Programa oficial de postgrado en
Ciencias, Tecnología y Computación
Master en Computación
Facultad de Ciencias - Universidad de Cantabria

Tesis de Máster

Adaptación de la arquitectura 'linux_lib' de MaRTE OS a multiprocesadores

Daniel Medina Ortega

medinad@unican.es

Tutor Mario Aldea Rivas
Grupo Computadores y Tiempo Real
Dept. Electrónica y Computadores

29 julio 2010
Curso 2009-10

Agradecimientos

A mi Familia, por el apoyo recibido a lo largo de todos estos años de estudios.

A Tamara, por estar siempre pendiente, apoyarme y aguantarme.

A Mario y Michael, por darme la oportunidad de realizar este proyecto y por toda la ayuda prestada para su finalización.

A María, el laboratorio está muy solitario últimamente.

A todos los compañeros del grupo CTR, por en gran ambiente que se respira.

A los amigos, por todos esos cafés, partidos y ratos de distracción.

Índice general

Índice de figuras

Índice de cuadros

p tu 1

Introducción

En este capítulo se presenta el sistema operativo de tiempo real MaRTE OS que ha sido utilizado en este proyecto, también se realiza un pequeño estudio de lo que son los sistemas multiprocesador. Finalmente se describen cuales han sido los objetivos que se han tratado de conseguir en este proyecto.

1.1. MaRTE OS

MaRTE OS [1, 2, 3] es un sistema operativo de tiempo real estricto que proporciona una base para el desarrollo de aplicaciones empotradas y que sigue el subconjunto mínimo de tiempo real definido en POSIX.13 (perfiles de entornos de aplicaciones de tiempo real). MaRTE OS proporciona un entorno fácil de utilizar para desarrollar aplicaciones de tiempo real con múltiples hilos.

La mayoría de su código se encuentra escrito en Ada con algunas partes en C y ensamblador. Permite el desarrollo cruzado de aplicaciones en Ada y C/C++ utilizando los compiladores GNU Gnat y Gcc.

Las principales características de MaRTE OS son:

- Cumple el perfil mínimo de sistemas de Tiempo Real (PSE51) definido en el estándar POSIX.13.
- Proporciona concurrencia a nivel de hilo.
- Orientado para aplicaciones de tiempo real embebidas.
- Orientado para aplicaciones estáticas (todos los recursos pre-colocados y configurables en tiempo de compilación).
- Todos los servicios con tiempos acotados de respuesta.
- Espacio de direcciones simple y sin proteger. Compartido por el kernel y la aplicación.
- Multilenguaje: soporta aplicaciones Ada, C, C++ y Java. En aplicaciones con varios lenguajes, se realiza una planificación coherente entre los hilos de distintos lenguajes.

Aunque originalmente fue concebido para sistemas empotrados, MaRTE OS ha sido también adaptado para comportarse como una librería de hilos POSIX para el sistema operativo Linux. Por lo que actualmente, MaRTE OS puede ser utilizado en las arquitecturas x86, linux y linux_lib.

x86 Cuando se compila para esta arquitectura, la aplicación MaRTE OS toma la forma de un ejecutable ELF, que incluye el sistema operativo y el código de la aplicación de usuario. Es un programa *stand-alone* que puede ser ejecutado en un PC x86 desnudo. Para poder lanzar el ejecutable en el computador empotrado se necesita un cargador de arranque local o por red. En esta arquitectura el entorno cruzado de MaRTE OS se encuentra formado por un PC ejecutando Linux como "Host" y un PC 386 desnudo como "Target", ambos sistemas conectados a través de una red LAN (para la carga del programa) y una línea serie (para el depurado remoto).

linux Cuando se compila para esta arquitectura, la aplicación MaRTE OS toma la forma de una aplicación estándar de Linux que es ejecutada

como cualquier otro proceso de usuario de Linux. Utilizar esta arquitectura es como ejecutar la aplicación MaRTE OS en un emulador hardware. Las aplicaciones generadas son prácticamente idénticas a las generadas por la arquitectura x86.

Se utiliza una librería C propia de MaRTE OS (`libmc.a`) y el sistema de ficheros de MaRTE OS. Las únicas diferencias con la arquitectura x86 se encuentran en la capa de interfaz con el hardware (HAL).

