Centro de Investigación y de Estudios
Avanzados
del Instituto Politécnico Nacional

Departamento de Ingeniería Eléctrica

Sección Computación

Kernel de tiempo real basado en Linux para una PDA.

Tesis que presenta

Francisco Javier Zuluaga Ramírez

para obtener el Grado de

Maestro en Ciencias

en la Especialidad de

Ingeniería Eléctrica

Director de la Tesis

Dr. Predro Mejía Álvarez

México, D.F. **2005**

Resumen

Durante los últimos cinco años la cantidad de dispositivos móviles existentes en el mercado ha crecido bastante, lo que ha generado la necesidad de desarrollar un mayor número aplicaciones para este tipo de dispositivos. Algunas de estas aplicaciones requieren cumplir con ciertas restricciones de tiempo. Como por ejemplo, la comunicación de datos entre dispositivos o el procesamiento de contenido multimedia, lo cual implica que el sistema operativo con el que se cuente en el dispositivo móvil ofrezca soporte a procesos de tiempo real.

Linux es un sistema operativo de código abierto muy popular, lo que permite un estudio minucioso de su arquitectura y no nos impone restricción alguna para la modificación de su kernel.

Esta tesis presenta el diseño e implementación de un kernel basado en Linux para una PDA (*Personal Digital Assistant*) que ofrece soporte para procesos de tiempo real, permitiendo el manejo de tareas periódicas y aperiódicas. Nuestro kernel incluye políticas de planificación especiales, tales como, *Rate Monotonic* (RM) y *Earliest Deadline First* (EDF), así como diferentes mecanismos de comunicación y sincronización entre procesos, como semáforos y paso de mensajes mediante buzones.

Abstract

During the last years the amount of mobile devices available at the market-place has increased. It has generated the necessity of developing more applications for this kind of devices. Some of these applications have some timing restrictions, for example, the data transference between devices or the processing of multimedia content; therefore the operating system implemented in the mobile device must offer support for real time processes.

Linux is a very popular open source operating system, that allows to study in detail its architecture, and does not impose any restriction on making modifications to the kernel.

This thesis presents the design and implementation of a kernel based on Linux for a PDA (Personal Digital Assistant), which has a support for real time processes, allowing the existence of periodic and non periodic tasks. The kernel includes scheduling policies of specific purpose, such as *Rate Monotonic* (RM) and *Earliest Deadline* (EDF), as well as mechanisms for communication and synchronization between processes, like semaphores and messages passing across mailboxes.

Agradecimientos

Le agradezco a Dios por la vida, y por todo lo que de ella tengo.

Quiero dar las gracias a todas las personas e instituciones que de una u otra forma estuvieron involucradas en este trabajo.

El doctor Pedro Mejía Álvarez con sus consejos, regaños y paciencia durante el tiempo que he trabajado con él, hizó posible la culminación de esta tesis.

De manera especial agradezco a los doctores Adriano de Luca y Jorge Buenabad por los comentarios hechos para la mejora de mi trabajo.

A los doctores Sergio Chapa, Francisco Rodríguez, Guillermo Morales, Arturo Díaz, Gerardo de la Fraga, profesores de la Sección de Computación, y al compañero Amilcar Viveros por haber compartido sus conocimientos conmigo y ser parte de mi formación académica.

Siento un gran placer poder agracer a mis compañeros de la Sección de Computación por su amistad y el tiempo compartido, especialmente a Abigail, Claudia, Grettel, Luis, Mario I. Florentino, Mónica, Rocio, Rosario y Ulises.

A Juan Manuel y Omar por las charlas y consejos, que tan útiles me fueron al desarrollar mi tesis.

Edith, Gary e Israel, además de ser tres excelentes personas y amigos mios, su punto de vista fue muy valioso al momento de redactar este documento.

Quiero agradecer y hacer un reconocimiento a las personas cuyo trabajo me sirvió como base o como guia. A Douglas Comer por Xinu, a Jorge Buenabad por Xinix, a Paolo Mantegazza y su equipo de trabajo por RTAI y a Linus Torvalds por Linux.

Debo destacar que el apoyo económico del CONACyT y del CINVESTAV fueron muy importantes e indispensables para que yo pudiera realizar mis estudios de maestria.

A los seres más importantes en mi vida,
mi mamá Ma. del Carmen,
mi papá Francisco,
y mis hermanos Adrián, Nadia y Liseth.

Índice general

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

En los últimos años el desarrollo de la tecnología ha permitido la miniaturización de los dispositivos electrónicos, lo que ha propiciado la proliferación de los mismos. Actualmente es común encontrar sistemas electrónicos en una gran cantidad de objetos de uso cotidiano, como teléfonos celulares, PDA's, video juegos, sistemas de control (como los de automóviles) e inclusive en aparatos electrodomésticos. Este tipo de sistemas, conocidos como sistemas empotrados, actualmente ha superado en número a los sistemas de cómputo tradicionales.

El creciente número de sistemas empotrados en el mercado ha generado la necesidad de desarrollar aplicaciones específicas para estos sistemas. En algunas de estas aplicaciones, tales como la transmisión de multimedia y robótica, no sólo se debe de asegurar el funcionamiento correcto, también es necesario cumplir ciertas restricciones de tiempo, pues de lo contrario no es posible asegurar su utilidad. A los sistemas empotrados en los cuales se ejecutan aplicaciones con restricciones de tiempo, los nombraremos sistemas empotrados de tiempo real.

En un sistema empotrado de tiempo real uno de los componentes más importantes es el sistema operativo, ya que es el encargado de controlar todos los recursos de hardware de la plataforma en la que se encuentra ejecutándose, y de administrar estos recursos de forma que se pueda asegurar un cumplimiento de los requerimientos temporales de los procesos que se ejecutan en el sistema. Debido a que se requiere una ejecución predecible de las aplicaciones en el sistema, el sistema operativo no debería de manejar discos, sistemas DMA (*Direct Memory Access*) o permitir a los procesos reservar memoria dinámicamente.

Este trabajo presenta el diseño y la implementación de un kernel de tiempo real basado en Linux para una PDA, el IPaq H3950. El trabajo consiste en una extensión al sistema RTAI (*Realtime Application Interface*) [1], el cual no cuenta con soporte para el dispositivo planteado.

1.1. Motivación

Las PDA son dispositivos que en los últimos años se han vuelto muy populares, debido a la reducción en sus costos y al incremento en la versatilidad de sus aplicaciones. La principal desventaja de este tipo de dispositivos comparados con las computadoras personales, son su limitado poder cómputo y poca capacidad de almacenamiento.

Algunas aplicaciones existentes para las PDA's, por ejemplo, la transmisión de datos en una comunicación inalámbrica entre dispositivos, el procesamiento de audio y video o inclusive algunos juegos, requieren cumplir con ciertas restricciones de tiempo para su operación adecuada. Esto hace necesario el desarrollo de un sistema operativo que ofrezca soporte a estas aplicaciones, pero en el cual se tome en consideración todas las restricciones arquitecturales de este tipo de dispositivos.

Linux es un sistema operativo que ha ganado gran aceptación tanto en ámbitos científicos como comerciales. Esto es debido no solamente a que se trata de un sistema de código abierto, sino también a que actualmente cuenta con un diseño que permite una fácil migración del sistema entre distintas arquitecturas. Dentro de su diseño, Linux cuenta con una capa mediante la cual, al realizar cambios en el kernel para migrarlo entre arquitecturas, sólo es necesario modificar una pequeña porción del código dependiente de la plataforma.

En los últimos años la cantidad de aplicaciones desarrolladas para Linux se ha incrementado en gran medida, a pesar de esto, el tamaño de su kernel se ha mantenido prácticamente constante, con un tamaño de alrededor de 0.5 Mbytes, a diferencia de otros sistemas operativos como Windows. Este hecho aunado a las características previamente mencionadas, hace sumamente atractiva la idea de implementar un kernel basado en Linux para la PDA.

Actualmente existen diferentes implementaciones de sistemas operativos que ofrecen soporte para procesos de tiempo real, sin embargo, la mayoría de las implementaciones son sistemas propietarios. Por lo que, tanto usuarios como desarrolladores se encuentran limitados por esta característica. En el caso

de sistemas de código abierto, el más representativo es RTAI. A pesar de que RTAI cuenta con gran soporte, actualmente no existe una implementación de la herramienta para una arquitectura como la planteada en el presente trabajo, y considerando el auge que los dispositivos móviles han tenido, es importante considerar el desarrollo de un sistema que ofrezca soporte para procesos de tiempo real para este tipo de dispositivos.

Se tomó la decisión de iniciar el desarrollo del kernel desde la etapa más básica, de modo que se tiene un completo conocimiento y control de la arquitectura de éste, esto permite que sólo sean incorporados los componentes requeridos por las aplicaciones de tiempo real, para así mantener a nuestro kernel lo más pequeño posible y con la capacidad de ser migrado a diferentes arquitecturas de manera simple. De esta forma, el kernel servirá como base para futuros estudios sobre sistemas de tiempo real.

1.2. Trabajos relacionados

Actualmente existe una gran cantidad de sistemas operativos de tiempo real, tanto propietarios como de desarrollo. En cada uno de ellos es posible observar ventajas y desventajas, así como los diferentes paradigmas de diseño adoptados. De los sistemas operativos comerciales revisaremos las características de AMX de KADAK y de RTLinux de FSMLabs [2]. En ambos casos se trata de sistemas de gran tamaño, que tratan de cubrir una amplia gama de aplicaciones para sistemas embebidos.

Por otro lado, tenemos a los sistemas operativos experimentales, que son de tamaño medio y mantienen diseños flexibles. En algunos sistemas, se implantan algunas técnicas de planificación propias, por ejemplo EMERALDS (*Extensible Microkernel for Embedded , ReAL Time, Distributed Systems*), en el que se implanta un algoritmo llamado CSD (*Combined Static/Dynamic Scheduler*), que combina los algoritmos de planificación RM y EDF. En el presente trabajo describimos a algunas características de EMERALDS y RTAI.

1.2.1. Sistemas operativos de tiempo real comerciales

RTLinux

RTLinux es un sistema originalmente desarrollado por Michael Barabanov y Victor Yodaiken, en *The New Mexico Institute of Mining and Technology*, en Socorro, New Mexico. Actualmente es distribuido por los laboratorios

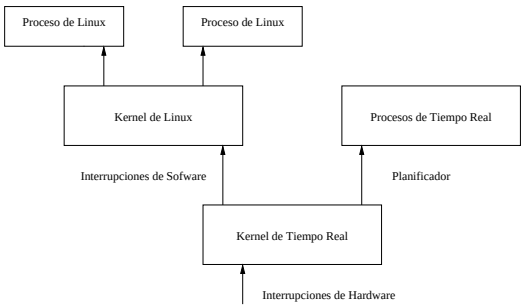

Figura 1.1: Diagrama a bloques de la arquitectura de RTLinux.

FSMLabs. Ofrece soporte a procesos de tiempo real duro y se encuentra disponible para distintas arquitecturas como ARM e Intel XScale, siendo la última la arquitectura para la cual se desarrolló nuestro sistema.

Su filosofía se basa en el kernel dual, es decir, existe más de un kernel ejecutándose al mismo tiempo en la misma computadora, en este caso el kernel de Linux y el kernel de tiempo real. La idea básica empleada en el desarrollo de RTLinux, es mantener al kernel de Linux ejecutándose bajo el control del kernel de tiempo real, como se muestra en la figura 1.1. Cuando en el sistema no existen procesos de tiempo real en espera de ser ejecutados, Linux hace uso de los recursos del sistema. En el momento en el que arriban procesos de tiempo real al sistema, el kernel de Linux es desalojado y el control del procesador es cedido al planificador de tiempo real.

Como se observa en la figura 1.1, RTLinux introduce una capa de emulación de interrupciones mediante software entre el kernel de Linux y el hardware controlador de interrupciones, con la cual RTLinux toma el control de ellas. En la capa de emulación se incluyen funciones que simulan la habilitación y deshabilitación de interrupciones.

El código fuente del kernel de Linux es modificado para reemplazar todas las apariciones de las instrucciones *sti, cli e iret*, por las funciones definidas en RTLinux, de forma que Linux no tiene acceso directo a este hardware y sólo hace uso de las instrucciones integradas en RTLinux [3].

AMX

Sistema operativo de tiempo real distribuido por KADAK. Se encuentra disponible para diversos procesadores incluyendo a ARM y MIPS. Ofrece

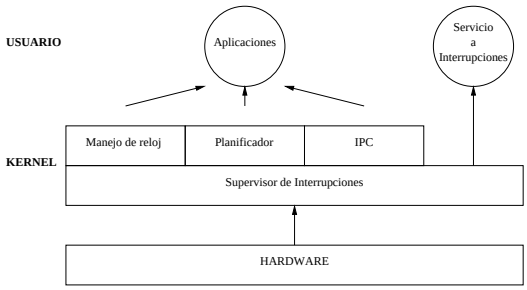

Figura 1.2: Diagrama a bloques de la arquitectura de AMX.

servicios para la creación, manipulación, eliminación y retrazo de procesos, así como para su comunicación y sincronización, y cuenta con características interesantes, como la protección en contra de inversión de prioridades.

En el diagrama de la figura 1.2 es posible observar los diferentes componentes de sistema y los servicios ofrecidos a las tareas del usuario. El kernel de AMX se encarga de la interacción con el hardware, el supervisor de interrupciones es el encargado de verificar la ocurrencia de una interrupción y de pasar el control del procesador a la rutina de servicio adecuada. En caso de que la interrupción ocurrida corresponda a la del reloj, el manejador de reloj es invocado. Este manejador se encarga de reanudar la tarea llamada *Kernel Task* (tarea del kernel) a intervalos periódicos. La tarea del kernel se encarga de verificar la existencia de tareas que hallan solicitado ser suspendidas por un intervalo de tiempo dado, y si el intervalo a expirado, marcar las tareas como listas para reanudar su ejecución. Las tareas en ejecución pueden hacer uso de distintos servicios ofrecidos por el kernel, que incluyen semáforos, paso de mensajes y rutinas para manejo de listas ligadas.

1.2.2. Sistemas operativos de tiempo real de desarrollo

EMERALDS

EMERALDS es un sistema de tiempo real diseñado para sistemas embebidos distribuidos de capacidad media o pequeña (10 procesadores o menos conectados a través de una red). Ofrece los servicios estándares de un sistema operativo como: manejo de procesos, planificadores de tiempo real, protección de espacios de direcciones, paso de mensajes, semáforos, y se trata de mantener el tamaño del kernel en sólo unas decenas de kilobytes [4].

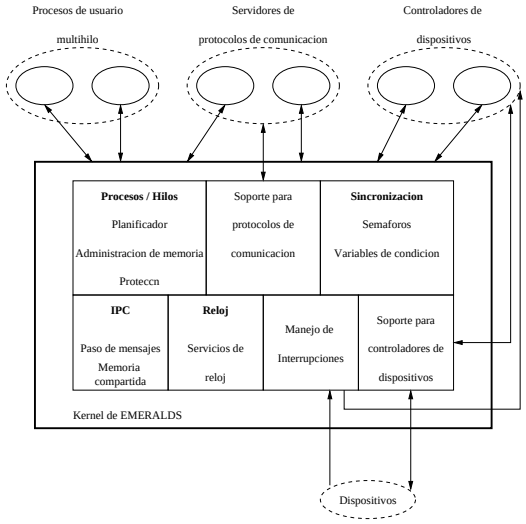

Figura 1.3: Diagrama a bloques de la arquitectura de EMERALDS.

En la figura 1.3 se muestra la arquitectura de EMERALDS. En su diseño se destaca la facilidad para agregar nuevos protocolos de comunicaciones, así como para la instalación de nuevos controladores de dispositivos, puesto que no forman parte del kernel. Cuenta con soporte a procesos multihilo y provee una completa protección de memoria entre los mismos. Permite comunicación entre procesos a través de paso de mensajes y memoria compartida, en tanto que para lograr la sincronización entre ellos se emplean semáforos y variables de condición [4].

RTAI

RTAI es un sistema desarrollado en el *Politecnico di Milano* en un proyecto dedicado a la investigación de sistemas de control de tiempo real basados en PC, que se encuentra a cargo de Paolo Mantegazza. El desarrollo de este sistema comenzó en la década de los 80. El sistema inicial se ejecutaba en ambiente de *DOS (Disk Operating System)*, en el cual el concepto de *Capa de Abstracción de Hardware* fue extendido para incluir los servicios requeridos por las aplicaciones de tiempo real. La necesidad de explotar de manera adecuada el modo de 32 bits de la PC, hicieron pensar en la migración del sistema. Se consideraron diversas opciones, como GNU-DOS y Linux 2.0.xx.

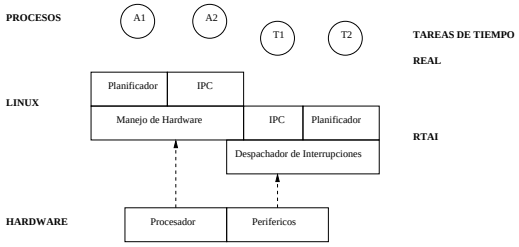

Figura 1.4: Diagrama a bloques de la arquitectura de RTAI.

La gran cantidad de cambios requeridos en el kernel de Linux 2.0.xx permitió ver que aún era complicada una migración a este sistema, sin embargo, al ser liberada la versión 2.2.xx los cambios requeridos en el kernel para la implementación de la capa de abstracción de hardware fueron mínimos, lo que permitió un desarrollo más extenso de RTAI.

Un módulo de gran importancia incluido en RTAI, es el de LXRT. El módulo de LXRT permite un desarrollo más rápido de aplicaciones de tiempo real, ya que permite realizar la verificación de su operación en el espacio de usuario antes de ser incluidas como tareas de tiempo real en el kernel.

Su arquitectura es similar a RTLinux, en la que se trata a Linux como una tarea de tiempo real de baja prioridad, la cual realiza sus operaciones de manera normal siempre que no existan tareas de tiempo real de mayor prioridad. En la arquitectura de RTAI se pueden distinguir dos tipos de interrupciones, aquellas generadas por el procesador y aquellas generadas por los periféricos, las primeras son procesadas de manera normal por el kernel de Linux, en tanto que las segundas son procesadas ahora por el despachador de interrupciones de RTAI. El despachador de RTAI envía las interrupciones a Linux cuando no existe actividad de tiempo real, de modo que, él puede realizar sus tareas de manera normal. Las macros de *sti* y *cli*, son sustituidas por funciones que envían las señales a RTAI, cuando las interrupciones son deshabilitadas en el kernel de Linux, RTAI encola las interrupciones generadas para después enviarlas al momento en que Linux vuelva a habilitar las interrupciones.

En el diagrama de la figura 3.1 también se puede observar que en RTAI es necesario implementar un planificador y un conjunto de mecanismos de comunicación entre procesos independientes de los Linux [1].

1.3. Implantación del kernel de tiempo real en la PDA

En la sección 1.2 se presentaron los sistemas AMX,EMERALDS, RTLinux y RTAI, a través de lo cual se dan a conocer diferentes alternativas para la implantación del kernel de tiempo real.
Los sistemas operativos EMERALDS y AMX han sido desarrollados de manera independiente a cualquier otro. En ambos sistemas operativos, a pesar de que los diseñadores conocen completamente su arquitectura, se requiere proveer al usuario con las herramientas de desarrollo y depuración necesarias para la creación de nuevas aplicaciones. Además para agregar soporte a algún dispositivo o protocolo, se debe diseñar completamente el controlador correspondiente.
En RTLinux y RTAI se hace uso del kernel dual, en el que dos sistemas operativos se ejecutan en una misma computadora al mismo tiempo (en este caso Linux y el kernel de tiempo real). La principal ventaja de tener a los sistemas operativos compartiendo el mismo espacio de direcciones, es que los servicios existentes en cada uno de ellos están disponibles para el otro. En tanto que, el principal riesgo es que una falla en alguno de ellos causará que el otro también falle.
En esta tesis se presenta el desarrollo de un kernel de tiempo real para la iPaq H3950, que consiste en una extensón a RTAI. Se decidió seguir la línea de trabajo de RTAI debido a que Linux cuenta con una amplia gama de herramientas para desarrollar y depurar aplicaciones, además de que la lista de dispositivos de hardware y protocolos soportados es muy extensa.
En la sección 1.2.2 se presentó la arquitectura de RTAI, en la cual un aspecto clave para permitir que Linux se ejecute a la par del kernel de tiempo real, es el esquema de manejo de interrupciones introducido por RTAI, que es como sigue:

Se cuenta con una capa de abstracción de hardware (RTHAL), como le llaman los autores, que es la encargada de mantener todo el control de las interrupciones. Cuando existen procesos de tiempo real en el sistema, las señales de interrupción son enviadas a ellos, pero si no existen, las señales son enviadas a Linux.

Al agregar el soporte de RTAI, el manejo de las interrupciones en Linux

debe ser modificado ligeramente. La aparición de las macros *cli* y *sti* se sustituye por funciones definidas en RTAI que interaccionan con la RTHAL. En RTAI se cuenta con una variable que permite realizar la habilitación y deshabilitación de las interrupciones mediante software. Cuando las interrupciones son deshabilitadas, la variable es reiniciada, en lugar de realmente deshabilitar las interrupciones. Al generarse una interrupción, se checa la variable. Si la variable se encuentra habilitada, la rutina de servicio a la interrupción (*ISR*) es invocada, en caso contrario se cuenta con otra variable en la cual se lleva el conteo de las interrupciones pendientes. La ISR invocada depende del kernel que tenga el control del procesador al momento de generarse la interrupción. [1].

El módulo de RTAI consta de dos submódulos:

- La capa de abstracción de hardware.
- Los servicios de RTAI.

La capa de abstracción de hardware se encarga de:

- Encapsula apuntadores a funciones del kernel de Linux en una estructura de datos, para permitir un fácil manejo de todas las funcionalidades que son importantes para las aplicaciones de tiempo real.
- Incluye funciones que llevan a cabo la interacción con el hardware (habilitar, deshabilitar y procesar interrupciones) de la misma forma en que lo hace Linux.
- En el kernel de Linux, sustituye las llamadas a función originales con llamadas a los apuntadores a función de la estructura rthal.

El módulo de servicios de RTAI hace uso de la capa de abstracción de hardware y contiene funciones para:

- El control de interrupciones vía software.
- El procesamiento de las interrupciones.

Inicialmente los apuntadores de la capa de abstracción de hardware, hace referencia a las funciones que procesan las interrupciones de manera identica a como lo hace Linux. Mediante el módulo de servicios de RTAI se agrega el soporte a procesos de tiempo real, al cargar este módulo en el kernel de Linux, los apuntadores de la RTHAL harán referencia a las funciones contenidas en él, que se encargan de la emulación mediente software y del procesamiento de las interrupciones.

