOU] *"Tostadores y POSIX" (Artículo)*
Michael González Harbour, C. Douglas Locke
Universidad de Cantabria, Lockheed Martín Corporation, 1997

[GON01a] *"POSIX de Tiempo Real" (Apuntes curso doctorado)*
Michael González Harbour
Universidad de Cantabria, Octubre 2001.

[GON01b] *"POSIX de Tiempo Real" (Artículo)*
Michael González Harbour
Universidad de Cantabria, Marzo 2001.

[MaRTE] *"MaRTE OS: Minimal Real-Time Operating System for embedded Applications" (página Web)*
Mario Aldea Rivas y Michael González Harbour
http://marte.unican.es

[MRCVa] *"Dynamic storage allocation for real-time embedded systems"*
M. Masmano, I.Ripoll, y A. Crespo
Universidad Politécnica de Valencia, Valencia.

[OBENL] *"Posix in Real Time"*
Kevin M. Obenland
http://xtrj.org/collection/posix_rtos.htm

[PAL03] *"Sistemas operativos de tiempo real (RTOS)" (Artículo)*
Héctor Palacios Pérez
Sistemas Embebidos, S.A, Marzo 2003.

[PIGW] *Performance Issues Working Group*
SIGAda, Special Interest Group in Ada
http://sigada.org/wg/piwg/piwg.html

[WCET] *WCET Analysis of Probabilistic Hard Real-Time Systems*
G. Bernat, A. Colin, S. M. Petters (2002)
Proceedings of the 23rd Real-Time Systems Symposium RTSS 2002

[1003.13] *"IEEE Standard for Information Technology Standardized Application Environment Profile - POSIX Realtime Application Support (AEP)"*
IEEE Computer Society, Septiembre 2004