- Se utiliza el temporizador de Linux en vez del temporizador hardware.
- Las señales de Linux hacen el papel de interrupciones hardware.
- Los drivers de consola y teclado se han modificado para que escriban directamente sobre los descriptores de fichero `stdin` y `stdout` (en vez de realizar una gestión directa del hardware).

linux_lib Cuando se compila para esta arquitectura, la aplicación MaRTE OS toma la forma de una aplicación estándar de Linux que es ejecutada como cualquier otro proceso de usuario de Linux. MaRTE OS se comporta como una librería de hilos POSIX que es utilizada para proporcionar concurrencia a nivel de librería a aplicaciones Ada y C que se ejecutan sobre Linux. En el caso del lenguaje Ada, MaRTE OS es utilizado como la librería Pthread que soporta las tareas Ada para el compilador de GNAT. Las principales diferencias entre esta arquitectura y la linux son:

- Se utilizan las librerías estándar de Linux, en particular la librería estándar C (en la arquitectura linux se utiliza una librería estándar C propia de MaRTE OS libmc.a).
- Como consecuencia de lo anterior, desde una aplicación MaRTE OS se puede acceder al sistema de ficheros de Linux y a otras características del sistema utilizando los interfaces POSIX, Linux o Ada como en cualquier otro programa de Linux.

En la figura 1.1 se aprecia la arquitectura linux_lib de MaRTE OS para aplicaciones escritas en C y Ada.

MaRTE OS configurado para la arquitectura linux, o preferentemente, para linux_lib, representa una buena elección para impartir cursos de programación de POSIX o Ada de tiempo real.

Las arquitecturas linux y linux_lib pueden ser utilizadas como un mecanismo rápido para verificar el comportamiento funcional de las aplicaciones

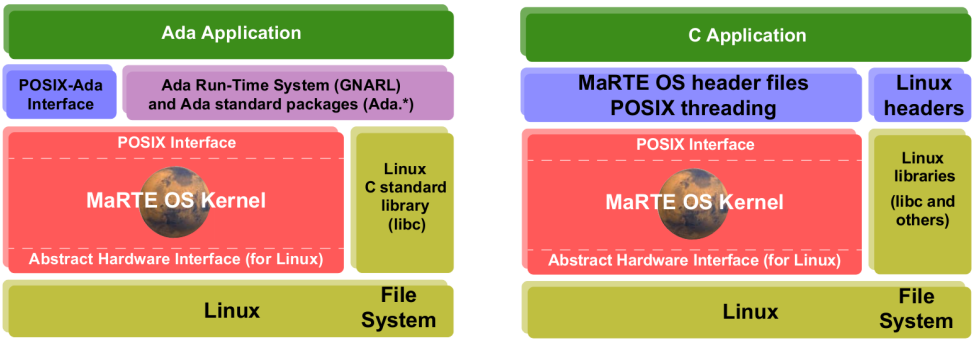

Figura 1.1: *Arquitectura linux_lib*

antes de realizar las pruebas definitivas en una máquina empotrada. Hay que tener en cuenta que cuando se utilizan estas arquitecturas no se puede conseguir un comportamiento de tiempo real estricto, ya que los ejecutables resultantes son procesos estándar que comparten el tiempo de ejecución con todos los demás procesos del sistema de acuerdo con las políticas de planificación de Linux. Al igual que el resto de los procesos se ven afectados por el intercambio de memoria (*memory swapping*), las actividades del kernel de Linux, etc.