La RTHAL se encarga de la interacción con el hardware, por lo que es dependiente de la arquitectura. Actualmente no existe disponible una versión de la capa de abstracción de hardware que pueda ser integrada al kernel de Linux que se ejecuta en la iPaq H3950, debido a esto, en la primera parte de este trabajo fue necesario desarrollar una versión de la RTHAL, para así poder hacer uso de los servicios de RTAI en la PDA.

El módulo de RTAI, incluye un kernel de tiempo real que cuenta, entre otros, con servicios para:

- Manejo de procesos.
- Comunicación entre procesos.
- Planificación de procesos mediante FIFO-Round-Robin.

Sin embargo, nosotros deseamos contar con pleno conocimiento del kernel e incluir sólo los servicios de nuestro interés, así que, la segunda parte del trabajo de tesis consistió en la implementación de un kernel de tiempo real, que hace uso de los servicios de RTAI para ejecutarse concurrentemente con Linux, el cual cuenta con servicios para:

- Manejo de procesos.
- Comunicación y sincronización entre procesos (semáforos y buzones).
- Planificación de procesos mediante FIFO-Round-Robin y mediante RM.

Es importante destacar que algunos servicios como: el manejo de colas y semáforos, fueron implementados haciendo uso de la aproximación empleada en los sistemas XINU y XINIX.

1.4. Organización de la tesis

En este capítulo se ha descrito de manera general nuestro trabajo de tesis, que consta de la implantación de la capa de abstracción de hardware de RTAI y de un kernel de tiempo real para la iPaq H3950.

El resto de la tesis ha sido organizado de la siguiente forma:

En el segundo capítulo se explican conceptos básicos de tiempo real y sistemas operativos. Se definen los sistemas de tiempo real y sus características, posteriormente se describe el concepto de proceso, y los parámetros requeridos para la representación de un proceso de tiempo real dentro de un sistema operativo. Presenta a los sistemas multitarea, y la necesidad de los algoritmos de planificación. Finalmente se incluye una clasificación de los algoritmos de planificación para sistemas de tiempo real, y se presenta el funcionamiento de algunos de estos algoritmos.

El tercer capítulo está dedicado a mostrar la arquitectura de Linux. Destaca las restricciones por las que no es posible utilizarlo como un sistema operativo de tiempo real, y plantea las alternativas existentes para convertirlo en un sistema de tiempo real. Presenta el concepto de interrupciones, la necesidad de ellas y su funcionamiento. Muestra los mecanismos empleados en Linux para el manejo de interrupciones, así como las estructuras de datos involucradas en estos mecanismos. La segunda parte del capítulo describe la capa de abstracción de hardware y los cambios requeridos en el kernel de Linux para su implantación en la PDA.

El cuarto capítulo presenta la arquitectura del kernel de tiempo real. En primera instancia de manera general, y posteriormente de manera especifica. Se muestran los diferentes manejadores incluidos en el kernel, así como las primitivas ofrecidas por cada uno de ellos a los procesos de usuario.

En el quinto capítulo se presentan los resultados obtenidos en las pruebas realizadas a nuestro kernel de tiempo real, se muestra el entorno requerido para la realización de estas pruebas y los pasos que un usuario de nuestro sistema debe seguir para hacer uso de él.

En el último capítulo se plantean algunos de los problemas encontrados a lo largo del trabajo de tesis, las alternativas para solucionarlos, sus ventajas

y desventajas, y el camino que se tomó en la tesis. También se proponen extensiones y mejoras a nuestro trabajo.

Capítulo 2

Tiempo real

2.1. Definición de sistema de tiempo real

Un sistema de tiempo real (STR) es un sistema informático que, interacciona repetidamente con su entorno y responde a los estímulos que recibe del mismo dentro de un plazo de tiempo determinado [5]. Para que el funcionamiento del sistema sea correcto, no basta con que las acciones del sistema sean correctas, si no que además, tienen que ejecutarse dentro del intervalo de tiempo especificado. Los resultados deben producirse en los momentos en que aún tengan validez dentro del ambiente a controlar.

Los sistemas de tiempo real controlan un ambiente que tiene restricciones de tiempo bien definidas (fig. 2.1). Es por ello que se vuelven más complejos y por lo tanto demandan una alta confiabilidad, con resultados correctos, predecibles y a tiempo.

Actualmente, los sistemas de tiempo real abarcan un amplio espectro de aplicaciones, desde los más simples microcontroladores (tales como un microcontrolador para el control de un automóvil) hasta sistemas grandes y distribuidos (como los sistemas de control de tráfico aéreo). Se espera que en un futuro los sistemas de tiempo real sean aun más complejos y se utilicen en el control de estaciones espaciales, en sistemas integrados de visión, robótica, inteligencia artificial y en entornos peligrosos como plantas químicas, nucleares, entre otros.

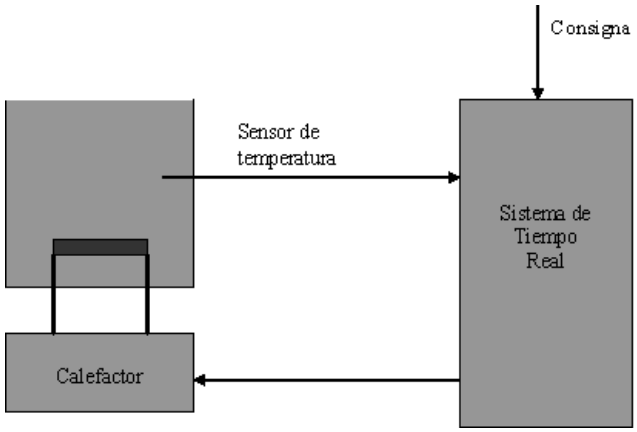

Figura 2.1: Sistema de Tiempo Real.

2.2. Elementos de un sistema de tiempo real

Un sistema de tiempo real consiste principalmente de computadoras, y elementos externos con los cuales el software de la computadora debe interactuar simultáneamente. En la figura 2.2 se muestran los elementos generales de un sistema de tiempo real donde el objetivo principal es controlar un ambiente. En dicha figura se distinguen los siguientes elementos:

- **Ambiente:** El término ambiente de la figura 2.2 se refiere al sistema controlado. Por ejemplo, un motor, un sistema de manufactura, un robot, o un avión, etc. El estado del ambiente (entorno físico) es supervisado por los sensores y puede cambiar por los actuadores.
- **Convertidores:** Los convertidores análogico-digital convierten las señales generadas por el ambiente (analógicas) a una serie de datos que la computadora interpreta (digitales).
- **Reloj de Tiempo Real:** El reloj de tiempo real permite al sistema contar el tiempo en que se ejecutan acciones. De la misma forma, el reloj de tiempo real permite al sistema configurar los tiempos para la planificación de las tareas. Mediante el reloj de tiempo real es posible configurar interrupciones para controlar dispositivos externos, recibir y sincronizar señales de comunicación, y monitorear el cumplimiento de los requerimientos temporales de las tareas del sistema. Sin el reloj de tiempo real no sería posible configurar los tiempos de ejecución de

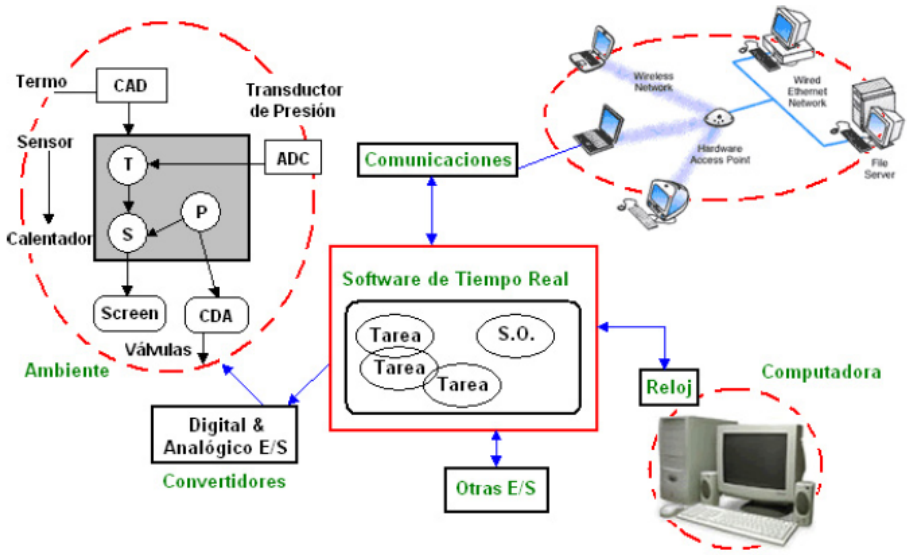

Figura 2.2: Elementos de un Sistema de Tiempo Real.

las tareas de tiempo real, y por tanto la planificación del sistema, ni tampoco sería posible saber si las tareas cumplen con sus restricciones temporales.

- **Software de Tiempo Real:** El software de tiempo real incluye un sistema operativo (o kernel) y de tareas las cuales son planificadas por el kernel. La estructura del kernel está compuesta de manejadores de tiempo, tareas, memoria, dispositivos, y de todos los recursos del sistema de cómputo. Las tareas del sistema (o procesos) son las entidades de software que permiten controlar el medio ambiente. Cada tarea es un procedimiento de software que se ejecuta de forma continua, sin embargo, la ejecución concurrente de las tareas es controlada por el manejador de procesos del kernel.

- **Comunicaciones:** En el sistema de tiempo real puede haber varias computadoras que interactúan entre sí. Entre ellas existe un medio físico de comunicación y un protocolo de comunicaciones que les permite enlazarse, compartir información y sincronizar su ejecución. Mediante la comunicación es posible compartir recursos de hardware (por ejemplo, dispositivos de entrada y salida), y mejorar la eficiencia de aplica-

ciones mediante la distribución del cómputo, y lograr mayor rapidez de ejecución.

- **Otras E/S (entradas y salidas):** Un sistema de tiempo real tiene como entrada principalmente el comportamiento del sistema físico controlado. Sin embargo, existen entradas y salidas con las que interactúan con el usuario, como son el teclado, el ratón y cámaras.

2.3. Características de los sistemas de tiempo real

Las características más destacables de un sistema en tiempo real son:

- **Interacción con el Entorno:** Los sistemas de tiempo real interactúan con un entorno externo, por lo que es necesario utilizar sensores que permitan realizar la toma de datos de él y un conjunto de actuadores que permitan modificar el estado del sistema controlado.

- **Restricciones de Tiempo:** Los sistemas de tiempo real tienen que procesar información en un plazo de tiempo finito. La obtención de resultados fuera de plazo puede ocasionar graves consecuencias aún cuando los resultados sean correctos. Esto les diferencia de otros sistemas donde se pueden imponer restricciones de tiempo para comodidad del usuario en donde su incumplimiento no es crítico.

- **Predecible o Determinista:** No es fácil diseñar e implementar un sistema que garantice que la salida apropiada se generará en el tiempo adecuado bajo cualquier circunstancia, sin embargo, los sistemas de tiempo real lo deben asegurar.

- **Fiabilidad y Seguridad:** Por la naturaleza de los sistemas de tiempo real, estos requieren que la computadora interactúe con el mundo externo (monitorizando sensores y activando actuadores), por lo que el hardware y el software de las computadoras en un sistema de tiempo real deben ser fiables y seguros.

2.4. Clasificación de los sistemas de tiempo real

Los sistemas de tiempo real se pueden clasificar de muchas maneras (por su arquitectura, por su especificación para la creación del software o por los flujos de ejecución de los que está compuesto). La clasificación más conocida es aquella en la que distinguen a los sistemas de tiempo real con base a sus requisitos temporales y de fiabilidad. Esta clasificación separa los sistemas de tiempo real en sistemas críticos y en sistemas acríticos.

2.4.1. Sistemas de tiempo real críticos (*hard real time systems*)

Son sistemas en los que, por su propia naturaleza, se puede llegar a tener un fallo muy grave en el caso de que no se cumpla con alguno de sus requisitos temporales. En su diseño se debe prestar especial atención a la disponibilidad de los recursos y asegurar que el sistema alcanza sus plazos temporales incluso en caso de un problema en sus componentes físicos[1]. Las características de los sistemas de tiempo real críticos son las siguientes:

- Tienen un plazo de respuesta estricto.
- Su comportamiento temporal es determinado por el entorno.
- El comportamiento en sobrecargas es predecible.
- Tiene requisitos de seguridad críticos.
- Provee tolerancia a fallos (mediante redundancia activa).
- El volumen de datos es reducido.

2.4.2. Sistemas de tiempo real acríticos (*soft real time systems*)

Son sistemas en los que es importante el cumplimiento de los requisitos temporales, pero si alguno de ellos no pudiera cumplirse el sistema no produciría un fallo. Por ejemplo, en los sistemas multimedia de tiempo real las

[1]Para lograr asegurar que el sistema responda aún cuando llegue a tener problemas con sus componentes físicos, generalmente se utiliza la redundancia de componentes.

imágenes y el sonido se deben transmitir dentro de plazos determinados, y si por alguna razón hubiera problemas y no se pudieran cumplir estos plazos, lo más que podría suceder es que una trama de sonido o de video se retrasara, o en el peor de los casos, que esta trama se tuviera que descartar, pero a pesar de los problemas, el cambio en el resultado no sería muy importante para el usuario. Las características principales de los sistemas acríticos son:

- Su plazo de respuesta es flexible.
- Tienen un comportamiento temporal determinado por la computadora.
- El comportamiento en sobrecargas es degradado.
- Los requisitos de seguridad son acríticos.
- Permite la recuperación en caso de fallos.
- Manejan un volumen de datos grande.

2.5. ¿Qué es un sistema operativo de tiempo real?

Un sistema operativo es un programa que actúa como un intermediario entre el usuario de una computadora y el hardware de ésta. El propósito de un sistema operativo es proveer un ambiente en el cual, el usuario puede ejecutar un programa de manera cómoda y eficiente. Actualmente existen muchos tipos de sistemas operativos, entre ellos resaltan los sistemas operativos multiprocesadores, los multitareas, los distribuidos, los empotrados, los paralelos y los de tiempo real; todos ellos, han sido diseñados para ser eficientes en determinadas áreas o condiciones de trabajo.

Un sistema operativo de tiempo real al igual que todos los sistemas operativos, es un programa que controla el hardware de la computadora y sirve de intermediario entre la máquina y el usuario. Lo que hace que sea de tiempo real es la capacidad de responder a estímulos generados externamente dentro de un plazo determinado y finito. Todo sistema operativo de tiempo real debe ser determinista, es decir, debe garantizar que las aplicaciones que éste controle se ejecuten dentro de una restricción de tiempo específico.

2.6. Proceso

2.6.1. Definición de proceso

Un proceso de manera informal puede ser visto como un programa en ejecución, sin embargo, es más que código de programa. Un proceso es la unidad de trabajo en la mayoría de los sistemas operativos y se caracteriza por tener información como: el código del programa, el *contador de programa* que indica la siguiente instrucción a ejecutar, una *pila* que contiene datos temporales como las direcciones de regreso, las variables temporales de datos y los parámetros del proceso. Además, los procesos tienen una sección de datos en donde se almacenan todas las variables globales que éste ocupe [6].

Un programa por sí solo no es un proceso; un programa es una *entidad pasiva*, como puede ser el contenido de un archivo o programa almacenado en disco, mientras que un proceso es una *entidad activa* controlada por el sistema operativo con un contador de programa, que especifica la siguiente instrucción a ejecutar y el conjunto de recursos asociados.

2.6.2. Definición de *quantum*

Un *quantum* es la unidad de tiempo que tiene asignada cada proceso para que éste se ejecute en el procesador. Cuando un proceso consume su *quantum*, el proceso es desalojado y se le da el control al próximo proceso LISTO.

2.6.3. Parámetros de un proceso de tiempo real

Como se muestra en la figura 2.3, los parámetros asociados a un proceso de tiempo real (τ_i) son:

- **Tiempo de Activación (a_i):** Es el tiempo en el cual el proceso o tarea (τ_i) está lista para su ejecución.
- **Período de Activación (T_i):** Es el momento en que el proceso (τ_i) realiza una petición de ejecución.
- **Tiempo de Cómputo (C_i):** Es el tiempo de ejecución del proceso (τ_i).

- **Tiempo de Inicio (s_i):** Tiempo en el cual el proceso (τ_i) inicia su ejecución.
- **Tiempo de Finalización (f_i):** Tiempo en el cual el proceso (τ_i) termina su ejecución.
- **Plazo de Respuesta (D_i):** Es el tiempo permitido para que el proceso (τ_i) finalice su ejecución.
- **Criticidad:** Es un parámetro relacionado a la consecuencia de la pérdida de plazos de respuesta, o se relaciona con la importancia de las tareas dentro del conjunto de tareas.
- **Latencia (L_i):** $L_i = D_i - f_i$, representa el retrazo en la terminación de una tarea o proceso con respecto a su plazo de respuesta. Si $L_i \leq 0$ la tarea (τ_i) no pierde su plazo.
- **Prioridad (P):** Es un número que representa la importancia del proceso o la tarea dentro del sistema.

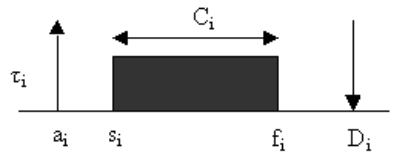

Figura 2.3: Parámetros de un Proceso de Tiempo Real

2.6.4. Estado del proceso

Durante la ejecución de un proceso, su estado cambia y por lo tanto, el estado de un proceso está definido por la actividad que ese proceso esté realizando. Como se muestra en la figura 2.4, cada proceso podría estar en uno de los siguientes estados:

- **Nuevo:** El proceso está siendo creado. En este estado, se le asignan recursos al proceso, se reserva un espacio en memoria para cargar código, datos y *pila (stack)*, y se genera su "Process Control Block" (cuyo contenido se explicará más adelante).

- **Corriendo:** Las instrucciones del proceso se están ejecutando.
- **Esperando:** El proceso está esperando por algún evento a ocurrir.
- **Listo:** El proceso está esperando para ser asignado al procesador.
- **Terminado:** Pasa a este estado, solamente si el proceso ha terminado su ejecución.

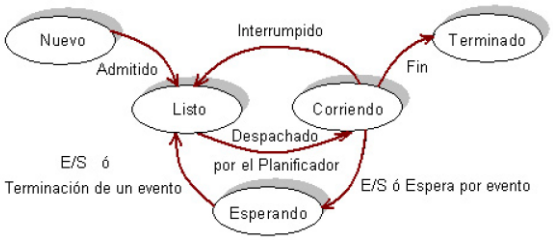

Figura 2.4: Diagrama de estado de los procesos.

Sólo un proceso a la vez puede estar corriendo en un procesador en un instante determinado, a la vez que varios procesos podrían estar en una cola de procesos listos esperando a que se les asigne el procesador. Los nombres de los estados son arbitrarios y varían entre los sistemas operativos. Los estados que aquí se presentan son mencionados en todos los sistemas operativos.

2.6.5. Transiciones entre estados

Como se muestra en la figura 2.4, los procesos pueden cambiar de estado durante su ejecución, debido a alguno de los siguientes eventos:

- **Admitido:** Esta transición ocurre cuando el proceso es creado y está listo para competir por los recursos.
- **Despachado:** Ocurre cuando el planificador elige al proceso de la cola de listo y le asigna el procesador.
- **Interrumpido:** Ocurre cuando el proceso cumple con su tiempo de computo (*Quantum*) o cuando algún proceso de mayor prioridad le quita el procesador.

- **E/S o Espera por evento:** Ocurre cuando el proceso necesita esperar por algún dispositivo de entrada/salida o por la llegada de un evento.
- **E/S o Terminación de un evento:** Es cuando el dispositivo o el evento que esperaba el proceso ya ocurrió y por lo tanto el proceso está listo para su ejecución.
- **Fin:** Ocurre cuando un proceso ha terminado de realizar todas sus tareas y por lo tanto es eliminado del sistema.

2.6.6. PCB (*Process Control Block*)

Cada proceso es representado en el sistema operativo por un Bloque de Control de Procesos (PCB) o también conocido como Bloque de Control de Tareas. Un PCB contiene muchas piezas de información asociadas con un proceso específico (fig .2.5).

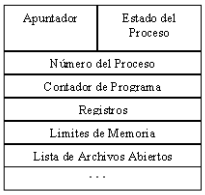

Figura 2.5: Bloque de Control de Procesos (PCB)

Esta información es la siguiente:

- **Estado del Proceso:** El estado podría ser nuevo, listo, corriendo, esperando, bloqueado, etc.
- **Contador de Programa:** Aquí se indica la dirección de la siguiente instrucción a ser ejecutada en el procesador.
- **Registros del CPU:** Los registros varían en número y tipo, dependiendo de la arquitectura de la computadora. Aquí se incluyen acumuladores, apuntadores a pilas, registros índices y registros de propósito general.
- **Información del Planificador de CPU:** Esta información incluye la prioridad del proceso, apuntadores a las colas de planificación y cualquier otro parámetro de planificación.

- **Información para el Manejo de Memoria:** Esta información podría incluir la tabla de páginas o segmentos y los registros límites. Todo depende de la memoria en el sistema utilizada por el sistema operativo.

- **Información Contable:** Esta información contiene la cantidad de CPU y tiempo real usado, limites de tiempo, números de procesos, etc.

- **Información del estado de I/O:** Aquí se encuentra la lista de dispositivos de entrada o salida (I/O) empleados por el proceso y la lista de archivos que ha abierto.

2.7. Elementos de un sistema operativo de tiempo real

2.7.1. Manejador de procesos

Un proceso es un programa en ejecución el cual necesita de ciertos recursos (procesador, memoria, archivos, dispositivos E/S) para lograr su actividad. El manejador de procesos proporciona servicios primordiales que todo sistema operativo debe proveer. Este incluye varias funciones de soporte como las de creación, terminación, planificación y despacho de procesos, así como el manejo del cambio de contexto [7].

Creación de procesos

Un proceso durante su ejecución puede crear muchos procesos nuevos mediante una llamada al sistema a la rutina de creación de procesos. El proceso que los creó es llamado *proceso padre* y a los nuevos procesos se les llama *hijos*. Cada uno de estos procesos hijos, pueden a su vez crear nuevos procesos formándose así un árbol de procesos. Generalmente, un proceso necesita de ciertos recursos (tiempo de CPU, memoria, manejo de archivos y dispositivos de E/S), por lo que cuando un proceso crea un subproceso, el subproceso podría ser capacitado para obtener sus recursos directamente desde el sistema operativo o ser obligado a tener un subconjunto de recursos desde el proceso padre.