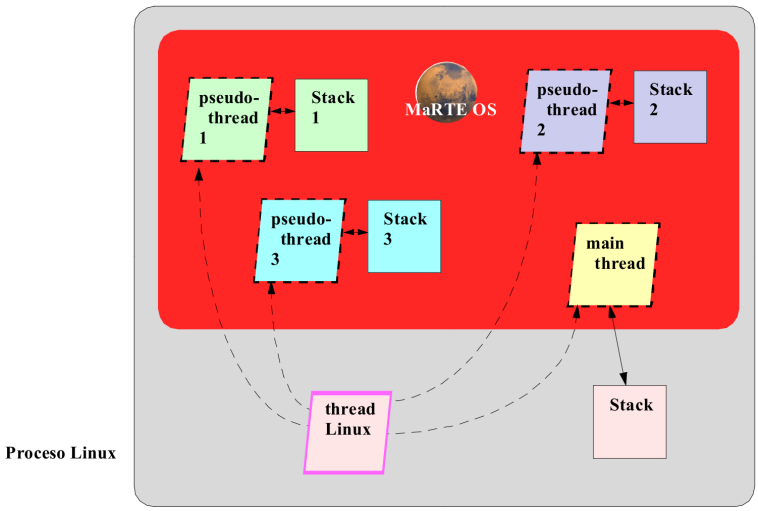

Figura 1.2: *Implementación de threads en MaRTE OS.*

Uno de los aspectos más interesantes de MaRTE OS como sistema operativo empotrado es que permite la gestión de tareas independientes. Para ello utiliza la interfaz *pthread* de POSIX con una implementación propia. En la arquitectura linux_lib la aplicación de MaRTE OS se ejecuta sobre un único proceso de Linux. Como se muestra en la figura 1.2, MaRTE OS, dentro de la memoria que Linux le ha proporcionado, coloca las diferentes pilas de cada tarea y diferencia las instancias de cada tarea. De esta forma,

aunque se está compartiendo la memoria las tareas poseen su propio espacio de trabajo. Para seleccionar qué tarea se ejecuta en cada momento, MaRTE OS modifica el contador de programa (que indica cual va a ser la siguiente instrucción a ejecutarse) y el registro de la pila (*stack*) del proceso Linux, sustituyéndolos en cada momento por los correspondientes a la tarea que se desea ejecutar. MaRTE OS hace que las tareas se ejecuten en el orden marcado por los parámetros de planificación al igual que lo harían en un sistema empotrado.

MaRTE OS se desarrolla en el grupo de Computadores y Tiempo Real del departamento de Electrónica y Computadores de la Universidad de Cantabria. Se utiliza principalmente para temas educativos y experimentales; además hay abiertos varios proyectos para su utilización en entornos industriales.

1.2. Sistema SMP

Un sistema SMP, o sistema de multiprocesado simétrico (symmetric multiprocessing en inglés), se trata de una arquitectura donde dos o más procesadores idénticos se encuentran conectados a una memoria principal compartida. En el caso de los procesadores multi-núcleo, la arquitectura se aplica a los diferentes núcleos del procesador, tratándolos por separado como procesadores. Este tipo de arquitectura es denominada UMA, *Uniform Memory Access*. La arquitectura de memoria **UMA** es una arquitectura de memoria compartida donde todos los procesadores acceden a la memoria de manera uniforme, o lo que es lo mismo, el tiempo de acceso a una posición de memoria es independiente de cual es el procesador que realiza la solicitud o de que chip de memoria contiene el dato a transferir. Como se puede ver en la figura 1.3 la memoria es única para todos los procesadores y acceden a ella mediante el mismo bus. Esto es la base de los problemas de coherencia de caché que acompañan a estos sistemas, sin embargo es importante que se mantenga la uniformidad en el acceso a la memoria. Para mantener esta uniformidad las cachés de cada procesador se encuentran sincronizadas entre sí. Por el contrario en los sistemas NUMA cada procesador tiene asociada una memoria local con la que trabaja de forma rápida, sin embargo, cuando tiene que acceder a un dato en la memoria local de otro procesador el tiempo empleado es alto.

Los sistemas UMA permiten a los procesadores trabajar con cualquier tarea de forma independiente a la localización de los datos de ésta en la memoria, con la restricción de que una tarea dada no se encuentre en ejecu-

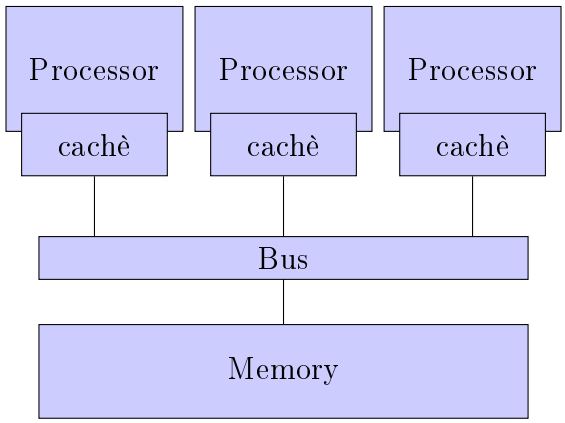

Figura 1.3: *Arquitectura UMA*

ción en dos o más procesadores al mismo tiempo. Con el soporte apropiado por parte del sistema operativo, estos sistemas pueden mover las tareas entre los procesadores para que la carga de trabajo sea repartida de forma eficiente.

Hoy en día la mayoría de los ordenadores para ámbito personal que se encuentran en el mercado contienen procesadores con varios núcleos debido a que el incremento del rendimiento no puede ser conseguido aumentando la frecuencia de trabajo sin disparar el consumo. Esta tendencia en los ordenadores personales se está trasladando también a la computación empotrada, donde cada vez se ejecutan un mayor número de aplicaciones y el consumo es un aspecto muy importante [4]. Una de las empresas punteras en la fabricación de procesadores para entornos empotrados, como es ARM, tiene actualmente un modelo con múltiples núcleos configurables [5]. En la figura 1.4 obtenida de [5] se observa como a partir de una cantidad de carga es más eficiente poseer un mayor número de núcleos.

El hecho de disponer varios procesadores, y que estos accedan de forma simultanea a un espacio de memoria compartida, va a provocar una serie de situaciones que en sistemas mono-procesador no se producen. La memoria compartida deberá ser protegida para que no se produzcan accesos simultáneos desde los diferentes procesadores, pudiendo llegar a provocar incoherencias en los datos que se están utilizando. Tener varias unidades de procesamiento permite la ejecución concurrente de tareas lo que nos conduce a nuevas formas de planificación y sincronismo.

Cuando se produce un evento de planificación, como puede ser que una tarea que está en ejecución se suspenda, bloquee o ceda el procesador, o que se active una tarea que se encontraba suspendida, el sistema operativo sigue un conjunto de reglas para decidir que tarea va a ser planificada. Normalmente

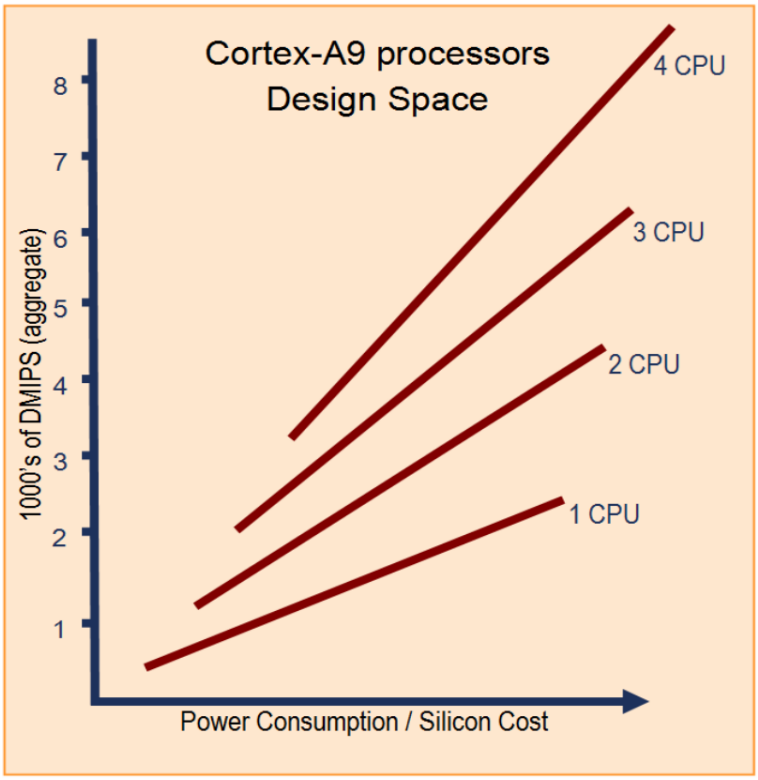

Figura 1.4: *Progresión del rendimiento con el aumento de los núcleos y consumo de energía.*

la regla más utilizada es la de definir la prioridad de las tareas, seleccionando la más prioritaria en cada momento; en los SMPs a esto se añade el concepto de afinidad. Se trata de indicar cual es el procesador o los procesadores donde se quiere que se ejecute la tarea dada, por lo que dentro de las tareas más prioritarias se planificará aquella que sea afín al procesador. Inicialmente se concibió para reducir el intercambio de tareas entre los procesadores, ya que si se produce el cambio de procesador hay que migrar a la caché del procesador toda la información asociada a la tarea. Si permanece ejecutándose en el mismo procesador, es muy probable que parte o toda la información necesaria se encuentre en la caché (depende del uso de la caché que hagan las tareas).