Terminación de un proceso

Un proceso se termina cuando éste finaliza su ejecución, en este punto el proceso le pide al sistema operativo que lo borre, regresa los datos a su proceso padre y todos los recursos que el proceso ocupaba (memoria física y virtual, archivos, *buffers de E/S*, etc.) son devueltos al sistema operativo.

Cambio de contexto

En un sistema operativo de (tipo) multiprogramación, se maneja el concepto de *contexto* para mantener información del estado en que se encuentran cada uno de los procesos. El *cambio de contexto* ocurre cuando un proceso es interrumpido de su ejecución para que otro proceso pueda utilizar el procesador (fig. 2.6). Esta interrupción puede ocurrir cuando al proceso se le termina su *quantum*, o cuando un proceso de mayor prioridad demanda al procesador.

Los pasos que se realizan en el cambio de contexto son los siguientes:

1. Se graba el estado de la tarea que está en ejecución.
2. Se utiliza al planificador para conocer qué tarea es la siguiente a ejecutar.
3. Se carga el estado de la siguiente tarea para que al finalizar el cambio de contexto sea ésta quien ahora haga uso del procesador.

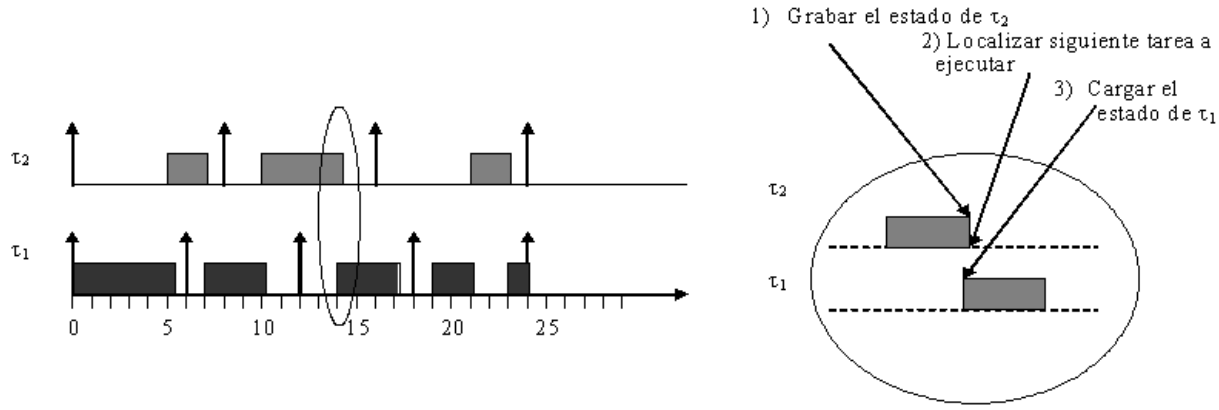

Figura 2.6: Cambio de Contexto.

El cambio de contexto es considerado como *sobrecarga*, ya que, durante el tiempo que éste se lleva a cabo, el sistema queda incapacitado para realizar alguna operación útil.

2.7.2. Manejador de memoria

La memoria es un arreglo de *words* o *bytes* en donde cada uno cuenta con una dirección propia, siendo además un dispositivo de acceso rápido por el procesador o los dispositivos de entrada/salida. El procesador la utiliza para leer las instrucciones de programa que se encuentran almacenadas aquí y además le sirve para escribir información temporal. Los dispositivos de E/S la utilizan para leer y escribir mediante el DMA (Acceso Directo a Memoria). Las funciones que el manejador de memoria debe realizar son:

- Decidir qué procesos cargar en memoria cuando ésta esté disponible.
- Conocer qué partes de la memoria están siendo utilizadas y por quién.
- Poder liberar y conceder espacio de memoria cuando se le requiera.

Los mecanismos que se utilizan para administrar la memoria pueden variar (paginación, segmentación, reemplazo de páginas, etc). La elección de estos mecanismos depende en gran parte del hardware para el cual se esté diseñando el sistema operativo de tiempo real. Por ejemplo, en los sistemas empotrados la memoria es escasa y todos los procesos se encuentran residentes en memoria, por lo que no es necesario tener un administrador de memoria complejo.

2.7.3. Manejador de reloj

El manejador de reloj realiza diferentes acciones de acuerdo a las interrupciones que genera el reloj. Entre las funciones que controla este manejador se encuentran:

- Provee información para el calendario de procesos (planificadores).
- Mantiene actualizada la fecha y la hora del día.
- Maneja las alarmas.

Como se puede ver, el reloj de tiempo real permite al sistema configurar los tiempos para la planificación de las tareas. Además, mediante el manejador de reloj es posible configurar interrupciones para controlar dispositivos externos, recibir y sincronizar señales de comunicación y monitorear el cumplimiento de los requisitos temporales de las tareas del sistema.

2.7.4. Mecanismos de sincronización y comunicación

Es otro mecanismo básico que todo kernel debe incluir. La *comunicación* provee mecanismos para permitir que los procesos se comuniquen y se sincronicen. La *sincronización* evita la inconsistencia de datos, el cual es un problema muy común en los sistemas concurrentes. La manera clásica para resolver la sincronización es utilizando mecanismos de exclusión mutua o semáforos. La comunicación se realiza a través de memoria compartida o por paso de mensajes mediante buzones.

Sección crítica

La sección crítica es una secuencia de instrucciones con un comienzo y un final claramente marcados que, por lo general, delimita la actualización de una o más variables compartidas. Cuando un proceso entra en una sección crítica, debe ejecutar todas las instrucciones incluidas en ella antes de que se pueda permitir a cualquier otro proceso entrar a la misma sección crítica.

Exclusión mutua

Se le llama exclusión mutua al hecho de que sólo el proceso que ejecuta la sección crítica tiene permitido el acceso a la variable compartida. Cuando un proceso se encuentra en exclusión mutua, a los demás procesos se les prohibe esa sección hasta que el proceso que está dentro la libera, dicho de otra manera, un proceso puede excluir temporalmente a todos los demás de utilizar un recurso compartido con el fin de asegurar la integridad del sistema.

Semáforos

El concepto de semáforo fue inventado por Dijkstra para solucionar el problema de la exclusión mutua. El mecanismo semáforo consta básicamente

de dos operaciones primitivas, *señal (signal) y espera (wait)*[2] que operan sobre un tipo especial de variable semáforo, S. La variable semáforo puede tomar valores enteros y sólo puede ser accedida y manipulada por medio de las operaciones *señal* y *espera*, con la excepción del valor en su inicialización. Ambas primitivas llevan un argumento cada una (la variable semáforo), y se definen de la siguiente forma:

- **Wait(S):** Es una operación que decrementa el valor de su argumento semáforo, S. La operación WAIT es indivisible. Si después de decrementar el semáforo, el valor de S es negativo, el proceso que llama a esta primitiva se bloquea (fig. 2.7).

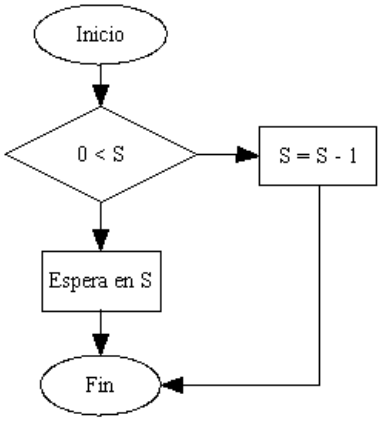

Figura 2.7: Operación WAIT.

- **Signal(S):** Es una operación que incrementa el valor de su argumento semáforo (S) siempre y cuando, no haya procesos bloqueados en la cola del semáforo. Si los hay, además de incrementar el valor del semáforo, desbloquea al proceso. La operación SIGNAL también es indivisible (fig. 2.8).

Un semáforo cuya variable sólo tiene permitido tomar los valores 0 (ocupado) y 1 (libre) se denomina *semáforo binario*. Para los semáforos binarios, la lógica de Wait(S) debería interpretarse como la espera hasta que la variable semáforo S sea igual a LIBRE, seguido de su modificación indivisible

[2]Las primitivas señal y espera originalmente fueron definidas como P y V (señal y espera en holandés) por Dijkstra.

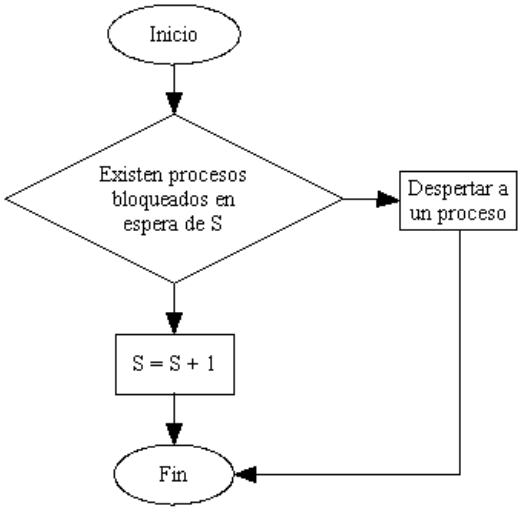

Figura 2.8: Operación SIGNAL.

para que se indique OCUPADO antes de devolver el control al invocador. La operación de WAIT implementa, por tanto, la fase de negociación del protocolo de exclusión mutua. SIGNAL pone el valor de la variable semáforo a LIBRE y representa por tanto la fase de liberación en la secuencia de exclusión mutua descrita anteriormente.

Comunicación entre procesos

Para que los procesos puedan comunicarse es necesario tener una forma de referenciarse unos con otros. Para lograr esto, existen mecanismos de comunicación que operan a través de comunicación directa o mediante la comunicación indirecta.

Comunicación directa (paso de mensajes)

En la comunicación directa, cada proceso que quiera comunicarse debe especificar el nombre del proceso con el que se desea comunicar de la siguiente manera:

send*(P,message)*. Esto indica que se quiere enviar un mensaje al proceso P.
receive*(Q,message)*. Esto indica que se quiere recibir un mensaje que provenga del proceso Q.

Este esquema de comunicación tiene una simetría en el direccionamiento, esto es, ambos procesos tanto el que envía como el que recibe, conocen el nombre del otro proceso con el que se va a comunicar. Existe una variante dentro de la comunicación directa en la que se tiene un direccionamiento asimétrico, esto es, que sólo el que envía conoce el nombre del proceso destino.

send*(P,message)*. Envía un mensaje al proceso P.
receive*(id,message)*. Recibe un mensaje de cualquier proceso; la variable *id* es utilizada para almacenar el nombre del proceso con el que se tuvo comunicación.

Comunicación indirecta (buzones)

Con la comunicación indirecta, los mensajes son enviados y recibidos desde buzones (también conocidos como puertos). Un buzón puede ser visto de manera abstracta como un objeto en el cual los procesos pueden colocar sus mensajes, así también, como un objeto donde los procesos pueden recibir sus mensajes. Algunos métodos para los sistemas operativos de tiempo real, ven a los buzones como la forma más eficiente de implementar comunicaciones entre procesos. Un ejemplo de esto, es el sistema operativo de tiempo real llamado iRMX de TenAsys [8], el cual suministra un buzón basado en memoria que permite tratar con eficacia la transferencia de datos. Este buzón, es un lugar para enviar y recibir punteros a los datos con la finalidad de eliminar la necesidad de transferir todos los datos, ahorrando así tiempo y sobrecarga.

En este esquema los procesos pueden comunicarse con cualquier otro proceso a través de uno o varios buzones. Como regla general, dos procesos pueden comunicarse sólo si alguno de ellos tiene un buzón compartido y por lo tanto, las primitivas de *envía (send)* y *recibe (recibe)* quedarían definidas como sigue:

send*(A,message)*. Enviar un mensaje al buzón A.
receive*(A,message)*. Recibe un mensaje desde el buzón A.

De esta manera, cualquier proceso que comparta el mismo buzón podrá recibir el mensaje.

2.7.5. Manejador de entradas/salidas

Existen diferentes tipos de manejadores de acuerdo a los tipos de dispositivos presentes en el sistema. Los manejadores de dispositivos son los encargados de comunicarse con los dispositivos de Entrada/Salida con la finalidad de poder realizar las operaciones que estos requieran. La comunicación entre los dispositivos y el sistema operativo es llevada a cabo principalmente a través de interrupciones.
El sistema de entradas y salidas consiste de:

- Un sistema de memoria cache mediante *buffers*.
- Una interfaz general con los manejadores de los dispositivos.
- Los manejadores para dispositivos de Hardware específico.

2.8. Planificación en sistemas de tiempo real

Cuando existen actividades con restricciones temporales, como ocurre en los sistemas de tiempo real, el principal problema a resolver sería el *planificar* esas actividades para poder cumplir con sus restricciones temporales. Para realizar esta planificación se necesitará de un *algoritmo de planificación*, el cual, no es mas que un conjunto de reglas que determinan qué tarea se debe ejecutar en cada instante. Estos algoritmos deben tener en cuenta las necesidades de recursos y tiempo de las tareas de tal manera que éstas cumplan ciertos requisitos de prestaciones.
Los objetivos a alcanzar por toda política de planificación de tiempo real son:

- El garantizar la correcta ejecución de todas las tareas críticas.
- Administrar el uso de recursos compartidos.

2.9. Tareas de tiempo real

El sistema operativo de tiempo real en relación con la planificación, considera a las tareas como procesos que consumen cierta cantidad de tiempo de procesador, y ciertos recursos del sistema. Para el planificador, los datos que necesita cada tarea así como el código que ejecuta y los resultados que producen, son totalmente irrelevantes [9].

2.9.1. Clasificación de las tareas de tiempo real

En base a los parámetros mostrados en la sección 2.6.3 se pueden realizar distintas clasificaciones de las tareas de tiempo real. De todas las clasificaciones que existen, la que más destaca es aquella en donde se clasifica con base a la necesidad del cumplimiento de su plazo por parte de las tareas. La división de las tareas queda de la siguiente manera:

- **Tareas de tiempo real duras (*hard*):** Estas tareas deben cumplir siempre con sus plazos de respuesta, por lo que un fallo en el cumplimiento es intolerable por sus consecuencias en el sistema controlado. Por ejemplo, en un sistema de control de un automóvil, el proceso encargado de inflar la bolsa de aire debe comportarse de tal manera que nunca pasen más de 2 milisegundos desde que se detecta una colisión hasta que se produce su respuesta. Si se supera ese límite, la respuesta del proceso tendría un valor negativo ya que la bolsa de aire se abriría cuando ya no fuera necesaria.

- **Tareas de tiempo real suaves (*soft*):** Para estas tareas el no cumplir con sus plazos de respuesta, produce una disminución en el rendimiento o en la calidad de respuesta, pero el funcionamiento se puede considerar todavía correcto. Es decir, se acepta que en alguna de las ejecuciones de la tarea exista una tardanza, sabiendo también que a medida que aumenta el tiempo de la tardanza el resultado cada vez sería menos útil.

Otra característica relacionada con los parámetros temporales, concierne a la *regularidad de la activación* de las tareas. Tomando en cuenta la regularidad en la ejecución de las tareas de tiempo real, éstas se pueden clasificar de la siguiente manera:

- **Periódicas:** Las tareas periódicas se ejecutan repetidamente a intervalos de tiempo regulares (fijos) llamados instancias. En cada instancia se ejecuta un *Job* de la tarea de tiempo real. Al inicio de cada instancia el *job* se encuentra listo para ejecución (fig. 2.9).

- **Aperiódicas:** Las tareas aperiódicas se activan de forma irregular al producirse determinados eventos de forma imprevisible. Las tareas aperiódicas se ejecutan sólo durante una instancia de ejecución al termino

de la cual desaparecen del sistema. Las tareas aperiódicas, no tienen restricciones críticas.

- **Esporádicas:** Las tareas esporádicas son tareas aperiódicas con restricciones temporales críticas (o duras). Si se monitorea el arribo de las tareas esporádicas es posible determinar una separación mínima entre activaciones consecutivas, lo cual podría permitir caracterizarlas como tareas periódicas (fig. 2.9).

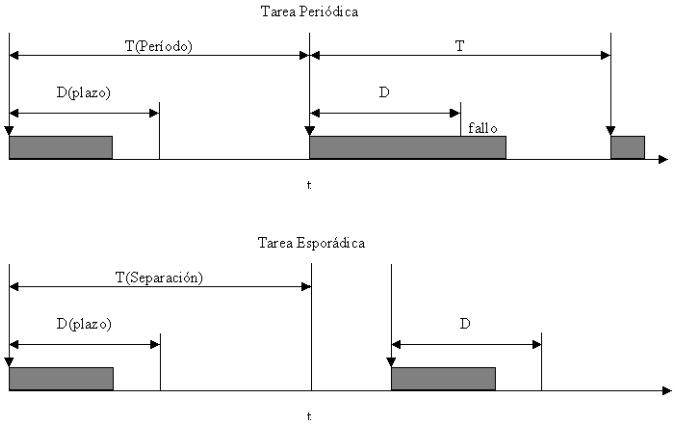

Figura 2.9: Tareas Periódicas y Esporádicas.

Otras clasificaciones de las tareas de tiempo real son las siguientes:

- **Expulsables y no expulsables:** Las tareas expulsables son aquellas que durante su ejecución pueden ser interrumpidas por el planificador debido a que se necesita ejecutar otra tarea de mayor prioridad. Por otro lado, las tareas no expulsables no podrán ser interrumpidas sino hasta que estas terminen su ejecución.

- **Con restricciones de precedencia:** Las tareas con restricciones de precedencia presentan un orden de ejecución con respecto a otras tareas del sistema. Si la tarea τ_i precede a la tarea τ_j, significa que la tarea τ_i sólo puede ejecutarse hasta que la tarea τ_j termine su ejecución. Este tipo de tareas suelen aparecen en sistema multiprocesadores o sistemas distribuidos, en los cuales varias tareas cooperan para proporcionar la respuesta a los eventos del sistema.

2.9.2. Tipos de restricciones de las tareas de tiempo real

Restricciones de tiempo

Los sistemas de tiempo real se caracterizan principalmente por tener restricciones que afectan el tiempo de ejecución de sus actividades. El *plazo de respuesta (o deadline)* es una restricción de tiempo muy común en las tareas de tiempo real, la cual representa el mayor tiempo que una tarea tiene para completar su actividad sin causar daño al sistema.

Restricciones de precedencia

Las aplicaciones de tiempo real, no pueden ser ejecutadas en orden arbitrario ya que tienen que respetar alguna relación de precedencia. Las relaciones de precedencia son descritas por un grafo dirigido acíclico G, donde las tareas son representadas por nodos y las relaciones de precedencia se representan por medio de vértices. Un grafo de precedencia G origina un orden sobre el conjunto de tareas.

- La notación $\tau_a < \tau_b$ establece que la tarea τ_a es la antecesora de la tarea τ_b, lo que significa que G tiene un camino directo del nodo τ_a al nodo τ_b.
- La notación $\tau_a \rightarrow \tau_b$ establece que la tarea τ_a es la antecesora inmediata de τ_b, lo que significa que G tiene un vértice directo de la tarea τ_a a la tarea τ_b.

La figura 2.10 muestra un grafo acíclico que describe las restricciones de precedencia de cinco tareas. Como puede observarse la tarea τ_1 es la única que puede iniciar su ejecución debido a que no tiene tareas antecesoras. Cuando τ_1 ha completado su cómputo, entonces τ_2 o τ_3 pueden iniciar su ejecución. La tarea τ_4 puede iniciar su ejecución sólo cuando τ_2 ha terminado, mientras que τ_5 tendrá que esperar hasta que τ_2 y τ_3 terminen su ejecución.

Restricciones de recursos

Un recurso es una estructura de software o de hardware que puede ser usada en forma concurrente por varios procesos. Típicamente, un recurso puede ser una estructura de datos, un conjunto de variables, un área de memoria,

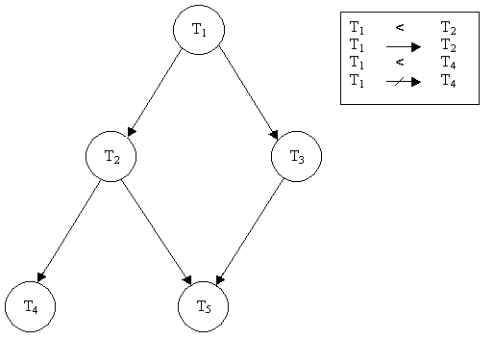

Figura 2.10: Relación de precedencia para cinco tareas.

un archivo, un fragmento de un programa, o un dispositivo periférico. Un recurso dedicado exclusivamente a un proceso particular es llamado *recurso privado o exclusivo*, mientras que un recurso que puede ser utilizado por una o más tareas se conoce como *recurso compartido*.

Cuando varios procesos accesan a un *recurso exclusivo*, su acceso debe de darse en forma ordenada y sincronizada. Esto se debe a que sólo un proceso a la vez puede estar haciendo uso de este recurso. Esta situación puede motivar perdidas de plazos, si algún proceso de baja prioridad hace uso de un recurso exclusivo por largos periodos de tiempo. Cuando se hace uso de *recursos compartidos*, varios procesos a la vez pueden accesar a dichos recursos. En el caso de los recursos compartidos sean datos (o bases de datos), un factor importante a mantener es la consistencia y la integridad en los datos.

Para mantener la consistencia de datos en *recursos compartidos*, no debe permitirse el acceso simultáneo por dos o más tareas a estos datos. En este caso deben implementarse mecanismos de *exclusión mutua* entre las tareas que compiten por este recurso. Supongamos que R es un recurso compartido exclusivo para las tareas τ_a y τ_b. Si A es una operación realizada por τ_a sobre R, y B es la operación realizada por τ_b sobre R, entonces A y B nunca pueden ejecutarse al mismo tiempo. Un fragmento de código ejecutado bajo restricciones de exclusión mutua se le conoce como *sección crítica*.

Para asegurar el acceso secuencial sobre recursos compartidos, los sistemas operativos normalmente proveen mecanismos de sincronización (tales como semáforos) que puedan ser usados por las tareas para crear regiones críticas.

2.10. Definición del problema de planificación

Para definir el problema de planificación necesitamos especificar tres conjuntos: un conjunto de n tareas $\Sigma = \{\tau_1, \tau_2,...\tau_n\}$, un conjunto de m procesadores $P=\{P_1, P_2,...P_m\}$ y un conjunto de recursos $R=\{R_1, R_2, ...R_s\}$. Además, es necesario especificar las relaciones de precedencia entre tareas y las restricciones de tiempos de cada tarea [7]. En este contexto, la planificación permite asignar los procesadores del conjunto P y los recursos del conjunto R a tareas del conjunto Σ de acuerdo a un orden que permita completar las tareas bajo las restricciones impuestas. Se ha demostrado que este problema es de tipo NP-completo, lo que significa que la solución es muy difícil de obtener. Para reducir la complejidad en la construcción de planificadores, podemos simplificar la arquitectura de la computadora, por ejemplo, restringiendo al sistema para que utilice solamente un procesador, adoptar un modelo con desalojo, usar prioridades fijas, eliminar precedencias o restricciones de recursos, asumir una activación simultánea de tareas y suponer que en el sistema existen conjuntos de tareas homogéneos (únicamente tareas periódicas o tareas aperiódicas).

2.11. Clasificación de políticas de planificación

En los sistemas de tiempo real existen muchas estrategias de planificación de tareas, sin embargo, estas pueden ser clasificadas en dos grandes grupos: los planificadores cíclicos y los planificadores basados en prioridades (fig. 2.11).

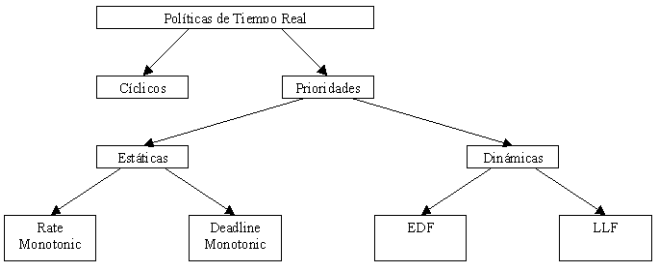

Figura 2.11: Clasificación de los algoritmos de planificación para sistemas de tiempo real.

Los planificadores cíclicos consisten en construir un plan de ejecución que

se repite cíclicamente. Este plan de ejecución se divide en un *ciclo principal* T_M, el cual a su vez se divide en *ciclos secundarios* con período T_S. En cada ciclo secundario se ejecutan las actividades correspondientes a cada tarea. El principal problema que presentan los ejecutivos cíclicos es la poca flexibilidad en el momento de modificar parámetros (añadir o borrar tareas), ya que esto conlleva a la re-elaboración de todo el plan.

En los planificadores basados en prioridades, la asignación de la prioridad de una tarea puede realizarse de forma estática (la asignación de prioridades se realiza al principio de la ejecución del sistema, y no cambia durante la ejecución.) o dinámica (la asignación de prioridades se realiza en forma dinámica durante la ejecución del sistema).

2.12. Planificador cíclico

En la planificación cíclica, todas las tareas del sistema se planifican dentro de un plan de ejecución. En este plan de ejecución existe un ciclo principal, y varios ciclos secundarios. La forma de calcular los ciclos principal y secundarios se describe mediante el siguiente ejemplo: Para construir el plan de ejecución, se debe calcular la duración del ciclo principal T_M y la duración de los ciclos secundarios T_S. La duración del ciclo principal corresponde al mínimo común múltiplo de los períodos de las tareas, $T_M = mcm(\tau_i)$. Resultando para el ejemplo de la tabla 2.1, $T_M = 100$.

$Tarea$	T	C
A	25	10
B	25	8
C	50	5
D	50	4
E	100	2

Tabla 2.1: Conjunto de tareas periódicas

Por otro lado, para calcular el tamaño de los ciclos secundarios, m, se deben considerar las siguientes condiciones:

- El ciclo secundario debe ser menor o igual que el plazo de ejecución de cada tarea, $m \leq \{D_i\}$.

- El ciclo secundario debe ser mayor o igual que el máximo de los tiempos de cómputo de las tareas, $m \geq max\{C_i\}$.
- El ciclo secundario debe ser divisor del período de cada tarea. Es decir, m divide a T_i.
- Se debe cumplit que: $2m - mcd(m, T_i) \leq D_i$, la cual es una condición necesaria y suficiente que indica el instante de activación de las tareas.

Aplicando las condiciones anteriores al ejemplo de la tabla 2.1 se obtiene:

- $m \leq \{D_i\} \Longrightarrow m = \{1, 2, 3, ..., 25\}$
- $m \geq max\{C_i\} \Longrightarrow m = \{10, 11, 12, ..., 25\}$
- m divide a $T_i \Longrightarrow m = \{10, 20, 25\}$
- $2m - mcd(m, T_i) \leq D_i \Longrightarrow m = \{10, 20, 25\}$

Con los valores obtenidos para m se puede calcular el número de ciclos secundarios del ciclo principal mediante $N_{T_S} = T_M/m$. Para $m = 10$ se tiene $N_{T_S} = 10$, para $m = 20$ se tiene $N_{T_S} = 5$ mientras que para $m = 25$ se tiene $N_{T_S} = 4$. Como la complejidad aumenta con el número de ciclos secundarios se elige $m = 25$ con lo que se tiene 4 ciclos secundarios por ciclo principal. Otro dato que se puede calcular es el número de ejecuciones N_{e_i} de cada tarea τ_i en el ciclo principal, mediante la expresión $N_{e_i} = T_M/T_i$. En el ejemplo de la tabla 2.1 el número de ejecuciones de cada tarea es $N_{e_{A,B}} = 4$ para τ_A y τ_B, $N_{e_{C,D}} = 2$ para τ_C y τ_D y $N_{e_E} = 1$ para τ_E.

En la figura 2.12 se presenta gráficamente la ejecución cíclica para el conjunto de tareas presentado en la tabla 2.1.

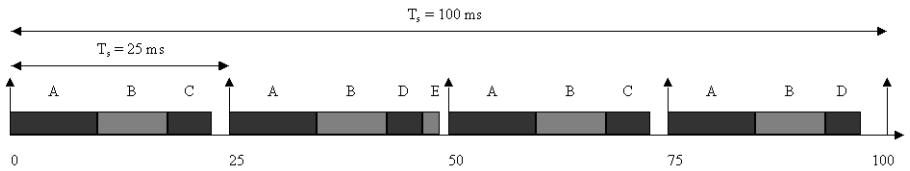

Figura 2.12: Planificación Cíclica.

2.13. Planificadores basados en prioridades estáticas

En esta planificación, las prioridades de las tareas se asignan en forma estática, antes de la ejecución del sistema, y no cambian durante su ejecución. En la planificación basada en prioridades estáticas (*off-line*), se realiza un análisis de planificabilidad de las tareas *fuera de linea*. Esta planificación tiene como ventajas:

- Producir sobre-cargas muy bajas, ya que la asignación de prioridades se realiza una sola vez antes de la ejecución. Las prioridades asignadas a las tareas son guardadas en una tabla la cual permite al planificador decidir el orden de ejecución de las tareas.
- Permite verificar el comportamiento temporal de las tareas a priori (predecibilidad). Es adecuada para trabajar con tareas con plazos críticos, y el análisis de planificabilidad nos permite comprobar a priori si dichas tareas cumplirán sus plazos de respuesta.
- En situaciones de sobrecarga del sistema, siempre es posible predecir que las tareas de menor prioridad serán las primeras en perder sus plazos.

Sus principales desventajas son:

- Requiere un conocimiento previo de todos los parámetros de las tareas.
- Es incapaz de tratar adecuadamente las tareas aperiódicas.
- Inflexible a operar bajo ambientes dinámicos en donde los parámetros cambian constantemente.

2.13.1. Rate Monotonic (RM)

En el algoritmo *Rate Monotonic*, la asignación de las prioridades se obtiene de acuerdo a los períodos de las tareas. A la tarea con menor período, se le asigna la mayor prioridad. En este algoritmo de planificación, el período es igual al plazo.Fueron Liu y Layland [10] quienes en 1973 propusieron el algoritmo *Rate Monotonic (RM)*, además demostraron que:

Teorema 1 *El algoritmo RM es óptimo dentro de los esquemas de asignación de prioridades estáticas.*

Por *óptimo* debemos entender que si se tiene una planificación factible para un conjunto dado de tareas mediante un algoritmo de asignación con prioridades estáticas, entonces ese conjunto de tareas también será planificable mediante el algoritmo *Rate Monotonic*.

En el análisis de este algoritmo [10], los autores proporcionan un *control de admisión* [3] de tareas para RM basado en la utilización del procesador. Este control de admisión, compara la utilización del conjunto de tareas con una *cota límite* que depende del número de tareas del sistema, es decir, un conjunto de n tareas no perderá su plazo si cumple la siguiente condición:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1) \tag{2.1}$$

Teorema 2 *La condición suficiente para que la planificación de un conjunto de n tareas periódicas sea factible mediante el algoritmo RM, es que su factor de utilización mínima* $n(2^{1/n} - 1)$ *cumpla la siguiente condición:*

$$U \leq U_{min} = n(2^{1/n} - 1) \tag{2.2}$$

La tabla 2.2 describe el comportamiento de la expresión 2.2 para distintos valores de n.

Como se puede apreciar en la tabla 2.2, al aumentar el número de tareas la utilización mínima garantizada converge en:

$$U_{min} = \ln 2 \approx 0{,}6931 \tag{2.3}$$

Consideremos el conjunto de tareas mostrado en la tabla 2.3. Como se puede observar, la utilización total del conjunto de tareas no excede la cota proporcionada por la condición 2.2, es decir, U=0.775 $<$ 0.779 $= U_{min}$. Por

[3] El control de admisión es un mecanismo que permite al planificador decidir sobre la aceptación de las tareas en el sistema. En sistemas de tiempo real con planificación estática, este mecanismo se ejecuta sólo una vez, al principio de la ejecución del sistema.

n	U_{min}
1	1
2	0.8284
3	0.7798
4	0.7568
5	0.7433
.	.
.	.
.	.
∞	0.6931

Tabla 2.2: Utilización mínima garantizada para n tareas

$Tarea$	T_i	C_i	*Utilización*
1	16	4	0.250
2	40	5	0.125
3	80	32	0.400
			0.775

Tabla 2.3: Conjunto de tareas, la utilización total cumple con la condición 2.2

lo cual, se garantiza que este conjunto de tareas no perderá ningún plazo de respuesta. La figura 2.13 muestra gráficamente el comportamiento del conjunto de tareas.

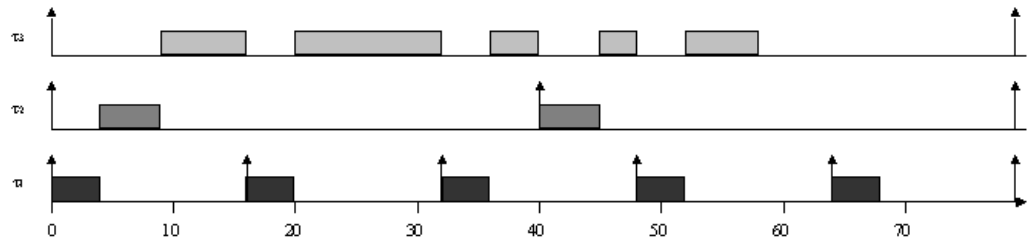

Figura 2.13: Conjunto de tareas que satisface 2.2 y por tanto es planificable.

La tabla 2.4 describe un conjunto de tareas cuya utilización del procesador es del 100 %, es decir, no satisface la condición 2.2, y aún así, el conjunto de tareas cumple con los plazos de respuesta. Es por esta razón, que a esta condición se le conoce como una condición suficiente, pero no necesaria. La

figura 2.14 muestra la ejecución de las tareas sin que estas hayan pérdido su plazo.

Tarea	T_i	C_i	*Utilización*
1	20	5	0.250
2	40	10	0.250
3	80	40	0.500
			1.000

Tabla 2.4: Conjunto de tareas armonico, es planificable

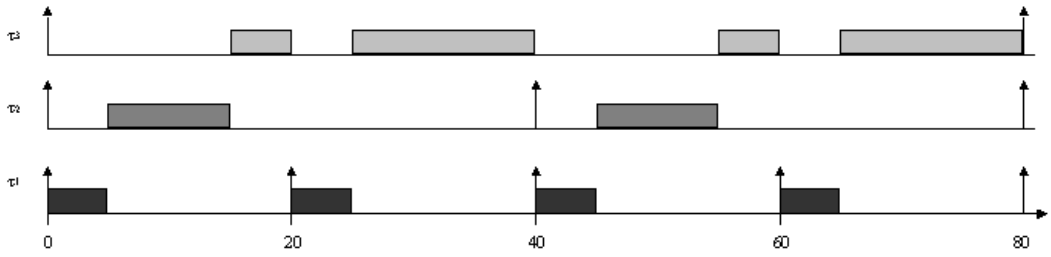

Figura 2.14: Conjunto de tareas que no satisface 2.2. Sin embargo, es planificable.

2.13.2. Deadline Monotonic (DM)

En 1982 Leung y Whitehead [11] proponen un esquema de asignación de prioridades fijas denominado Deadline Monotonic (DM), el cual consiste en asignar la prioridad más alta a las tareas que tengan plazos más cortos. Es decir, la tarea con el plazo más corto se le asigna la prioridad más alta.

En DM, el plazo puede ser menor que el periodo, sin embargo, DM es equivalente a RM cuando el período = plazo de respuesta.

Teorema 3 *[11] El algoritmo DM es óptimo dentro de los esquemas de asignación de prioridades fijas.*

Esta política de planificación es en principio idéntica al RM pero eliminando una restricción: las tareas pueden tener un plazo de ejecución menor

o igual a su período. Las prioridades se asignan de forma inversamente proporcional al plazo máximo de ejecución. Se puede ver que el RM es un caso particular del DM en el que todas las tareas tienen un plazo de ejecución igual a su período.

2.14. Planificadores basados en prioridades dinámicas

En este tipo de planificación las prioridades de las tareas son asignadas durante la ejecución del sistema, sin embargo, todas las tareas y sus parámetros temporales son conocidos desde el inicio de la ejecución. En la planificación basada en prioridades dinámicas (*on-line*, en línea), se realiza un análisis de planificabilidad en línea.
Sus principales ventajas son:

- Es posible aprovechar mejor el procesador. Debido a que el planificador asigna en forma dinámica las prioridades de las tareas, el CPU puede utilizarse de forma más eficiente que en el caso de la planificación por prioridades estáticas.

Sus principales desventajas son:

- En una sobrecarga del sistema no es posible predecir que tareas pierden sus plazos de respuesta. Lo que es más grave, es que en una sobrecarga, la perdida de plazos de algunas tareas puede producir un efecto en cascada de perdida de plazos de otras tareas.

- Se incrementa la sobrecarga causada por la planificación. Esto se debe a que la planificación continuamente tiene que ordenar las tareas por orden de prioridad, lo cual introduce sobrecarga al sistema.

2.14.1. Earliest Deadline First (EDF)

Es uno de los algoritmos más conocidos para el esquema de asignación dinámica, es el algoritmo de planificación guiado por plazos (*DDSA: Deadline Driven Scheduling Algorithm*), al cual posteriormente se le denominó como el plazo más próximo primero (*EDF: Earliest Deadline First*).

En el algoritmo EDF las prioridades se asignan en forma dinámica. La política de asignación de prioridades consiste en asignar la prioridad más alta a la tarea con plazo más cercano. Para este algoritmo la condición de planificabilidad se define a continuación.

Teorema 4 *La condición necesaria y suficiente para que un conjunto de tareas periódicas tenga una planificación factible mediante el algoritmo EDF es:*

$$U \leq 1 \tag{2.4}$$

Esta condición de planificabilidad permite que EDF consiga un factor de utilización del 100 % para los conjuntos de tareas que planifica. Por lo que podemos decir que EDF es óptimo globalmente, es decir, que si existe un algoritmo que propocione una planificación factible con un determinado conjunto de tareas periódicas, entonces EDF también proporcionará una planificación factible para dicho conjunto de tareas.

Ejemplo: La tabla 2.5 muestra un conjunto de tareas, cuya utilización total es del 91 % y la figura 2.15 muestra el comportamiento de la ejecución de las tareas bajo el algoritmo de planificación EDF. Como puede observarse no hay pérdida de plazos ya que cumple con la condición 2.4

$Tarea$	T_i	C_i	*Utilización*
1	30	10	0.333
2	30	10	0.333
3	40	10	0.25
			0.916

Tabla 2.5: Conjunto de tareas, cuya utilización es del 91 %

2.14.2. Least Laxity First (LLF)

El algoritmo LLF consiste en asignar las prioridades a las tareas de manera inversamente proporcional a su holgura. Es decir, le asigna la prioridad más alta a la tarea con la menor holgura.

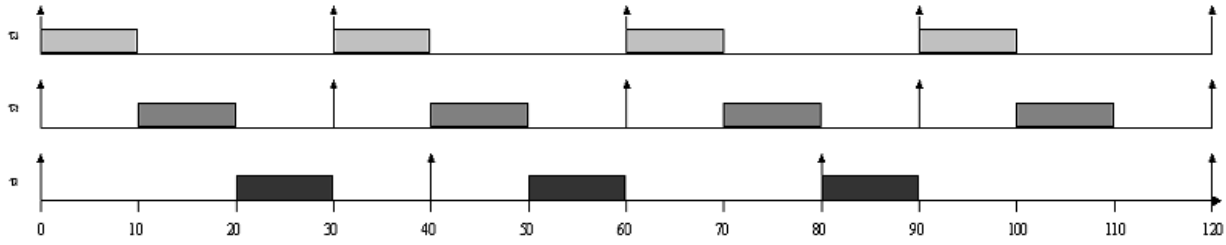

Figura 2.15: Planificación de las tareas de la tabla 2.5 bajo EDF.

La holgura (*laxity*) de una tarea con tiempo de respuesta D_i en cualquier instante de tiempo t es:

$$holgura = D_i - t - C_i(t) \tag{2.5}$$

donde $C_i(t)$, es el tiempo de cómputo pendiente por ejecutarse para que la tarea termine.

Consideremos el siguiente conjunto de tareas $\Gamma = \{\tau_1$=(6,3), τ_2=(8,2), τ_3=(7,2)$\}$. Para τ_1, en algún instante de tiempo t antes que su tiempo de cómputo finalice, su holgura es: $6 - t - (3 - t)$. Supongamos que la tarea τ_1 es desalojada en el instante de tiempo $t = 21$ por la tarea τ_2 que se ejecuta desde el tiempo 21 al 23. Durante este intervalo, la holgura de τ_1 decrece de 2 a 0. (En el tiempo 23, el resto del tiempo de ejecución de τ_1 es 1, es decir 24-23-1=0). La figura 2.16 muestra la planificación del conjunto de tareas bajo LLF.

El algoritmo LLF también puede conseguir, al igual que con EDF, un factor de utilización del 100 %. En el algoritmo EDF no es necesario conocer de los tiempos de cómputo de las tareas, mientras que el algoritmo LLF si se necesita de dicho parámetro temporal. Esto es una desventaja debido a que la holgura es obtenida con base al tiempo máximo de cómputo, es decir, la determinación de la holgura es inexacta ya que el algoritmo asume el peor caso para calcularla.

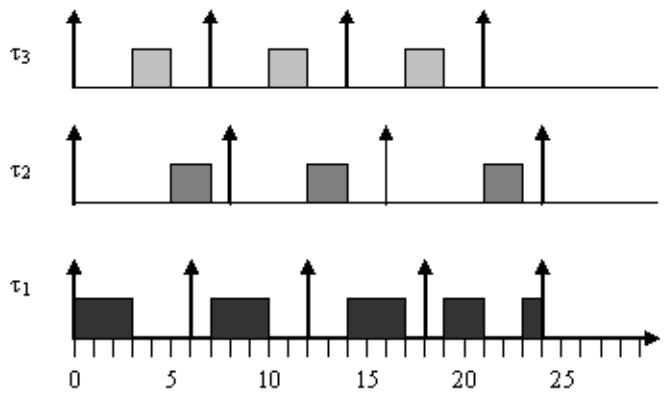

Figura 2.16: Planificación de tareas bajo el algoritmo LLF.

Capítulo 3

Kernel de tiempo real en la PDA

En el capítulo dos se presentaron las restricciones impuestas a los procesos de tiempo real, y los distintos algoritmos de planificación de propósito específico empleados para lograr el cumplimiento de dichas restricciones.

Este capítulo presenta el diseño de la capa de abstracción de hardware de RTAI en la PDA.
Para la implantación del kernel de tiempo real en la PDA se tiene diferentes posibilidades, por ejemplo [12]:

- Construir el kernel desde cero.
- Modificar un sistema operativo existente.

El primer enfoque es interesante, pero complicado. Al seguir ese camino, a pesar de tener conocimiento completo del sistema operativo, sería necesario implementar herramientas para el desarrollo y depuración de aplicaciones en él, lo cual es una tarea difícil.

El segundo enfoque es más simple, y por tanto más atractivo. Actualmente Linux es uno de los sistemas operativos de propósito general más populares, tanto en ámbitos científicos como comerciales; no sólo debido a que mantiene su código fuente abierto, sino también a que cuenta con una gran cantidad de herramientas que permiten el desarrollo, optimización y prueba de las aplicaciones [13]. Por lo anterior, Linux es una buena opción para el desarrollo de nuestro sistema operativo, de esta forma tendremos soporte a los procesos

de tiempo real, pero se mantendrá disponible todo el soporte a hardware y protocolos con los que cuenta Linux.

Al agregar soporte de tiempo real a Linux se tienen las siguientes opciones. La primera es realizar modificaciones a los algoritmos de planificación en el kernel, o reducir tiempos de latencia en procesamiento de interrupciones y cambio de contexto [14]. Sin embargo, al realizar este tipo de cambios se pierde la generalidad del kernel de Linux.

La segunda opción, es el enfoque del kernel dual, seguido en RTLinux y RTAI. Al igual que el código del kernel de Linux, el código fuente de RTAI es abierto y se encuentra disponible para arquitecturas similares a la empleada en este proyecto, por lo que se decidió realizar nuestro diseño haciendo uso del enfoque empleado en dicho sistema.

3.1. RTAI en la PDA

En la figura 3.1 se presenta un diagrama general de la forma en cómo RTAI debe de ser implementado en la PDA para que sea posible incluir el kernel de tiempo real en ella.

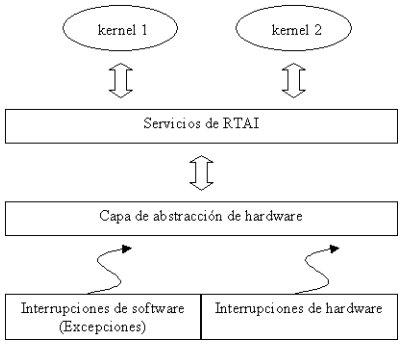

Figura 3.1: Diagrama general de la implementación de RTAI en la PDA.

La figura 3.1 muestra que RTAI consta de dos módulos, la capa de abstracción de hardware y el conjunto de servicios de RTAI. La capa de abstracción de hardware contiene referencias a un grupo de funciones empleadas para habilitar, deshabilitar y procesar a las interrupciones. Tanto Linux como el kernel de tiempo real, hacen uso de las funciones referenciadas en la capa de abstracción de hardware para interactuar con el hardware (que se encuentra en la parte inferior de la arquitectura propuesta).