El análisis de la planificabilidad en sistemas multiprocesador no está completamente resuelto y no se ha encontrado todavía un algoritmo óptimo. En sistemas de tiempo real, este análisis es importante puesto que nos permite prever como se va a comportar el sistema. Utilizando los mecanismos que proporciona la afinidad se puede reducir la complejidad del análisis.

Existen diferentes formas de encontrar la tarea más prioritaria para planificarla. En los sistemas monoprocesador suele haber una única cola donde

las tareas se ordenan en base a un conjunto de parámetros, que pueden ser fijos o variables en el tiempo, ya sean prioridades fijas, prioridades dinámicas, etc.

En los sistemas multiprocesador existe una aproximación a los sistemas monoprocesador que consiste en agrupar de forma fija las tareas en tantas colas como procesadores se tengan. Las tareas no van a poder cambiar de procesador, por lo que el problema se convierte en un problema de planificación monoprocesador. Por otro lado, se puede abordar desde un punto de vista más global, teniendo una única cola de prioridad donde la primera tarea ha de competir con todas aquellas que se encuentran en ejecución. Esta última forma conlleva un mayor coste computacional al tener que realizar comprobaciones más complejas y con un mayor número de tareas, sin embargo, permite una mejor gestión de la carga del sistema. Con un sistema de colas locales es posible que se de el caso de que un procesador se encuentre en un momento dado con una carga alta mientras que otros puedan estar completamente ociosos, y por el contrario, será más sencillo analizar su planificabilidad.

1.3. Objetivo

El objetivo principal de este proyecto es conseguir que MaRTE OS, originalmente un sistema operativo de tiempo real monoprocesador, en su arquitectura linux_lib, sea capaz de planificar tareas en varios procesadores.

Para la consecución del objetivo principal se tiene que trabajar en dos aspectos. Por un lado se ha de modificar el kernel de MaRTE OS para que el algoritmo de planificación sea capaz de asignar las tareas a los diferentes procesadores, de forma que se puedan utilizar los recursos disponibles de un sistema multiprocesador. Esto se convierte en un primer paso para obtener un sistema operativo multiprocesador de tiempo real.

Por otro lado se tiene que modificar la plataforma sobre la que se asienta MaRTE OS en Linux para que se puedan tener tantas unidades de procesamiento disponibles como se desee (normalmente el mismo número que procesadores) manteniendo la aplicación en un único proceso.

En la figura 1.5 se muestra una idea conceptual de la arquitectura buscada. Donde el proceso Linux contiene el kernel de MaRTE OS y la aplicación de usuario. El kernel crea varios hilos de Linux que emulan cada uno de