Las funciones a las que hacen referencia los apuntadores de la capa de abstracción de hardware, pueden corresponder a funciones con el mismo comportamiento que las funciones originales de Linux, si solamente Linux se está ejecutando en el procesador, o bien, corresponden a funciones incluidas en el módulo de RTAI, si el kernel de tiempo real se encuentra en ejecución en el procesador.
El módulo de RTAI contiene funciones para habilitar y deshabilitar de manera lógica a las interrupciones, así como para el procesamiento de las mismas. También incluye funciones para el cambio de entorno entre Linux y el kernel de tiempo real, las cuales se encargan de modificar el valor de los apuntadores en la capa de abstracción de hardware (ver sección 1.2.2).
A diferencia de la capa de abstracción de hardware, el módulo de RTAI no es dependiente de arquitectura y sólo es necesario que la capa de abstracción de hardware se encuentre operando de manera adecuada para poder hacer uso de él.

Las distribuciones más recientes de RTAI no incluyen versión de la capa de abstracción de hardware que pueda ser incluida en el kernel de Linux que se ejecuta en la iPaq H3950, por lo que fue necesario implantarla. Las siguientes secciones describen los aspectos más relevantes en la implementación de la capa de abstracción de hardware en nuestra PDA.

3.2. Interrupciones en el procesador PXA250

Una interrupción es un evento que altera la secuencia de instrucciones a ejecutar por el procesador. Tales eventos corresponden a señales eléctricas generadas por circuitos internos y externos al procesador. Cuando se genera una interrupción, el procesador debe detener lo que está haciendo y ejecutar las instrucciones relativas a la interrupción. Iniciar la ejecución de una interrupción es similar a un cambio de contexto entre procesos, sin embargo, el código ejecutado por el manejador de una interrupción no es un proceso, sino una secuencia de instrucciones del kernel ejecutadas como parte del proceso en ejecución [15].

La iPaq H3950 cuenta con un procesador *Intel XScale* [1], el cual soporta siete tipos de interrupciones [2], la tabla 3.1 muestra una lista de ellas. Cuando

[1] Procesador basado en la arquitectura ARM
[2] En la literatura las llaman excepciones

se genera una interrupción, la ejecución de instrucciones se continúa en una dirección de memoria fija, llamada vector de interrupción, que depende del tipo de interrupción [16].

Excepción	Dirección del Vector	Dirección Alternativa del Vector
Reset	0x00000000	0xFFFF0000
Instrucción no definida	0x00000004	0xFFFF0004
Interrupción de software	0x00000008	0xFFFF0008
Error al leer instrucción	0x0000000C	0xFFFF000C
Error al leer datos	0x00000010	0xFFFF0010
Interrupción	0x00000018	0xFFFF0018
Interrupción Rápida	0x0000001C	0xFFFF001C

Tabla 3.1: Interrupciones en la arquitectura Intel XScale.

Tres tipos de interrupciones relevantes para la implementación de la capa de abstracción de hardware en el iPaq son: las interrupciones de software, la petición de interrupción y la petición de interrupción rápida, descritas a continuación:

- **Interrupción de software:** Este tipo de interrupciones pueden ser generadas por los programas de usuario a través de la instrucción *swi*, permiten llevar a cabo una petición para la ejecución de una función del sistema operativo (*llamada al sistema*).
- **Petición de interrupción (IRQ):** Generada externamente mediante la habilitación de la terminal IRQ del procesador. Estas interrupciones son habilitadas y deshabilitadas modificando el valor del bit I en el registro de control del procesador.
- **Petición de interrupción rápida (FIQ):** Generada externamente con la habilitación de la terminal FIQ del procesador. Estas interrupciones son diseñadas para transferencia de datos desde dispositivos que cuentan con varios registros propios, lo que evita hacer uso de registros del procesador y la necesidad de salvarlos, reduciendo la sobrecarga de este proceso. Son habilitadas y deshabilitadas mediante el bit F del registro de control del procesador.

3.3. Procesamiento de interrupciones en Linux

El procesamiento de interrupciones en Linux conceptualmente puede dividirse en dos módulos. El primero de ellos, llamado manejador de excepciones, se encarga del manejo de las interrupciones de software. El segundo, conocido como manejador de interrupciones, se hace cargo de atender las interrupciones generadas a través de las terminales externas al procesador (IRQs y FIQs).

3.3.1. Manejadores de excepción

Cuando un proceso de usuario genera una interrupción de software (con la instrucción *swi*), el procesador continua la ejecución de instrucciones en la dirección indicada en la tabla 3.1. La figura 3.2 presenta un diagrama de flujo de la función referenciada en esta dirección, con la que Linux sirve las llamadas a sistema.

3.3.2. Manejadores de interrupción

Tabla descriptora de interrupciones

La Tabla Descriptora de Interrupciones (*IDT-Interrupt Descriptor Table*), es una tabla presente en el kernel de Linux que asocia cada vector de interrupción con la función encargada de atender la interrupción (Manejador de Interrupción o Manejador), cada entrada en la tabla corresponde a un vector de interrupción [15]. Al generarse la *i-esima* interrupción, en el registro contador de programa se carga la dirección del manejador al que se hace referencia en la *i-esima* entrada de la IDT. Debido a las limitaciones en el hardware, varios dispositivos pueden compartir la misma línea de interrupción. Por lo que, la función manejadora de interrupción debe de ser capaz de atender a más de un dispositivo. Para este propósito, es posible asociar a cada manejador varias rutinas de servicio a interrupción (*ISR-Interrupt Service Routine*).

Manejadores de interrupción

Como se planteó, cada interrupción tiene asociada una función manejadora de interrupción, la figura 3.3 muestra las tareas realizadas en cada una de ellas.

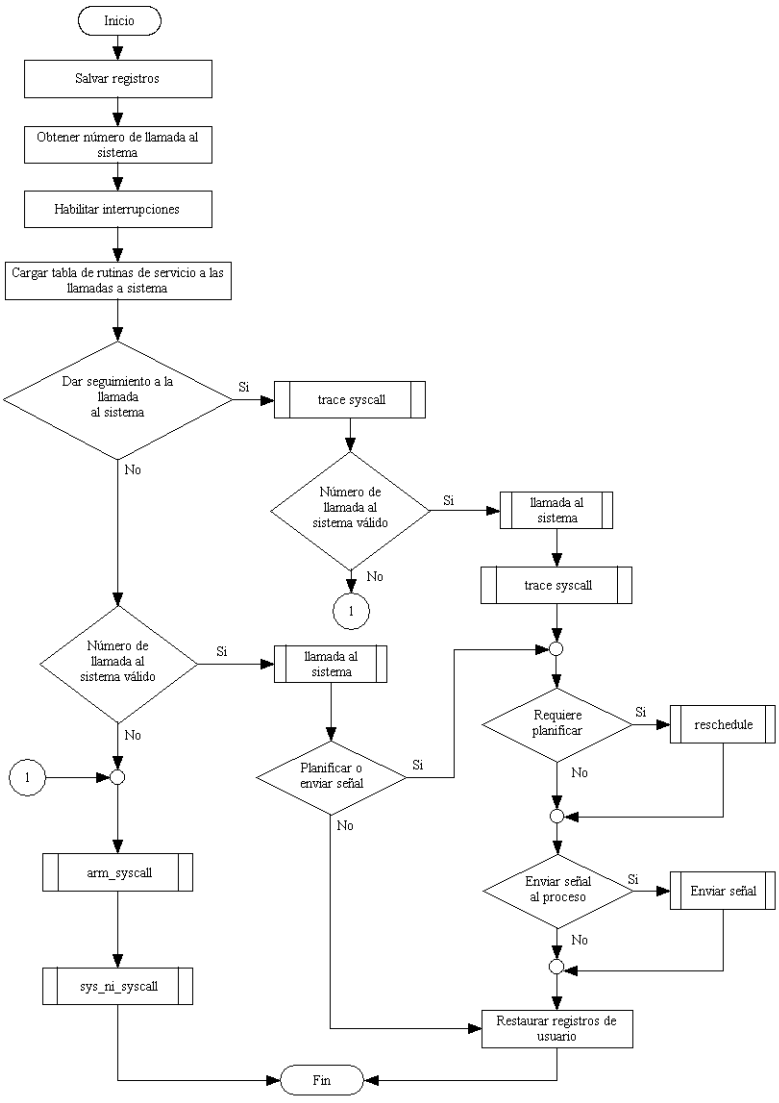

Figura 3.2: Diagrama de flujo de la función para el manejo de excepciones en Linux.

El diagrama de flujo muestra que los manejadores son funciones genéricas, cuya operación difiere en las ISRs que cada uno invoca.

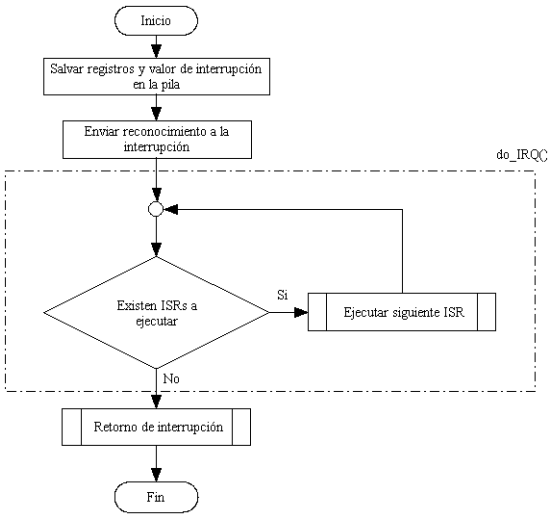

Figura 3.3: Diagrama de flujo de la función para el manejo de interrupciones en Linux.

3.4. Capa de abstracción de hardware

La capa de abstracción de hardware (*HAL-Hardware Abstraction Layer*) es la encargada de permitir la ejecución de más de un kernel en la computadora al mismo tiempo, realiza las siguientes tres funciones [17]:

- Encapsula apuntadores a funciones del kernel de Linux en una estructura de datos, llamada *rthal*, para permitir un fácil manejo de todas las funcionalidades que son importantes para las aplicaciones de tiempo real, principalmente las relacionadas al hardware, de forma que puedan ser intercambiadas por las funciones proporcionadas por RTAI cuando el soporte a tiempo real se requiera.

- Incluye funciones que permiten la interacción con el hardware (de la misma forma en que lo hace Linux) y hace que los apuntadores en la estructura *rthal* apunten a ellas.

- En el kernel de Linux, sustituye las llamadas a función originales con llamadas a los apuntadores a función de la estructura *rthal*.

Las funciones incluidas en la capa de abstracción de hardware son aquellas que permiten habilitar y deshabilitar interrupciones y las que se encargan del

procesamiento de éstas, tanto excepciones como interrupciones. Al introducir la HAL, el kernel de Linux no es afectado drásticamente, sólo existe una ligera pérdida de desempeño debido al llamado de las funciones para habilitar y deshabilitar interrupciones de la HAL, en lugar de las funciones de hardware existentes [17].

3.5. Cambios realizados en el kernel de Linux

Para la implantación de la capa de abstracción de hardware en el kernel de Linux de la PDA, se realizaron los siguientes cambios:

- Se agregó la estructura *rthal*.
- Se sustituyó la aparición de las macros para habilitar y deshabilitar interrupciones por llamadas a los apuntadores de función incluidos en la estructura *rthal*.
- Se modificaron las funciones para el procesamiento de interrupciones para que dicho procesamiento sea realizado por las funciones referenciadas en la HAL.

3.5.1. La estructura *rthal*

La figura 3.4 muestra los campos incluidos en la estructura *rthal*.

```
struct rt_hal {
      void (*do_IRQ)(int, struct pt_regs*);
      long long (*do_SRQ)(int, unsigned long);
      int (*do_TRAP)(int, struct pt_regs*);
      void (*disint)(void);
      void (*enint)(void);
      unsigned int (*getflags)(void);
      void (*setflags)(unsigned int);
      unsigned int (*getflags_and_cli)(void);
      void (*fdisint)(void);
      void (*fenint)(void);
      volatile u32 timer_match;
      void (*copy_back)(unsigned long, int);
      void (*c_do_IRQ)(int, struct pt_regs*);
};
```

Figura 3.4: Estructura de datos *rthal*.

3.5.2. Funciones para habilitar y deshabilitar interrupciones

El conjunto de funciones empleadas para habilitar y deshabilitar interrupciones, también incluye aquellas que permiten salvar y restaurar las banderas del registro de control del procesador. Cada función sólo invoca a la macro ya existente en el kernel encargada de realizar la tarea deseada. La idea de reemplazar las macros por las funciones, es poder intercambiarlas de manera dinámica con las funciones existentes en RTAI. Como ejemplo, la figura 3.5 muestra las funciones para habilitar y deshabilitar las interrupciones rápidas en el registro de control.

```
static void linux_fcli(void)
{
      hard_clf();
}

static void linux_fsti(void)
{
      hard_stf();
}
```

Figura 3.5: Nuevas funciones para habilitar y deshabilitar interrupciones rápidas.

3.5.3. Cambios en el manejo de excepciones

Los cambios realizados al manejo de excepciones en el kernel de Linux permiten que el módulo de RTAI incluya llamadas al sistema propias. En el diagrama de flujo de la figura 3.6 se observa que si el número de llamada al sistema corresponde a una función de Linux, éste se encarga de procesarla, sin embargo, cuando el número de llamada al sistema es igual al asignado a RTAI, el procesamiento es realizado por dicho módulo.

3.5.4. Cambios en el manejo de interrupciones

El fin de realizar cambios en el manejo de interrupciones es permitir que estás sean procesadas por Linux cuando no existan procesos de tiempo real, o de lo contrario, por un manejador (incluido en RTAI) que se encargue de enviar las señales de interrupción a los procesos de tiempo real. Los cambios en el kernel, consisten en invocar a la función referenciada por los apuntadores en la estructura *rthal*, en lugar de invocar directamente a las funciones

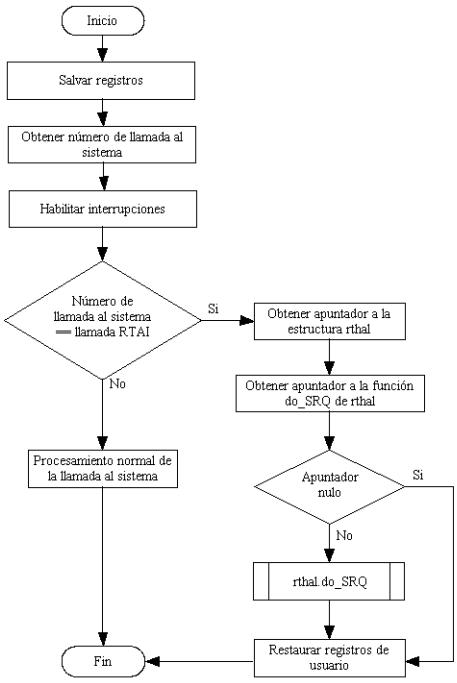

Figura 3.6: Diagrama de flujo de la nueva función para el manejo de excepciones en Linux.

incluidas en el kernel de Linux. La figura 3.7 muestra el nuevo diagrama de flujo de los manejadores.

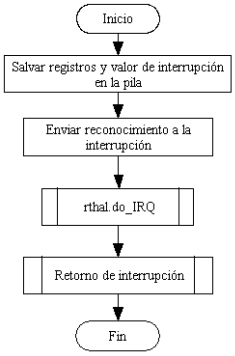

Figura 3.7: Diagrama de flujo de la nueva función para el manejo de interrupciones en Linux.

Inicialmente los apuntadores de la estructura de datos *rthal* hacen refe-

rencia a funciones con un comportamiento identico a las funciones del kernel de Linux, después al agregar el módulo de RTAI, estos apuntadores son modificados para hacer referencia a las funciones de RTAI.

Una vez implementada la capa de abstracción de hardware en la PDA, es posible hacer uso de las funciones incluidas en el módulo de RTAI, y así poder ejecutar de manera simultanea a el kernel de tiempo real desarrollado y a Linux en la PDA.

Capítulo 4

Diseño del kernel de tiempo real

Una vez implantada la capa de abstracción de hardware de RTAI en el kernel de Linux para la PDA, discutido en el capítulo tres, es posible la ejecución concurrente de nuestro kernel de tiempo real y de Linux en ella. Para la implementación del kernel de tiempo real existen diversas metodologías, por ejemplo:

- Estructura de capas.
- Microkernel.
- Estructura monolítica.

La alternativa seguida en nuestro diseño es una estructura monolítica, en la que el sistema operativo es un solo programa encargado de ofrecer los servicios necesarios para la ejecución adecuada de los programas del usuario. Existen varios manejadores que interaccionan de manera muy estrecha, pero cada uno ofrece servicios específicos, y puede hacer uso de funciones de otros para efectuar sus tareas. Mediante este esquema no se requiere una definición rigurosa de los módulos del sistema operativo, basta con agrupar a las funciones asociadas a un determinado módulo en un manejador. Se tienen funciones accesibles a los procesos de usuario conocidas como *llamadas al sistema*, a través de las cuales se da la interacción con el kernel.

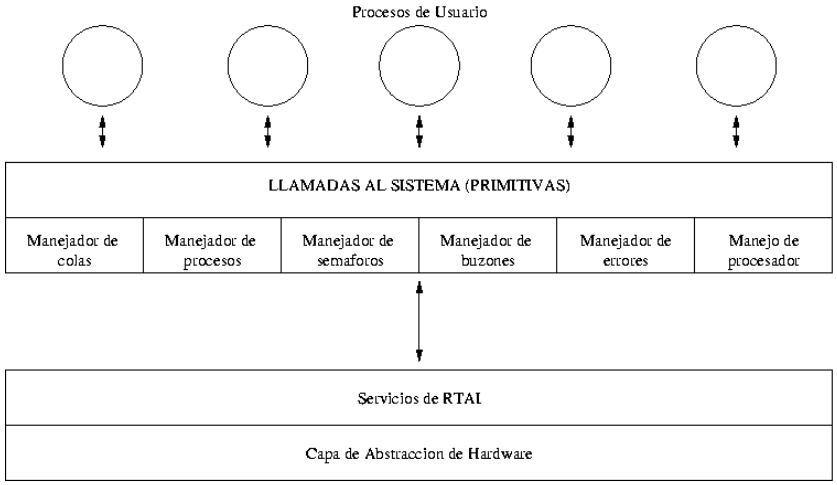

Figura 4.1: Diagrama a bloques de la arquitectura del kernel de tiempo real.

4.1. Arquitectura general

La figura 4.1 presenta un diagrama de la arquitectura del kernel de tiempo real. En el nivel inferior de la arquitectura se encuentra la capa de abstracción de hardware y las rutinas de servicio de RTAI. El siguiente nivel, los manejadores del kernel, hacen uso de RTAI para interaccionar con el hardware, es en este nivel donde se encuentran las funciones encargadas de la manipulación directa de las estructuras de datos del kernel. En el tercer nivel se puede observar la interfaz ofrecida a los procesos de usuario, mediante las distintas llamadas al sistema disponibles. Finalmente en la parte superior de la arquitectura se encuentran los procesos de usuario que se ejecutan de manera concurrente.

4.2. Procesos (tareas)

La unidad básica de trabajo en el kernel es el proceso (o tarea), inicialmente el kernel está configurado con un número máximo de 25 procesos, sin embargo, al compilar el kernel se puede modificar este número dependiendo de los requerimientos.
Los procesos pueden ser definidos de manera estática, definidos durante la inicialización del sistema, o dinámica. Para agregar un proceso de manera dinámica al sistema, es necesario incluirlo como un módulo del kernel de

Linux en el que se invocan funciones para la creación de tareas de tiempo real.

En el sistema existe un proceso especial, al que llamaremos el proceso LINUX, éste corresponde a la ejecución de Linux. Es manejado de manera similar a los procesos de tiempo real. Dentro del kernel se reserva un bloque de control de proceso para él, y siempre se encuentra listo para ser ejecutado. Sin embargo, a pesar de lo anterior, para facilitar su manejo, nunca es insertado en alguna cola del kernel, y el control del procesador se le asigna sólo si no existe alguna tarea de tiempo real lista para ejecutarse, es decir, su prioridad es la más baja.

4.3. Bloque de control del proceso (BCP)

Dentro del sistema se requiere un mecanismo para mantener el control de la ejecución de los procesos. El Bloque de Control de Proceso es una estructura asociada a cada proceso, que mantiene información referente al estado del proceso y los recursos que está empleando. La figura 4.2 muestra la información contenida en el BCP.

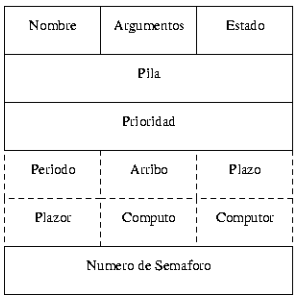

Figura 4.2: Bloque de Control de Proceso en el kernel.

La información contenida en al BCP, se puede clasificar en tres tipos:

- **Información del proceso:** Incluye el nombre del proceso, argumentos que recibe, así como su estado actual y la pila del proceso. La pila del proceso es utilizada para el almacenamiento de variables locales durante la ejecución del proceso, pero también es utilizada para almacenar el contexto del mismo al momento de ser desalojado del procesador. Es en ella donde se copia el contenido de los registros para ser recuperado cuando al proceso se le asigna tiempo de cómputo nuevamente.

- **Información para el planificador:** Corresponde a campos que permiten al planificador tomar decisiones para colocar o sacar de ejecución al proceso, como la prioridad, el tiempo de cómputo, período. Es importante notar que al momento de compilar el kernel, el usuario define el tipo de planificación a utilizar, y dependiendo de la elección se incluyen o no algunos campos. En el diagrama se muestran entre marcos punteados los campos que sólo son incluidos en caso de elegir una planificación de tipo RM.
- **Información de recursos empleados por el proceso:** En este caso, se mantiene un campo que contiene el identificador de semáforo que el proceso está utilizando.

4.4. Estados de los procesos en el kernel

La figura 4.3 muestra un diagrama que ilustra los posibles estados de los procesos en el sistema, así como las posibles transiciones entre ellos.

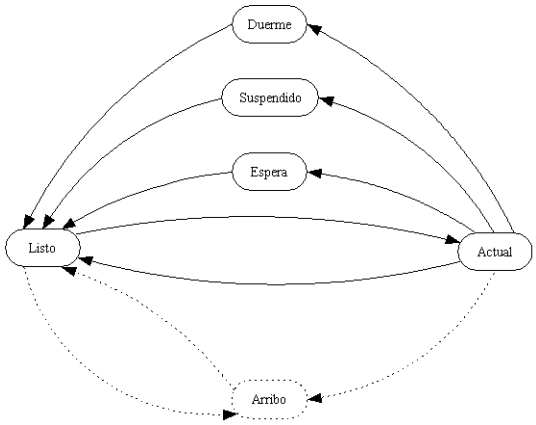

Figura 4.3: Diagrama de estados de los procesos en el kernel.