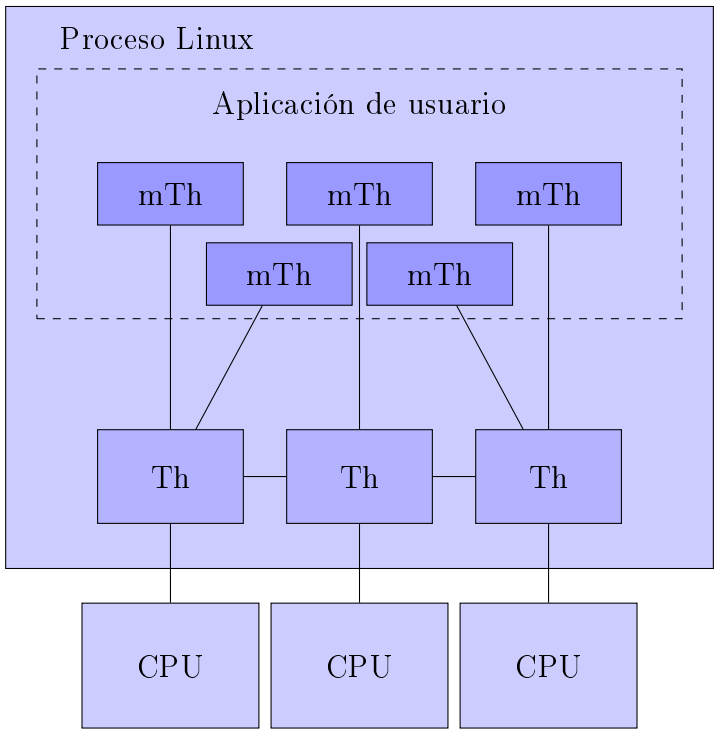

Figura 1.5: *Arquitectura linux_lib_SMP de MaRTE OS*

ellos un procesador. Por norma general se crea un hilo por cada procesador disponible, siendo a su vez cada hilo asociado a un procesador. Estos hilos pueden comunicarse entre ellos a través de primitivas del sistema operativo. La aplicación de usuario, dentro del proceso Linux, tiene tantos hilos como el usuario genere, los cuales son repartidos entre los procesadores por el planificador de MaRTE OS.

Como se ha visto en la descripción de MaRTE OS la arquitectura linux_lib, sobre la que se va a trabajar, no cumple de forma estricta las características de tiempo real, por lo que no es posible introducir entre los objetivos las características de tiempo real del sistema.

Este proyecto pretende ser el arranque de un proyecto con objetivos más amplios que intenta definir una interfaz de planificación multiprocesador para MaRTE OS, de forma que se puedan realizar pruebas de algoritmos de planificación de forma sencilla.

p tu 2

Soporte de Linux para SMP

Para poder cumplir la modificación de la plataforma presentada en la sección 1.3 se necesita un mecanismo para la creación de los hilos que van a emular los procesadores. La afinidad permite que cada hilo se ejecute siempre sobre el mismo procesador real, esto ayuda a mejorar la emulación del procesador. En los sistemas multiprocesador reales los diferentes procesadores se comunican entre ellos a través de interrupciones específicas denominadas IPI (*InterProcessor Interrupt*). Para emular las IPIs se utilizarán las señales POSIX, uno de los mecanismos de comunicación entre hilos proporcionado por Linux.

En este capítulo se presentan los diferentes servicios proporcionados por Linux que van a ser utilizados por MaRTE OS para implementar un sistema multiprocesador sobre su arquitectura linux_lib. Se abordan el tema de la creación de los hilos que van ha servir como unidades de cálculo, su afinidad a los diferentes procesadores de la máquina, los mecanismos de señalización

entre los hilos y la identificación del procesador en el que se encuentra la ejecución actual.

2.1. Creación de hread

Aunque estemos sobre un sistema multiprocesador, MaRTE OS no deja de ser una aplicación monoproceso, y se comportará como tal al ser ejecutado. Como se ha presentado en la sección 1.3 la forma que se ha escogido para que MaRTE OS disponga de varias unidades de proceso es la de lanzar desde la inicialización de MaRTE OS varios threads Linux que emularán los procesadores del sistema. El objetivo es poseer tantas unidades de ejecución como procesadores se desee, todo ello bajo un mismo proceso de Linux.

Antes de entrar en los detalles de las diferentes implementaciones que se han abordado, es interesante analizar las diferencias que existen entre proceso y hilo (*thread*). Los procesos y los hilos son entidades diferentes, aunque Linux los trata de forma similar. Es cierto que ambos son muy similares y son mecanismos que se utilizan para paralelizar aplicaciones. Sin embargo, un proceso es definido como una instancia de un programa que está siendo ejecutado, incluyendo las variables y la información que describe el estado del programa. Cada proceso es una entidad independiente a la que se le asigna recursos del sistema, y es ejecutado en un espacio de memoria separado, no pudiendo acceder a la información de otro proceso. Los métodos utilizados para la comunicación entre procesos suelen ser los ficheros, las tuberías o los sockets.

Por el contrario, el hilo o thread (que es un acortamiento de *hilo de ejecución*) se refiere a un camino particular de ejecución dentro de un proceso. Su funcionamiento específico depende del sistema operativo, pero en general un conjunto de hilos pueden compartir la información de un único proceso, comparten el espacio de memoria y los recursos del sistema. Se pueden comunicar directamente a través de variables u otras estructuras de memoria. Como comparten el espacio de memoria los cambios de contexto entre hilos son bastante más rápidos que entre procesos.

El sistema operativo Linux proporciona dos mecanismos para crear hilos: la interfaz POSIX y la función `clone`. En un primer momento se intentó utilizar la interfaz POSIX, pero debido a problemas que surgieron, y que se describen brevemente a continuación, se ha realizado una implementación a más bajo nivel, utilizando la llamada al sistema `clone` para generar los hilos.

2.1.1. Interfaz POSIX Linux

En un primer momento se intentó utilizar la interfaz *pthread*, interfaz POSIX que gestiona la creación, control y limpieza de hilos (threads).

Para poder utilizar la librería hay que incluirla de forma manual en el comando de compilación mediante la opción `-lpthread` (añade la librería con nombre `libpthread.a`). Sin embargo, MaRTE OS también utiliza la interfaz POSIX para la gestión de los hilos propios, por lo que para incluir la librería de gestión de hilos de Linux en MaRTE OS se hace una copia y se cambia su nombre por el de `libpthread_linux.a`. Aún así, siguen siendo dos librerías que definen las mismas funciones, lo que provoca que en el momento de enlazar los diferentes ficheros objeto del programa se produzcan varios conflictos con los nombres de los símbolos (nombres de funciones y variables). El enlazador (*linker*) no permite la existencia de una definición múltiple de un mismo símbolo.

Con el objetivo de permitir la compatibilidad de las dos librerías, se modificó las propiedades de la nueva librería (`libpthread_linux.a`) para evitar los conflictos entre los símbolos. Explorando con los comandos `nm` y `grep` la librería *pthread* de Linux se observa que el símbolo público de una función tiene correspondencia con otros símbolos internos con diferente nombre a los que existen en la librería de MaRTE OS. Las llamadas que necesitamos hacer a la librería de Linux se encuentran localizadas y son escasas, por lo que se optó por utilizar los símbolos internos de las funciones, debilitando los públicos para que no entren en conflicto con los símbolos de la librería de MaRTE OS. Los símbolos que entran en conflicto se modifican con el comando `objcopy` y las opciones `-weaken-symbol=<symbol>` para debilitar el símbolo y que se tome por defecto el símbolo fuerte perteneciente a MaRTE OS; y `-redefine-sym <old>=<new>` para modificar alguna llamada a símbolos que se habían debilitado. Se utilizó también el comando `ar` con las opciones `xv` y `rv` para extraer e insertar los ficheros objeto de la librería. De esta forma los símbolos originales hacen referencia a la implementación de threads de MaRTE OS, mientras que se llama a los símbolos internos de la librería Linux modificada para la creación de los threads que van a dar soporte multiprocesador.

Esta pequeña modificación de la librería implicó la modificación de algún símbolo interno extra y no se consiguió hacer funcionar de forma correcta y estable. Esto, junto con la previsión de que iba a continuar generando problemas al ir implementando más elementos de la librería `pthread` en MaRTE OS, propició la decisión de abandonar esta vía de trabajo.

2.1.2. Función 'clone'

Una vez abandonada la implementación a través de la librería *pthread* de Linux, se decidió aumentar un nivel de complejidad e intentar realizar la implementación a través de la llamada al sistema `sys_clone`. Esto implica que no se va a poder hacer uso de las funciones de la librería *pthread* que facilitan el manejo de los hilos.

La función `clone` es una función de biblioteca que llama de forma directa a la llamada al sistema antes mencionada. `clone` crea un proceso de la misma forma que lo hace `fork`. Sin embargo, en este caso `clone` permite especificar si se quiere compartir o no con el proceso hijo partes del contexto de ejecución del proceso invocador. En el caso que nos ocupa va a interesar permitir que los diferentes procesos que se creen compartan la información referente al sistema de ficheros (`CLONE_FS`), la tabla de descriptores de ficheros (`CLONE_FILES`) y la tabla con los manejadores de señal (`CLONE_SIGHAND`), pero no la máscara. Para reflejar un sistema SMP se hace que todos los procesos creados se ejecuten en el mismo espacio de memoria (`CLONE_VM`), siendo visible desde los procesos todas las modificaciones que realicen el resto. Para una mejor gestión de los procesos creados, por ejemplo, a la hora de enviar señales, se hace que los nuevos procesos sean tratados como hilos, de forma que todos compartan el mismo identificador de grupo (`CLONE_THREAD`). Por lo que Linux pasa a considerarlos hilos a todos los efectos.