El significado de cada estado es presentado enseguida.

- **Listo:** Indica que el proceso se encuentra en espera de que el procesador le sea asignado para iniciar o continuar su ejecución.
- **Actual:** En el sistema sólo hay un proceso en este estado, y corresponde a aquel que está haciendo uso del procesador.

- **Espera:** Hace referencia a un proceso esperando por un semáforo. Puede haber varios procesos en espera de un mismo semáforo.
- **Duerme:** Un proceso es llevado a esta condición al solicitar un retraso en su ejecución.
- **Suspendido:** El proceso trató de leer datos de un buzón vacío, o escribir a uno sin espacio libre.
- **Arribo:** Estado empleado solamente en planificaciones de tipo RM, el proceso se encuentra en una cola esperando el instante de tiempo en que será activado.

4.5. Transiciones entre estados

Las razones para que se de una transición entre estados son las siguientes:

- **De Listo a Actual:** El proceso en ejecución libera el procesador, de manera voluntaria o forzada, y el primer proceso en la cola de listos pasa a tomar el control de él, cambiando su estado a Actual.
- **De Actual a Listo:** El proceso en ejecución pasa a estado de Listo si el planificador decide que el tiempo de cómputo disponible para él se agotó, o bien, un proceso de más prioridad arribó al sistema (depende del planificador usado).
- **De Actual a Espera:** Esta transición se da si el proceso trata de acceder a un semáforo empleado por otro.
- **De Actual a Duerme:** El proceso solicita al sistema que su ejecución sea retardada por un determinado lapso de tiempo.
- **De Actual a Suspendido:** Al realizar una operación de lectura en un buzón vacío o de escritura en un buzón sin espacio libre.
- **De Espera a Listo:** Una vez que el semáforo por el que el proceso se encontraba esperando fue liberado y tiene el control de él.
- **De Duerme a Listo:** Ha transcurrido el lapso de tiempo que el proceso solicitó ser retrasado.

- **De Suspendido a Listo:** Algún proceso escribió datos al buzón que estaba vacío, o bien se leyeron datos del buzón y existe espacio disponible para escribir en él.
- **De Actual a Arribo:** El proceso ha concluido su operación y tiene que esperar el nuevo instante al que será activado.
- **De Arribo a Listo:** Este cambio de estado ocurre cuando el instante de activación del proceso ha llegado.
- **De Listo a Arribo:** El sistema verifica la perdida de plazos en los procesos listos, y los inserta en la cola de arribos si su plazo se perdió para esperar su nuevo instante de activación.

Las tres últimas sólo en caso de usar planificación de tipo RM.

4.6. Manejadores del kernel y llamadas al sistema

Existen distintos puntos de vista para analizar la arquitectura un sistema operativo. Por ejemplo, examinando los servicios que ofrece, los módulos que lo componen, o bien estudiando la interfaz a los programas de usuario con que cuenta [6].A continuación describimos los distintos componentes del kernel de tiempo real (manejadores) y la interfaz con los programas de usuario de cada uno de ellos (llamadas al sistema).

4.6.1. Manejador de colas

El manejador de colas cuenta con las estructuras de datos y funciones necesarias para la manipulación de listas doblemente ligadas, que se comportan como colas de prioridad (fig. 4.4). Lo integran funciones genéricas que los manejadores emplean de diferentes modos a fin de lograr comportamientos específicos, como es el caso de la cola de arribos empleada en el planificador RM, y ha sido desarrollado bajo la premisa de que un proceso se encuentra en sólo una cola a cada momento.

El componente básico en el manejador es un arreglo cuya estructura se presenta en la figura 4.5. Cada entrada del arreglo representa el nodo de una

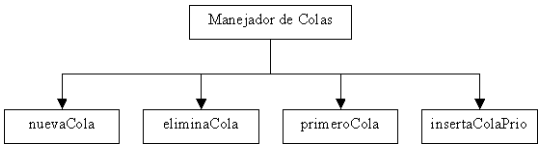

Figura 4.4: Funciones que integran al manejador de colas.

cola, cuenta con campos para mantener información de los nodos siguiente y previo en la cola, y los campos llave y tiempo. La llave es empleada para ordenar los nodos dentro de la cola, puede tratarse de la prioridad, del período o del tiempo de arribo del proceso, dependiendo del manejador que esté haciendo uso de la cola.

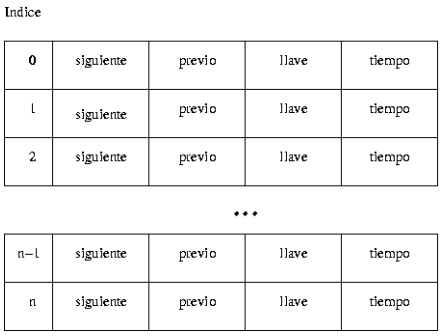

Figura 4.5: Estructura del arreglo empleado en el manejo de colas.

Una cola tiene dos nodos especiales, uno indica su inicio y el otro marca el fin de ella. La figura 4.5 muestra que el arreglo se divide en dos secciones, la primera corresponde a los nodos que servirán para mantener el rastro de los procesos en las colas, y la segunda contiene los nodos de inicio y fin de éstas. Debido a que un proceso está presente únicamente en una cola a un instante dado, se requiere sólo una entrada en el arreglo para hacer referencia a él; por lo tanto, el tamaño del arreglo corresponde al número máximo de procesos en el sistema más el número de nodos de cabecera (inicio y fin) [1] .

El manejador de colas no tiene interfaz hacia los programas de usuario, sus funciones son empleadas de manera interna al kernel. Se tienen las siguientes funciones:

[1]Como se explica más adelante, en el sistema se requiere una cola por cada semáforo, una por cada buzón, la cola de procesos listos, la cola de procesos durmiendo, y en el caso de RM la cola de arribos de procesos.

nuevaCola
PROTOTIPO: `int nuevaCola(void).`
FUNCIÓN: Reserva los nodo inicial y final para una nueva cola (fig. 4.6).
ENTRADA: Ninguna.
SALIDA: El índice del arreglo correspondiente al nodo inicial de la nueva cola, o error si no se pudo crear.

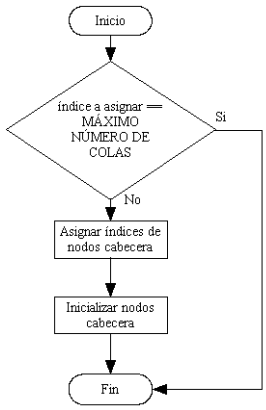

Figura 4.6: Diagrama de flujo de la función para crear una nueva cola.

eliminaCola
PROTOTIPO: `int eliminaCola(int numProc).`
FUNCIÓN: Elimina el proceso indicado de la cola en la que se encuentre.
ENTRADA: El número del proceso (que corresponde con su posición en el arreglo) a eliminar (fig. 4.7).
SALIDA: El número del proceso eliminado.

primeroCola
PROTOTIPO: `int primeroCola(int cabeza).`
FUNCIÓN: Elimina al primer proceso de una cola dada (fig. 4.8).
ENTRADA: El valor del nodo inicial de la cola.
SALIDA: Número del proceso eliminado, o error si la cola está vacía.

insertaColaPrio
PROTOTIPO: `int insertaColaPrio(int proceso, int cabeza, int prioridad).`

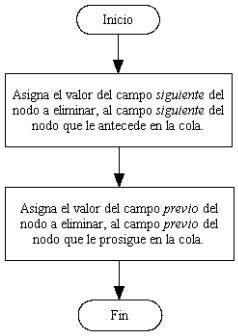

Figura 4.7: Diagrama de flujo de la función para eliminar un proceso de una cola.

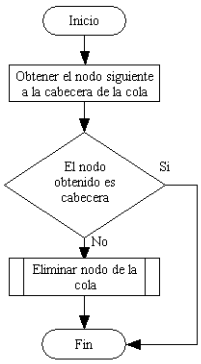

Figura 4.8: Diagrama de flujo de la función para eliminar al primer proceso de una cola.

FUNCIÓN: Inserta el proceso indicado en una cola, en la que los procesos están ordenados por su prioridad (fig 4.9).
ENTRADA: Número del proceso a insertar, valor del nodo inicial de la cola y la prioridad del proceso.
SALIDA: La constante OK indicando que la inserción fue hecha.

4.6.2. Manejador de procesos

Las estructuras de datos y funciones que integran el manejador de procesos, permiten la administración de los procesos en el kernel, a través de la primitivas disponibles en él, se puede crear, retrasar su ejecución o eliminar procesos. La figura 4.10 presenta las funciones que componen a este maneja-

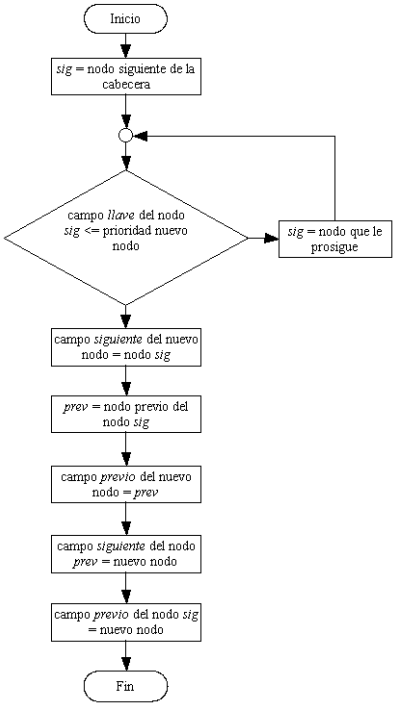

Figura 4.9: Diagrama de flujo de la función para insertar, dependiendo de su prioridad, a un proceso en una cola.

dor.

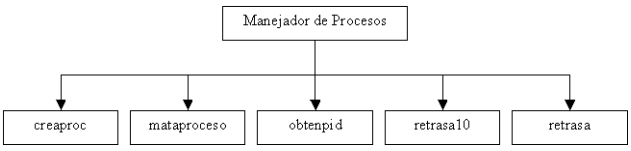

Figura 4.10: Primitivas disponibles en el manejador de procesos.

La estructura más importante en el manejador de procesos es el Bloque de Control de Proceso, mostrado en la figura 4.2, y las primitivas encargadas de su manipulación se detallan enseguida.

creaproc
PROTOTIPO1: `PRIMITIVA creaproc(void (*proceso)(int),int periodo,`

`int computo, int plazo, char *nombre, int numArgs, int args).`
PROTOTIPO2: `PRIMITIVA creaproc(void (*proceso)(int), int prioridad, char* nombre, int numArgs, int args).`
FUNCIÓN: Crear un nuevo proceso en el sistema e insertarlo en la cola de listos (fig. 4.11).
ENTRADA: Un apuntador a la función correspondiente al proceso, el nombre del proceso, el número de argumentos que recibe, y los argumentos. Dependiendo del planificador empleado también recibe la prioridad del proceso o el período, tiempo de cómputo y su plazo.
SALIDA: El identificador asignado al nuevo proceso o error si no se creo.

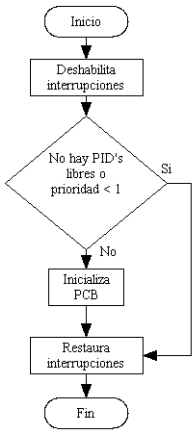

Figura 4.11: Diagrama de flujo de la primitiva para la creación de procesos.

mataproceso
PROTOTIPO: `PRIMITIVA mataproceso(int pid).`
FUNCIÓN: Eliminar del sistema el proceso indicado (fig. 4.12).
ENTRADA: El identificador del proceso a eliminar.
SALIDA: La constante OK indicando la finalización correcta de la función, o un error si no se pudo eliminar el proceso.

obtenpid
PROTOTIPO: `PRIMITIVA obtenpid(void).`
FUNCIÓN: Devolver el identificador del proceso en ejecución.
ENTRADA: Ninguna.
SALIDA: El identificador del proceso en ejecución.

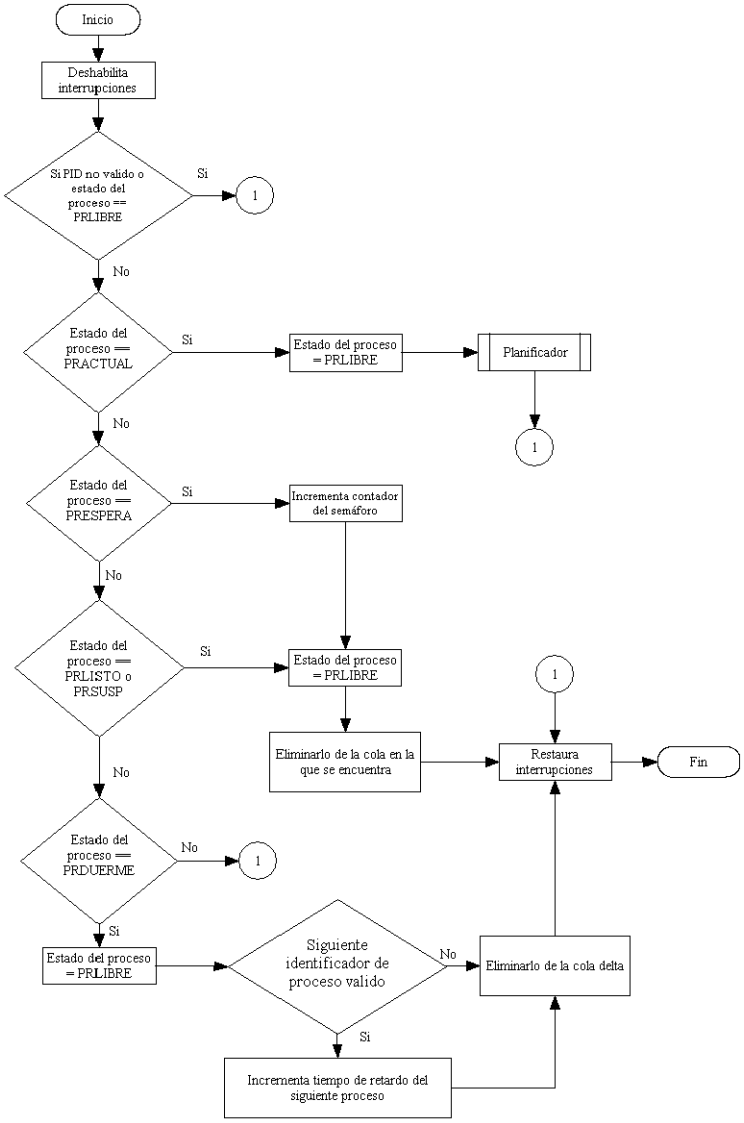

Figura 4.12: Diagrama de flujo de la primitiva para la eliminación de procesos.

retrasa10
PROTOTIPO: `PRIMITIVA retrasa10(int n).`
FUNCIÓN: Retrasar la ejecución del proceso actual por n lapsos de 10 milisegundos (fig. 4.13).

ENTRADA: El número de lapsos de 10 milisegundos a retrasar el proceso.
SALIDA: La constante OK indicando la finalización del retardo.

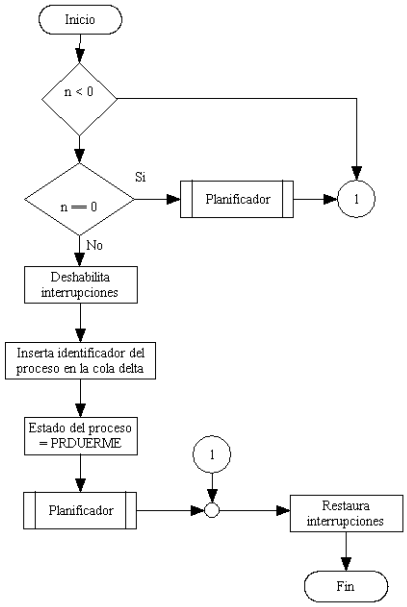

Figura 4.13: Diagrama de flujo de la primitiva para retrasar procesos en 10 décimas de segundo.

retrasa
PROTOTIPO: `PRIMITIVA retrasa(int n).`
FUNCIÓN: Retrasar la ejecución del proceso actual por n segundos (fig. 4.14).
ENTRADA: La cantidad n de segundos a retrasar el proceso.
SALIDA: La constante OK indicando el final del lapso de retraso.

Existe una función dentro del manejador que no se encuentra disponible al usuario, pero es de gran importancia, pues se encarga de reanudar la ejecución de los procesos que solicitaron ser retardados.

despierta
PROTOTIPO: `PROCINT despierta(void).`
FUNCIÓN: Insertar en la cola de listos los procesos que solicitaron ser retrasados (fig. 4.15).

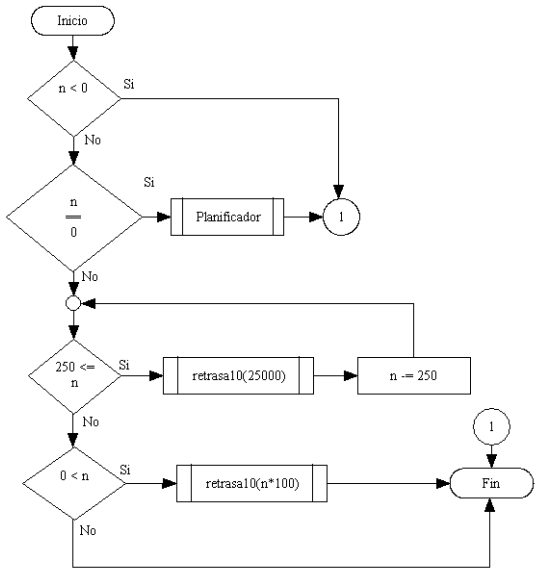

Figura 4.14: Diagrama de flujo de la primitiva para retrasar procesos n segundos.

ENTRADA: Ninguna.
SALIDA: La constante OK para indicar su fin.

4.6.3. Manejador de semáforos

El manejador de semáforos incluye estructuras de datos y funciones que las manipulan, requeridas para que los procesos puedan tener acceso a mecanismos de sincronización. En la figura 4.16 se muestran los componentes de la estructura usada para representar un semáforo, se incluye una variable para indicar el estado, un contador y dos variables para almacenar el inicio y el fin de una cola, empleada para mantener a los procesos que se encuentren bloqueados en espera del semáforo.

La figura 4.17 muestra las primitivas implantadas dentro del manejador de semáforos, y a continuación se describe cada una de ellas.

creasem
PROTOTIPO: `PRIMITIVA creasem(int contador).`

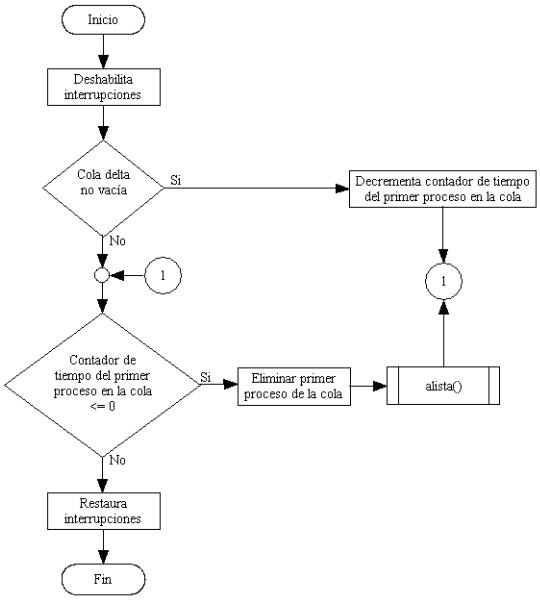

Figura 4.15: Diagrama de flujo de la función encargada de reanudar los procesos dormidos.

Figura 4.16: Estructura empleada en el manejo de semáforos.

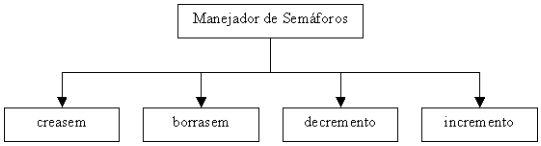

Figura 4.17: Primitivas disponibles en el manejador de semáforos.

FUNCIÓN: Inicializa los componentes de un semáforo, contador y cola (fig. 4.18).
ENTRADA: Contador inicial del semáforo.
SALIDA: Identificador del semáforo creado, o error si el contador inicial es negativo o no existen recursos para la creación del semáforo.

borrasem

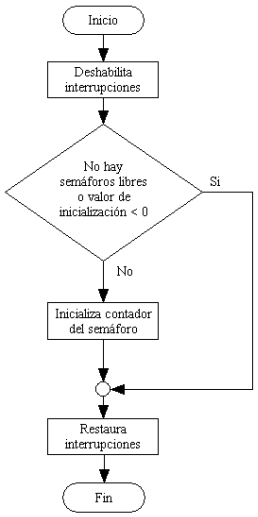

Figura 4.18: Diagrama de flujo de la primitiva para crear un semáforo.

PROTOTIPO: `PRIMITIVA borrasem(int semaforo).`
FUNCIÓN: Libera los recursos empleados por el semáforo (fig. 4.19).
ENTRADA: Identificador del semáforo a liberar.
SALIDA: La constante OK, o error si el identificador del semáforo es incorrecto.

decremento
PROTOTIPO: `PRIMITIVA decremento(int semaforo).`
FUNCIÓN: Decrementa de manera atómica el contador del semáforo (fig. 4.20).
ENTRADA: Identificador del semáforo a decrementar.
SALIDA: La constante OK, o error si el identificador del semáforo no es correcto.

incremento
PROTOTIPO: `PRIMITIVA incremento(int semaforo).`
FUNCIÓN: Incrementa de manera atómica el contador del semáforo (fig. 4.21).
ENTRADA: Identificador del semáforo a decrementar.
SALIDA: La constante OK, o error si el identificador del semáforo no es correcto.

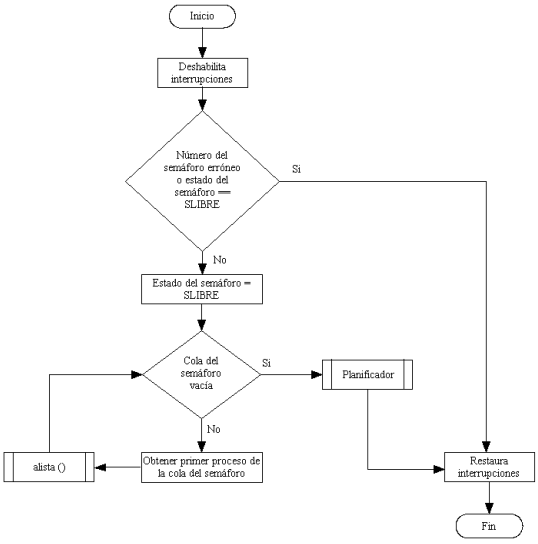

Figura 4.19: Diagrama de flujo de la primitiva para liberar un semáforo.

4.6.4. Manejador de buzones

A través del manejador de buzones se pone a disposición de los procesos un medio de comunicación asíncrona. La figura 4.22 muestra el esquema de la estructura de datos alrededor de la que se construyó el manejador. Cada buzón cuenta con un *buffer* de datos tratado como *buffer circular*. Al escribir datos, si no existe espacio en el *buffer*, el proceso se suspende en espera de espacio disponible. Al leer de un buzón vacío, el proceso también se suspende hasta que haya datos en el buzón.

En el diagrama de la figura 4.23 se observan las rutinas que conforman el manejador de buzones, las cuales se detallan enseguida.

envia
PROTOTIPO: `PRIMITIVA envia(int numbuzon, const char *datos, int bytes_enviar).`
FUNCIÓN: Escribe datos en el buzón indicado (fig. 4.24).
ENTRADA: El identificador del buzón, un apuntador al *buffer* donde se en-

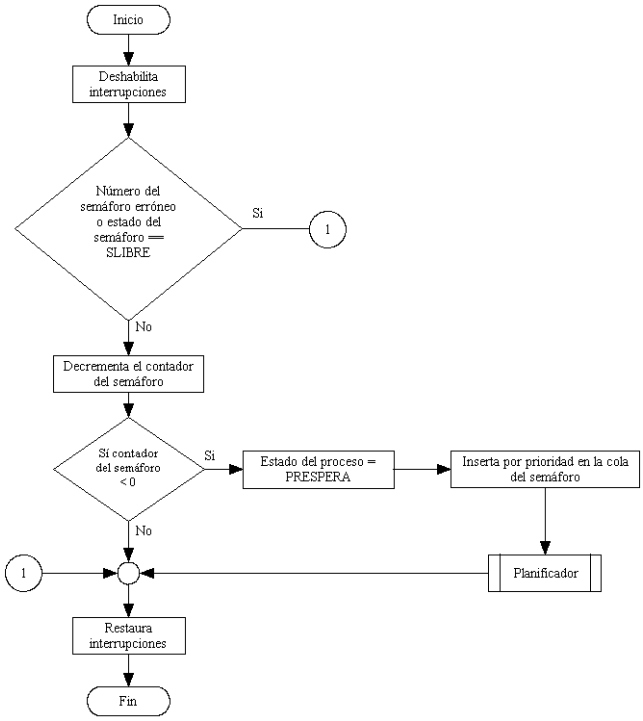

Figura 4.20: Diagrama de flujo de la primitiva para decrementar un semáforo.

cuentran los datos a copiar al buzón y el número de bytes a ser escritos.
SALIDA: Número de bytes escritos en el buzón o error si el identificador del buzón no es válido.

recibe
PROTOTIPO: `PRIMITIVA recibe(int numbuzon, char *datos, int bytes_leer).`
FUNCIÓN: Lee datos del buzón indicado (fig. 4.25).
ENTRADA: El identificador del buzón, un apuntador al *buffer* donde se desean almacenar los datos y el número de bytes a leer.
SALIDA: Número de bytes leídos del buzón o error si el identificador del buzón no es válido.

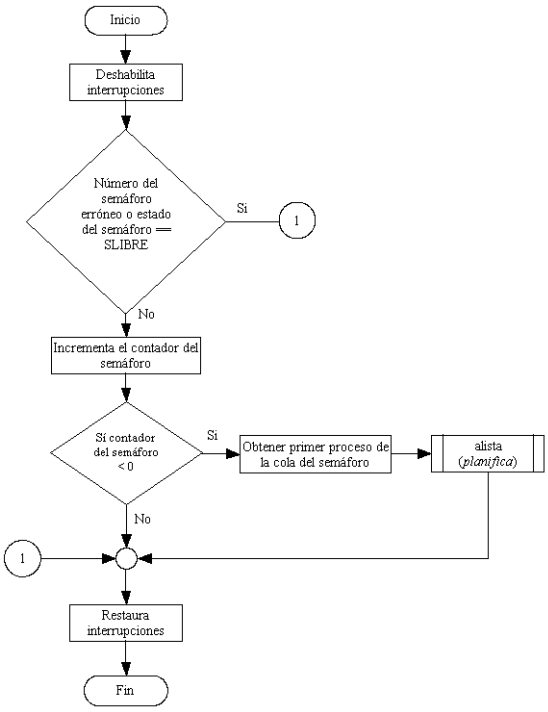

Figura 4.21: Diagrama de flujo de la primitiva para incrementar un semáforo.

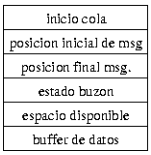

Figura 4.22: Estructura empleada en el manejo de buzones.

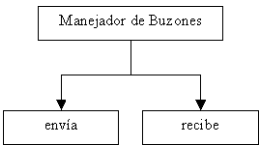

Figura 4.23: Primitivas disponibles en el manejador de buzones.

4.6.5. Manejo del procesador

La administración de recursos es una de las tareas más importantes del sistema operativo, y en un sistema de tiempo compartido el control del pro-

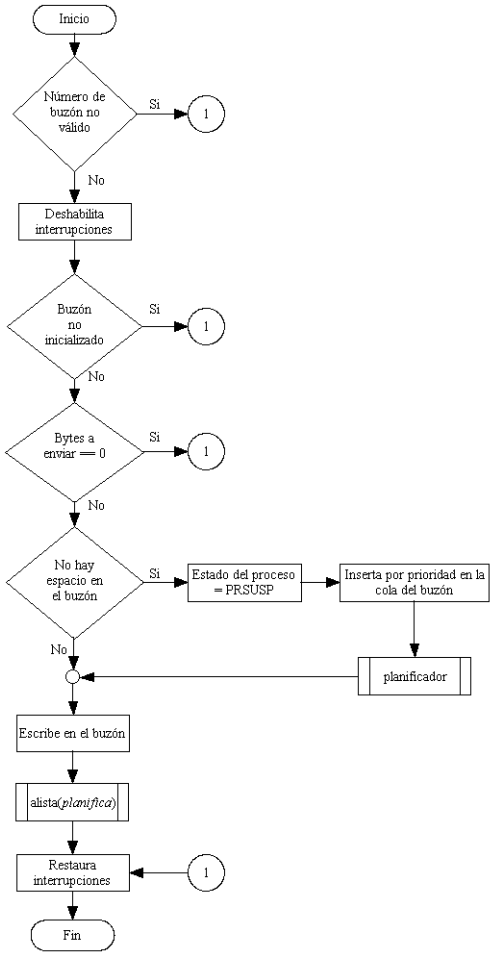

Figura 4.24: Diagrama de flujo de la primitiva para escribir un buzón.

cesador cobra gran relevancia. El diagrama de la figura 4.26 muestra las primitivas que permiten llevar a cabo el manejo del procesador en el kernel de tiempo real.

En el kernel de tiempo real, el módulo para el manejo del procesador está integrado por dos submódulos, el planificador y el despachador. El primero se encarga de elegir de entre los procesos listos, a aquel que debe de ser ejecutado. Existen diferentes algoritmos para realizar la elección, en el kernel se han implementado los algoritmos *FIFO-Round-Robin* y *Rate Monotonic*. El tipo de planificación empleada en el sistema es determinada por el usuario al momento de la compilación. El módulo de planificación ha si-

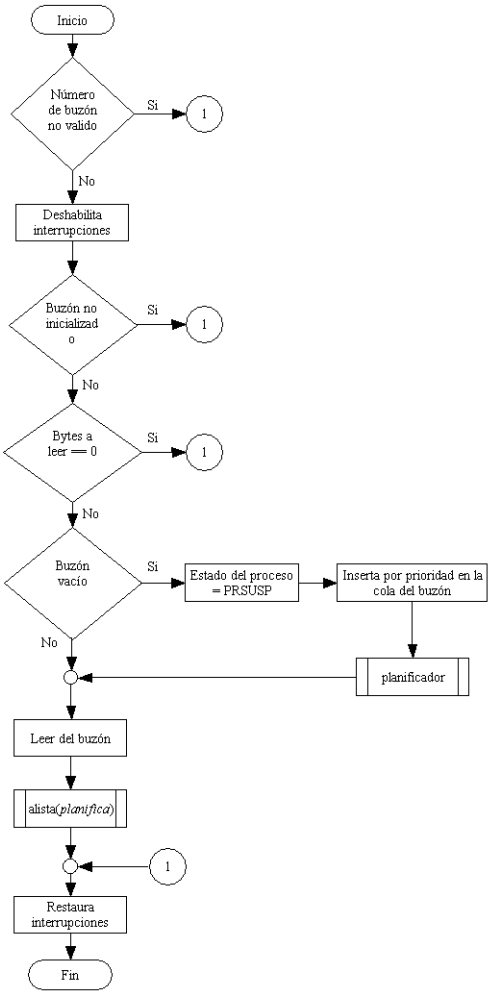

Figura 4.25: Diagrama de flujo de la primitiva para leer un buzón.

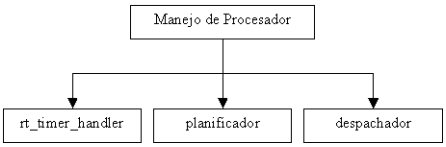

Figura 4.26: Funciones para el manejo del procesador.

do divido en dos funciones, la primera se ejecuta al momento de generarse una interrupción de reloj, verifica la existencia de tareas de tiempo real y de

ser necesario invoca las funciones de RTAI para realizar el cambio de entorno entre Linux y tiempo real, también se checa que el *quantum* de tiempo asignado al proceso no se halla consumido (fig. 4.27). La segunda función integrante del planificador se encarga de obtener un proceso de la cola de listos e invocar al despachador para que inicie o reanude su ejecución (fig. 4.28).

rt_timer_handler
PROTOTIPO: `void rt_timer_handler(void).`
FUNCIÓN: Procesa las interrupciones de reloj, de ser necesario realiza el cambio de entorno entre Linux y tiempo real (fig. 4.27).
ENTRADA: Ninguna.
SALIDA: Ninguna.

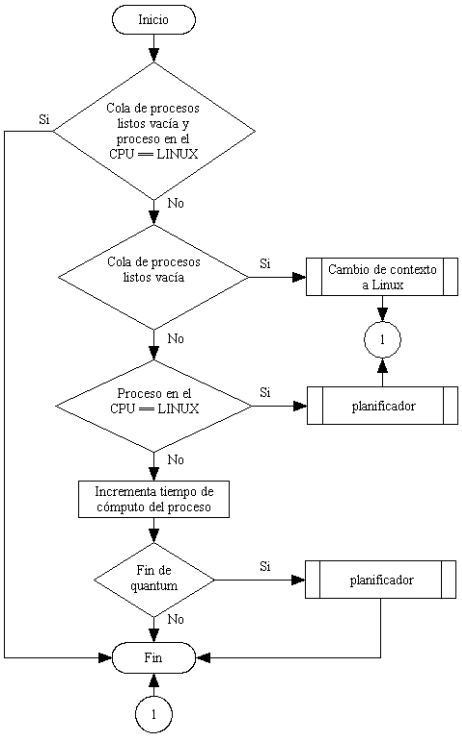

Figura 4.27: Diagrama de flujo de la función para el manejo de la interrupción de reloj.

Una vez que el planificador ha elegido el proceso a ejecutar, el despacha-

dor es el encargado de realizar el cambio de contexto. El cambio de contexto se refiere a salvar el valor actual de los registros en la pila del proceso en ejecución, y asignar a ellos el valor contenido en la pila del nuevo proceso (fig. 4.29).

planificador
PROTOTIPO: `void planificador(void).`
FUNCIÓN: Elige al siguiente proceso en la cola de listos para ejecutarlo (fig. 4.28).
ENTRADA: Ninguna.
SALIDA: Ninguna.

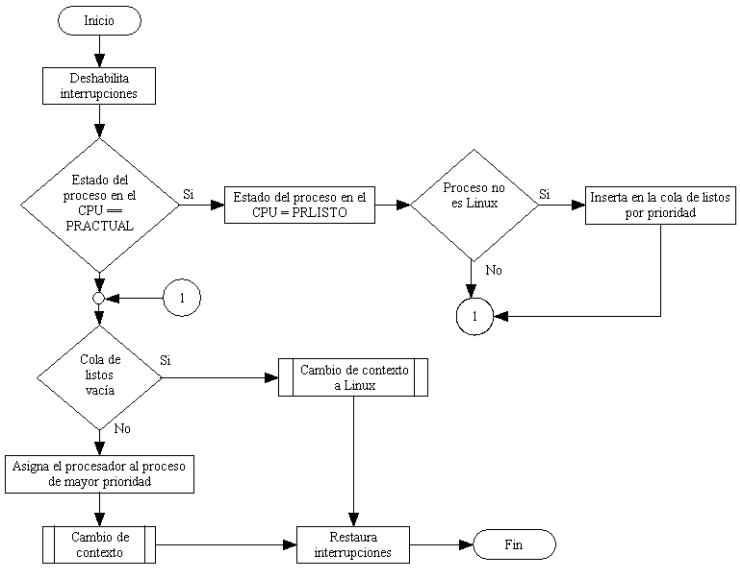

Figura 4.28: Diagrama de flujo del planificador.

despachador
PROTOTIPO: `void despachador(void *nproc, void *vproc, void *lproc).`
FUNCIÓN: Realiza el cambio de contexto entre dos procesos (fig. 4.29).
ENTRADA: Un apuntador al bloque de control de proceso de la tarea en ejecución, la nueva tarea y al descriptor de Linux.
SALIDA: Ninguna.

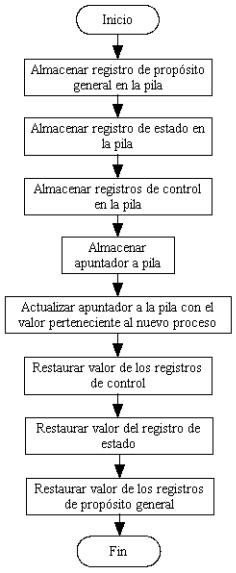

Figura 4.29: Diagrama de flujo del despachador.

4.6.6. Manejo de errores

En cada una de las funciones del kernel, antes de iniciar la manipulación de estructuras de datos es necesario realizar una validación de los parámetros recibidos. De esta forma, se evita un comportamiento anómalo de la función. El kernel cuenta con manejador de errores que consta de una serie de números de error empleados para indicar posibles situaciones no esperadas en el kernel, y una macro que permite visualizar un mensaje relacionado al número de error obtenido.

msgerror
DEFINICIÓN EN LÍNEA:`#define msgerror(errno).`
FUNCIÓN: Devolver el mensaje de error asociado a un número de error.
ENTRADA: El número de error a decodificar.
SALIDA: El mensaje de error asociado al número recibido.

Los posibles códigos de error son:

BZNOVAL	(-1)	:	Número de buzón inválido.
BZNOINI	(-2)	:	Buzón no inicializado.
CONOLIB	(-3)	:	No es posible crear una cola más en el sistema.
PRPNOVA	(-4)	:	No es posible crear proceso o prioridad del proceso errónea.
PRPIDNL	(-5)	:	No es posible crear proceso.
PRPIDNV	(-6)	:	Número de proceso erróneo.
SEMNOVA	(-7)	:	Número erróneo de semáforo o semáforo libre.
SEMPNVA	(-8)	:	No es posible crear semáforo o contador de semáforo.
SEMNOLI	(-9)	:	No es posible crear semáforo.
TIPNOVA	(-10)	:	Tiempo de retraso inválido.

4.7. Configuración e inicialización del kernel

4.7.1. Configuración del kernel

A través de la modificación de algunos parámetros, el usuario puede establecer los recursos disponibles en el kernel, así como, definir aspectos relativos a la ejecución del mismo. Las diferentes variables que el usuario puede modificar se presentan el la tabla 4.1.

Variable	Archivo	Parámetro afectado
TICK_TIME	mancpu.h	Establece el tiempo para generación de las interrupciones de reloj.
QUANTUM	mancpu.h	Número de ticks de reloj asignados a la ejecución de un proceso en el planificador Round-Robin.
NSEM	mansem.h	Número de semáforos disponibles.
NBUZONES	manbuzon.h	Número de buzones existentes en el kernel.
TAMBUZON	manbuzon.h	Tamaño, en bytes, de los buzones.
NUMPROC	manproc.h	Número máximo de procesos en el kernel.
MAXPRIO	rkernel.h	Prioridad máxima a ser asignada a un proceso.
MINPRIO	rkernel.h	Prioridad mínima a ser asignada a un proceso.

Tabla 4.1: Variables que permiten modificar el comportamiento del kernel.

Es necesario recordar que, el kernel asigna las prioridades en orden inverso, entre menor sea el número asignado a la prioridad del proceso, la prioridad dentro del kernel es mayor.

Por defecto, el kernel hace uso de los valores mostrados en la tabla 4.2.

Variable	Valor
TICK_TIME	10ms
QUANTUM	20
NSEM	25
NBUZONES	5
TAMBUZON	512
NUMPROC	25
MAXPRIO	20
MINPRIO	1

Tabla 4.2: Valores asignados a las variables de configuración del kernel.

4.7.2. Inicialización del kernel

La inicialización del kernel de tiempo real se realiza en la función *init()* del módulo, que se encuentra en el archivo *zinux.c*, y se puede dividir en tres etapas:

- Inicializar recursos del kernel de tiempo real.
- Crear de procesos de tiempo real.
- Inicializar el módulo RTAI.

En la primera etapa se reservan los recursos necesarios para el funcionamiento adecuado del kernel de tiempo real, incluye: inicializar las tablas de procesos, buzones y semáforos. También se crean las colas de procesos listos, dormidos, y en el caso de *Rate Monotonic*, la cola de procesos en espera de su arribo.
En la segunda etapa de la inicialización, se puede realizar la creación de procesos de tiempo real del usuario. Es importante recordar que, el usuario puede agragar más procesos al sistema de manera dinámica, a través de módulos del kernel de Linux.
Finalemente, cuando el se han reservado todos los recursos necesarios para la operación del kernel y se cuenta con procesos de tiempo real, es necesario iniciar los servicios de RTAI, que serán los encargados de realizar la captura de las interrupciones en el sistema, y pasarlas a Linux o a los procesos de tiempo real.

Capítulo 5

Resultados

En este capítulo se muestran los resultados obtenidos en la implantación del kernel de tiempo real. El principal objetivo de las pruebas es, verificar que la planificación de las tareas de tiempo real, sea la esperada bajo las diferentes circunstancias que se pueden presentar al ser ejecutadas en el kernel.

5.1. Entorno de pruebas

El sistema ha sido desarrollado empleando la iPaq H3950, la cual cuenta con un procesador *Intel XScale a 400 MHz*, y en ella se ejecuta el sistema operativo *Linux Familiar GPE version 7.0*, desarrollado con base en el kernel de Linux *linux-2.4.19*. Para poder hacer uso del kernel de tiempo real en la PDA es necesario llevar a cabo los siguientes pasos:

1. Compilar el kernel de Linux para agregar el soporte a RTAI e instalar la nueva imagen en la PDA.
2. Compilar RTAI e instalar el módulo obtenido en la PDA.
3. Compilar el kernel de tiempo real e instalarlo en la PDA.

5.1.1. Compilación del kernel de Linux con soporte a RTAI

Para poder emplear el módulo de RTAI, y así ejecutar el kernel de Linux concurrentemente con el kernel de tiempo real, se requiere compilar el kernel

de Linux, al que se le tienen que aplicar los siguientes *parches* [1] (en orden):

1. patch-2.4.19-rmk6.gz
2. diff-2.4.19-rmk6-pxa1.gz
3. patch-2.4.19-rmk6-pxa1-hh13.gz
4. parche-zinux.gz

Los tres primeros parches agregan al kernel original de Linux, los cambios necesarios para su funcionamiento adecuado en la iPaq H3950, y el parche *parche-zinux.gz* contiene los cambios planteados en el capítulo 3, para incluir la capa de abstracción de hardware en el kernel de Linux.
Siendo el directorio del código fuente de Linux el directorio actual de trabajo, se puede aplicar un parche al kernel de Linux mediante el siguiente comando:

zcat nombre_parche |patch -p1

Después de aplicar los parches indicados, haciendo uso del *crosscompiler* adecuado, se puede compilar el kernel de Linux mediante los siguientes comandos [18]:

- ***make h3900_config***. Copia a los directorios adecuados los archivos de configuración específicos de la arquitectura de la PDA.
- ***make oldconfig***. Establece los parámetros de compilación para el kernel. En este paso se indica que se desea agregar el soporte de RTAI.
- ***make dep***. Resuelve las dependencias existentes entre los módulos del kernel.
- ***make zImage***. Genera la imagen del kernel.

Una vez obtenida la nueva imagen del kernel de Linux, está se debe copiar al directorio */boot* de la iPaq H3950. Para que al ser reiniciada, se cuente con el soporte a RTAI.

[1] Un parche es un archivo que contiene los cambios en uno o varios archivos de código fuente de alguna aplicación, requeridos para extender su funcionalidad.

5.1.2. Compilación e instalación de RTAI

La versión de RTAI empleada en el desarrollo de la tesis fue *rtai-3.0*, cuyos archivos fuente se pueden obtener en *www..rtai.org*. Para compilar el módulo de RTAI para la iPaq, se debe de ejecutar el siguiente comando [19]:

make menuconfig -f makefile ARCH=arm CROSS_COMPILE=arm-linux-

El comando anterior provee de un entorno gráfico para establecer los parámetros de configuración de RTAI. Una vez seleccionados los parámetros deseados, el entorno de configuración se finaliza y de manera automática se inicia la compilación de RTAI. Por defecto, el módulo de RTAI se encuentra en el directorio */usr/realtime/modules*, con el nombre de *rtai_hal.o*, este archivo debe de ser copiado al iPaq.

5.1.3. Compilación del kernel de tiempo real

Durante la compilación del kernel de tiempo real, es posible establecer la política de planificación empleada en él. Para llevar a cabo la generación del módulo del kernel de tiempo real, se emplea el archivo *Makefile* que se encuentra en el directorio de archivos fuente del kernel. En el archivo *Makefile* se debe de agregar cualquiera de las siguientes líneas:

CFLAGS = $(CFLAGS) -DRATEMONOTONIC
o
CFLAGS = $(CFLAGS) -DROUNDROBIN

La primera establece una planificación de tipo *Rate Monotonic* y la segunda una planificación de tipo *Round Robin*.

El módulo generado se llama *ZINUX* y al ser instalado en la iPaq pone a disposición de los procesos de tiempo real los servicios ofrecidos por los manejadores del kernel.

5.2. Los procesos de tiempo real

Al implementar una aplicación de tiempo real haciendo uso del kernel desarrollado, es necesario dividirla en dos partes. La primera parte debe de

incluir a las actividades que requieren cumplir con restricciones de tiempo, como puede ser el muestreo de datos o el envío de señales de control, y por lo tanto deben de ser implementadas como procesos de tiempo real dentro del kernel. La segunda parte corresponde a actividades que no tienen que cumplir restricciones de tiempo, y pueden ser implantadas como procesos de usuario, como por ejemplo la visualización de datos.

Al realizar la división de las aplicaciones del modo descrito previamente, reducimos el tiempo de cómputo requerido por cada una de ellas dentro del kernel de tiempo real, sin embargo, se crea la necesidad de establecer un mecanismo de comunicación entre las dos partes de la aplicación, ya que una se ejecutara en el espacio del kernel de Linux, y la otra en el espacio de los procesos de usuario. Para resolver este problema, se han implantado dispositivos virtuales de lectura y escritura que mantienen un comportamiento de FIFO, y pueden ser accedidos por los procesos ejecutándose en el kernel de tiempo real y por los procesos de usuario ejecutándose en Linux.

Se ha desarrollado un módulo del kernel de Linux que contiene funciones que permiten a los procesos de tiempo real realizar las siguientes operaciones en los dispositivos virtuales:

- Inicializar el dispositivo.
- Leer datos de él.
- Escribir datos a él.
- Liberar el dispositivo.

Además, el módulo incluye funciones que permiten a los procesos de usuario acceder a los dispositivos virtuales como a un dispositivo de tipo carácter, que se encuentra referenciado en el directorio de dispositivos de Linux (*/dev*).

Estos dispositivos no forman parte del kernel de tiempo real, sólo son empleados como una herramienta de comunicación entre los procesos de tiempo real y los procesos de usuario, la persona que se encuentre haciendo uso del kernel puede implantar algún otro mecanismo alternativo, como pueden ser las variables compartidas, o inclusive otra versión de los dispositivos aquí descritos.

5.3. Estructura de un proceso de tiempo real

Como se explicó, los procesos de tiempo real deben de ser implementados dentro de un módulo del kernel de Linux, el cual cuenta con dos funciones básicas *init_module()* y *clean_module()*. La función *init_module()*, es ejecutada sólo una vez, al momento de integrar el módulo al kernel, por lo que se sugiere en ella realizar lo siguiente:

- Crear los buzones empleados por el proceso de tiempo real.
- Inicializar los semáforos requeridos en el proceso de tiempo real.
- Inicializar dispositivos virtuales que vaya a emplear el proceso.
- Crear el proceso de tiempo real.

Los tres primeros pasos pueden ser realizados en un orden distinto, pero es importante haber inicializado todos los recursos requeridos por el proceso de tiempo real antes de que sea creado.

La función *clean_module()* también es ejecutada una sola vez, al momento de dar de baja del kernel al módulo. En ella se pueden liberar todos los recursos empleados por el proceso de tiempo real, además de eliminarlo del kernel. Aunque también es posible liberar los recursos dentro del mismo proceso, y permitir que finalice por sí mismo al llegar al fin de la función que ejecuta.

La figura 5.1 muestra un diagrama de flujo que presenta la estructura de las funciones a ser ejecutadas por los procesos de tiempo real. Se observa que estas funciones deben de estar constituidas de un ciclo, en donde cada iteración del ciclo representa a una instancia del proceso de tiempo real. Si no se conoce el tiempo de cómputo exacto de cada iteración, se puede echar mano de la función *completa_iteración()* que se encarga de agregar el tiempo de cómputo requerido para completar el tiempo total de cómputo de la instancia, que es establecido al momento de crear el proceso.

Para que un proceso pueda hacer uso de las primitivas disponibles en los diferentes manejadores, es necesario que incluya en el código fuente de la aplicación el archivo de cabecera correspondiente a cada manejador requerido. La tabla 5.1 muestra el nombre del archivo de cabecera para cada uno de los manejadores disponibles en el kernel.

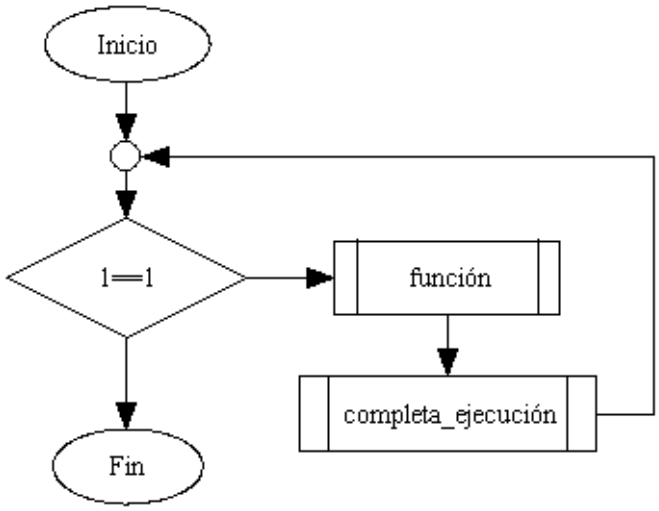

Figura 5.1: Estructura de un proceso de tiempo real.

Manejador	Archivo de cabecera
Procesos	manproc.h
Semáforos	mansem.h
Buzones	manbuzon
Primitivas de tiempo	mantiem.h
CPU (Planificador Round-Robin)	mancpu.h
CPU (Planificador Rate Monotonic)	treal.h

Tabla 5.1: Archivos de cabecera correspondientes a cada manejador del kernel de tiempo real.

5.4. Ejecución de las pruebas

El principal objetivo de las pruebas realizadas es verificar el funcionamiento correcto del mecanismo de planificación *Rate Monotonic* en el kernel de tiempo real. Las pruebas consisten en ejecutar un conjunto de tareas de manera concurrente, bajo las siguientes consideraciones:

- Todas las tareas de tiempo real son independientes.
- Las tareas se encuentran listas para ser ejecutadas justo al inicio de su período, y liberan el CPU sólo cuando su ejecución ha finalizado.

- Una tarea puede ser expulsada del CPU en cualquier momento de su ejecución.
- El plazo de las tareas coincide con el inicio de su siguiente período.
- Las tareas de mayor prioridad son aquellas que tienen el menor período, la criticidad de las tareas no es considerada.
- La ejecución de las tareas es consistente con su prioridad bajo *Rate Monotonic*: una tarea nunca es ejecutada cuando una tarea de mayor se encuentra lista para ser ejecutada.

Con los puntos anteriores en mente, se generaron diferentes grupos de tareas que permiten observar el comportamiento del kernel a lo largo de un hiper-período de ejecución de cada grupo, las características de cada uno de ellos son:

- En el primer grupo se tienen tres tareas que cumplen con la restricción planteada en la ecuación 2.2, por lo que no existen perdidas de plazos.
- El segundo grupo, presenta un conjunto de tres tareas que no cumplen la restricción 2.2, y muestra la detección de la perdida de plazos en el kernel.
- El tercer grupo, a pesar de no cumplir con el límite impuesto por la ecuación 2.2, tienen una planificación sin perdida de plazos de las tareas.

Haciendo uso del sistema de bitácoras del kernel de Linux, en cada ejemplo se despliega un mensaje de texto correspondiente a cada uno de los siguientes eventos:

- El arribo de la instancia de una tarea al sistema.
- La finalización del tiempo de cómputo de la instancia de una tarea.
- El desalojo del CPU de una tarea por el arribo de otra tarea de mayor prioridad.
- La perdida del plazo de una instancia de una tarea.

A través de los mensajes desplegados en el sistema, se construyeron las gráficas que muestran la ejecución de las tareas en él.

5.4.1. Ejemplo 1

La tabla 5.2, muestra un conjunto de tres tareas cuya utilización total del procesador es de 0.775, y por lo tanto podemos asegurar que será planificable bajo el algoritmo de *RM*. En la figura 5.2 se presenta la ejecución de un hiper-período estas tareas.

Tarea	Período	Cómputo	Utilización
τ_1	5	1	0.2
τ_2	8	3	0.375
τ_3	40	8	0.2
			0.775

Tabla 5.2: Tareas que cumplen con 2.2 por lo que son planificables bajo *RM*

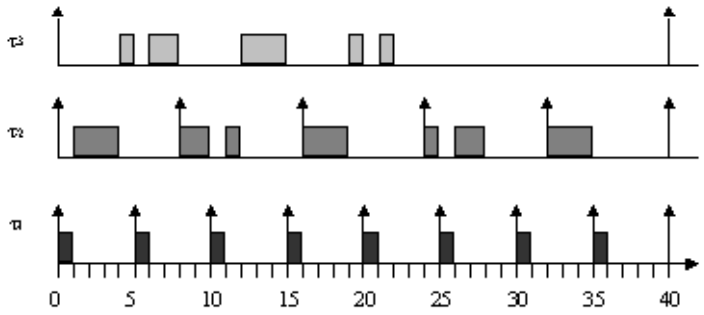

Figura 5.2: Planificación de tareas que cumplen con 2.2

5.4.2. Ejemplo 2

La tabla 5.3 presenta un conjunto de tres tareas que no cumplen con 2.2, por lo que no es posible asegurar su ejecución correcta en el sistema, y en la figura 5.3 se observa la perdida de plazos de la tarea de menor prioridad.

5.4.3. Ejemplo 3

La restricción impuesta por 2.2 es necesaria mas no suficiente, y si un conjunto de tareas no cumple con ella no es posible asegurar la perdida de plazos de alguna de las tareas. En la tabla 5.4 se tiene un conjunto de tres

Tarea	Período	Cómputo	Utilización
τ_1	5	1	0.2
τ_2	6	2	0.333
τ_3	7	3	0.428
			0.961

Tabla 5.3: Tareas que no cumplen con 2.2 y no son planificables bajo *RM*

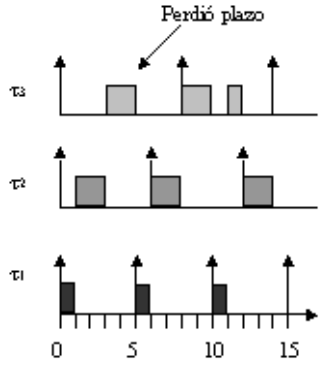

Figura 5.3: Planificación de tareas que no cumplen con 2.2

tareas cuya utilización total es de 1, sin embargo, en la figura 5.4 se observa que no existe perdida de plazos en alguna de las tareas.

Tarea	Período	Cómputo	Utilización
τ_1	6	2	0.333
τ_2	10	4	0.4
τ_3	30	8	0.266
			1

Tabla 5.4: Tareas que a pesar de no cumplir con 2.2 son planificables bajo *RM*

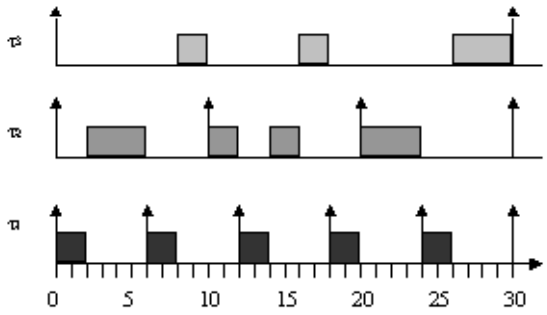

Figura 5.4: Planificación de tareas que no cumplen con 2.2, pero no existen perdidas de plazos

Capítulo 6

Conclusiones y trabajo futuro

6.1. Conclusiones

El creciente número de dispositivos móviles ha generado la necesidad de crear aplicaciones específicas para estos, algunas de las cuales demandan alta confiabilidad, y por tanto, sus resultados deben ser correctos, precisos y predecibles. En un sistema de cómputo, el encargado de asegurar que los procesos cumplan con las restricciones que se les han impuesto es el sistema operativo.

En la tesis se diseñó e implementó un kernel basado en Linux, para la iPaq H3950, el cual fue desarrollado haciendo empleando el enfoque de kernel dual planteado en RTLinux y RTAI, en la que se ejecuta a más de un kernel en la misma cómputadora. Al inicio del trabajo de tesis se estudiaron las diversas posibilidades existentes para su implementación, que incluyen:

- Desarrollar un kernel desde cero.
- Modificar un sistema operativo existente.

Observamos que la segunda aproximación es más viable, pues es posible hacer uso de las herramientas de desarrollo y depuración existentes en el sistema a modificar. En tanto que, al crear el sistema desde cero no se cuenta con esta ventaja.

Otro aspecto importante al decidir modificar un sistema operativo existente, Linux en este caso, fue el número de usuarios y desarrolladores involucrados en proyectos relativos a él. En principio, esto resulta una ventaja por la documentación existente y la cantidad personas que podrían brindarnos su ayuda en la solución de problemas a lo largo del proyecto. Sin embargo, un punto de mÃ¡s peso es el hecho de que el impacto del trabajo realizado puede ser mayor.

RTAI es un sistema en el que un gran número de personas participa en su mantenimiento. Existe soporte para diferentes arquitecturas, incluyendo: *Intel x86*, *MIPS* y *ARM* [1] , aunque, previo a nuestro trabajo no existía una versión del sistema que pudiera ejecutarse en la iPaq H3950.

En la sección 3.1 se describe la arquitectura de RTAI empleada en su implantación en la PDA, se observa que consta de dos módulos principales: la capa de abstracción de hardware y los servicios de RTAI. A diferencia del segundo módulo, la capa de abstracción de hardware, es dependiente de la plataforma. Al no existir una versión específica de la HAL para la PDA, fue necesario llevar a cabo una implementación de la misma dentro del kernel de Linux para la PDA, y así permitir una ejecución simultanea del kernel de tiempo real y el kernel de Linux.

Los módulos del kernel de Linux modificados debido a la introducción de la capa de abstracción de hardware son:

- Manejo de interrupciones.
- Manejo de excepciones (interrupciones de software).

El número de archivos en el kernel de Linux modificados son veinte, sin embargo, los cambios en cada uno involucran unas pocas líneas, y en su mayoría son simples, como se explica en la sección 3.5.

Al inicio, el kernel fue implantado en una computadora con procesador *Intel Centrino*, debido a que la capa de abstracción de hardware ya existía para esta arquitectura. Posteriormente, el kernel fue migrado al procesador *Intel XScale* de la iPaq, por lo que fue necesario realizar estudios sobre:

[1] Siendo esta la base para el procesador *Intel XScale de la iPaq H3950*

- Sistemas operativos.
- Sistemas de tiempo real
- Algoritmos de planificación en sistemas de tiempo real.
- Arquitectura de RTAI.
- Arquitectura del iPaq H3950.
- Arquitectura del procesador *Intel XScale*.

Se decidió emplear la metodología previa, ya que, a consecuencia de sus limitaciones en cómputo y almacenamiento, en la PDA no se cuentan con los medios suficientes para la creación y prueba de las aplicaciones. Se requiere contar con mecanismos que permitan escribir, compilar y verificar el funcionamiento de los programas, en nuestro caso se hizo uso de un *cross-compiler*, desarrollado en trabajos anteriores en el Laboratorio de Tiempo Real. A las personas interesadas en la escritura de aplicaciones para dispositivos con este tipo de restricciones, se recomienda también hechar mano de emuladores y herramientas de depuración tales como *gdb (The GNU Debugger)*.

Una vez que RTAI se encuentra operando de manera apropiada en la PDA, es posible integrar el kernel de tiempo real, que cuenta con servicios para:

- Creación y eliminación de procesos.
- Planificación de procesos mediante *Round-Robin* y *Rate Monotonic*.
- Comunicación entre procesos a través de buzones.
- Sincronización de procesos con semáforos.
- Retrazo en la ejecución del proceso.

Haciendo uso de los servicios incluidos en el kernel, es posible tener tareas de tiempo real períodicas ejecutandose de manera concurrente en la iPaq H3950.

Como resultado del trabajo realizado se obtuvó un kernel con las siguientes características:

- **Código simple de entender.** En el kernel sólo se incluyeron los servicios de nuestro interés, lo que facilitará su entendimiento y modificación. De forma que podrá ser empleado en proyectos futuros o inclusive en la enseñanza de sistemas operativos o sistemas de tiempo real.
- **Tamaño reducido (29 KBytes).** Este hecho es importante debido a que los dispositivos móviles tiene limitaciones en cuanto a su capacidad de almacenamiento.
- **Fácil migración.** El kernel ha sido escrito en su mayoria en lenguaje C, excepto el cambio de contexto entre procesos, ya que es dependiente de arquitectura, y al migrar el kernel a otra plataforma únicamente este bloque de código deberá ser modificado.

En la sección 1.2.2 se planteó, que una de las principales ventajas de tener corriendo al kernel de tiempo real de manera concurrente con Linux, es el poder hacer uso de los servicios implantados en el kernel de éste, incluyendo protocolos de comunicación y controladores de dispositivos.

Si bien en la etapa de pruebas se confirmó el funcionamiento apropiado de los algoritmos de planificación, debemos notar que todas las pruebas son realizadas con procesos dentro de la misma computadora. Un punto pendiente, de gran relevancia, es crear un conjunto de tareas de tiempo real, residentes en diferentes dispositivos y a través de los módulos de comunicación existentes en Linux puedan interactuar. Lo anterior requiere estudiar la forma en cómo los componentes de Linux han sido escritos, pero ya no es necesario construirlos.

En este punto debemos resaltar haber empleado investigaciones existentes como base para nuestro trabajo, por ejemplo la implementación del kernel de Linux y RTAI para la plataforma ARM, puesto que constituyeron una guia muy importante. De la misma forma, se espera que nuestro trabajo sea de utilidad para estudios futuros.

6.2. Trabajo futuro

Una vez implantado el kernel de tiempo real en la iPaq, es posible realizar diferentes extensiones o modificaciones al mismo, así como la implantación

de módulos que hagan uso de sus servicios y permitan aprovechar las características con que cuenta. Algunas de las posibles modificaciones a realizar en el kernel son:

- Modificar el planificador de tipo *Rate Monotonic* para tener un planificador *EDF* [2].
- Incluir algoritmos de planificación propios.
- Agregar el soporte a tareas esporádicas.
- Modificar (o agregar) las primitivas para manejo de semáforos de forma que sean no bloqueantes.
- Implantarlo en arquitecturas parecidas a la de la iPaq H3950.
- Agregar módulos necesarios para la existencia de procesos distribuidos. Lo que implica tener que proveer al kernel con:
  - Algoritmos de asignación de tareas entre procesadores.
  - Comunicación y sincronización entre procesos en diferentes computadoras.

Referente al último punto, en el Laboratorio de Tiempo Real se desarrolló la tesis titulada S̈eguridad en Sistemas Multimedia de Tiempo Real¨ [20], en la que se plantea un esquema de comunicación entre dispositivos (incluyendo dispositivos móviles) en una red inalambrica, que está compuesto de diferentes niveles de seguridad y en cada uno de ellos se ofrece una calidad de servicio específica a los procesos.
Un trabajo de interés es integrar al kernel de tiempo real con la aplicación descrita en [20], para contar con un entorno en el que el nivel de seguridad en la comunicación entre los procesos pueda controlarse dependiendo de las restricciones temporales impuestas a cada uno.

[2] Por la forma en que ha sido implantado el algoritmo Rate Monotonic, sólo es requerido un cambio al momento de asignar la prioridad de las tareas.

Bibliografía

[1] Cloutier Pierre and Mantegazza Paolo. DIAPM-RTAI Position Paper. *Real Time Operating Systems Workshop*, 2000.

[2] M. Barabanov. A Linux-based RealTime Operating System, 1997. at. citeseer.ist.psu.edu/barabanov97linuxbased.html.

[3] Ayers and Barabanov Victor Yodaiken. Introducing Real-Time Linux. *Linux J.*, 1997(34es):5, 1997.

[4] Khawar M. Zuberi, Padmanabhan Pillai, and Kang G. Shin. EMERALDS: a small-memory real-time microkernel. In *Symposium on Operating Systems Principles*, pages 277–299, 1999. at. citeseer.ist.psu.edu/zuberi99emeralds.html.

[5] Juan A. de la Puente Alfaro. Apuntes del Doctorado en Sistemas de Tiempo Real, 2002-2003.

[6] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Applied Operating System Concepts*. John Wiley & Sons, Inc., New York, NY, USA, 2000.

[7] Giorgio C. Buttazzo and Giorgio Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[8] TenAsys Corporation. "iRMX86 Real-Time Operating System for Intel Architecture". at. http://www.tenasys.com/irmx.html.

[9] José Ismael Ripoll Ripoll. Planificación en Sistemas de Tiempo Real, 2000.

[10] C.L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real Time Environment. *J. ACM*, 20(1):46–61, 1973.

[11] J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2:237–250, 1982.

[12] William von Hagen. Real-time and performance improvements for the 2.6 linux kernel. *Linux J.*, 2005(134):8, 2005.

[13] Nicholas Mc Guire, FSMLabs. MiniRTL - Hard Real Time Linux for Embeddes Systems. *Dedicated Systems Magazine 01q3 RTOS Update*, 2001.

[14] MontaVista Software Inc. Enabling Native Linux Real-time Response, 2004. at. http://www.mvista.com/dswp/ds_realtime.pdf.

[15] Daniel Bovet and Marco Cesati. *Understanding the Linux Kernel, Second Edition*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[16] David Seal. *ARM Architecture Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[17] P. Mantegazza. Dissecting DIAPM RTHAL-RTAI, 1997. at. http://www.aero.polimi.it/ rtai/documentation/articles/paolo-dissecting.html.

[18] Kwan Lowe. Kernel Rebuild Guide, 2004. at. http://www.digitalhermit.com/linux/kernel-Build-HOWTO.html.

[19] Bruno Rouchouse. RTAI Installation Guide (Kilauea version). at. http://www.fdn.fr/ brouchou/rtai/rtai-doc-prj/installation-guide.html.

[20] Luis de Jesús González . Seguridad en Sistemas Multimedia de Tiempo Real , 2005.

Los abajo firmantes, integrantes del jurado para el examen de grado que sustanterá **Francisco Javier Zuluaga Ramírez**, declaramos que hemos revisado la tesis titulada:

Kernel basado en Linux para una PDA, con soporte para procesos de tiempo real

Y consideramos que cumple con los requisitos para obtener el Grado de Maestría en Ciencias en la especialidad de Ingeniería Electrica opción Computación.

Atentamente,

Nombre del jurado 1

Nombre del jurado 2

Nombre del jurado 3