La función `clone` tiene la siguiente forma:

```
#include <sched.h>
int clone(int (*fn)(void *), void *child_stack,
          int flags, void *arg);
```

Donde el primer argumento, `fn`, es un puntero a la función que va a ser ejecutada por el nuevo proceso. Cuando la función regresa, el proceso hijo finaliza con el mismo código de salida que la función. El último argumento, `arg`, se pasa a la función `fn` como argumento. El segundo argumento indica la posición de la pila utilizada por el proceso hijo. El proceso invocador debe preparar un área de memoria para la pila del hijo y pasar un puntero a dicha área (normalmente el puntero de pila apunta a la dirección más alta de la zona). En el argumento `flags` se indican las propiedades que ha de tener el proceso creado. Las banderas utilizadas son las que se indican entre paréntesis en el párrafo anterior. Al indicar la bandera `CLONE_THREAD` el proceso pasa a comportarse como un hilo en Linux.

Esta función permite generar tantos procesos o hilos como se desee. En la implementación multiprocesador de MaRTE OS se identifica cada hilo creado con una unidad de procesamiento, normalmente se generará un hilo por procesador real disponible. Siempre se podría generar un número mayor o menor de hilos en función de las necesidades, aunque se debe tener en cuenta que en este caso se perdería la analogía con un sistema SMP real, sobre todo cuando nos encontramos con más procesos que procesadores, pues el tiempo de ejecución se ha de compartir.

Inicialmente los hilos ejecutan un lazo infinito que no realiza ninguna operación útil. Más adelante, durante la inicialización de MaRTE OS, es cuando se planificará sobre este hilo una tarea de MaRTE OS.

Al especificar en la creación con `clone` la bandera `CLONE_THREAD` los procesos que estamos creando van a compartir el identificador de proceso (`PID`) con el proceso invocador, formando un grupo. Su diferenciación va a poder ser realizada mediante el identificador de thread (`TID`). Este identificador va a ser necesario para establecer la afinidad del hilo a un procesador y para implementar la comunicación entre los hilos a través de señales.

Para obtener el `TID` de un proceso es necesario realizar una llamada al sistema. El formato de la llamada es el siguiente:

```
tid = syscall (SYS_gettid);
```

2.2. Afinidad de lo hread

Como se ha mencionado en la sección 1.2 en sistemas con múltiples procesadores puede ser importante disponer de un mecanismo que permita que un proceso siempre se ejecute en el mismo procesador (de esta forma las variables de la tarea se encuentran en la caché del procesador). Evitando el movimiento innecesario de datos de un procesador a otro se puede aumentar el rendimiento del sistema.

Para realizar esto, el sistema operativo Linux proporciona una serie de primitivas y macros que permiten fijar la afinidad de un hilo a un procesador, o conjunto de ellos, determinado del sistema.

En nuestro caso el objetivo no es aumentar el rendimiento del sistema, sino que cada hilo siempre se ejecute sobre el mismo procesador. La idea subyacente es que cada hilo simula un procesador, si se consigue que cada hilo

esté siempre en el mismo procesador, la sobrecarga por encontrarnos ejecutando un programa sobre otro sistema operativo será mínima. Obviamente, no se va a poder considerar que la ejecución cumple los requisitos para poder ser llamada de tiempo real, pero sí que va a permitir poder enfrentarnos a los problemas derivados de una concurrencia real de los hilos del sistema.

Las funciones para fijar o leer el valor de la afinidad toman como argumentos el identificador de proceso (`pid`, en este caso se utiliza el `TID` devuelto por la función `clone`), el tamaño de la máscara y un puntero a la máscara. La máscara es un conjunto de bits donde cada bit identifica a un procesador, los bits activos indican los procesadores donde se permite ejecutar dicho hilo. Las macros permiten realizar modificaciones sobre la máscara de forma sencilla.

```
#include <sched.h>
int sched_setaffinity(pid_t pid,
                      unsigned int cpusetsize,
                      cpu_set_t *mask);
int sched_getaffinity(pid_t pid,
                      unsigned int cpusetsize,
                      cpu_set_t *mask);

void CPU_CLR(int cpu, cpu_set_t *set);
int CPU_ISSET(int cpu, cpu_set_t *set);
void CPU_SET(int cpu, cpu_set_t *set);
void CPU_ZERO(cpu_set_t *set);
```

Para no entrar en conflicto con futuras implementaciones de MaRTE OS que hagan uso de la afinidad de los hilos, las llamadas a las funciones que definen la afinidad de los hilos Linux de MaRTE OS se realizan a través de las llamadas al sistema propias de Linux `SYS_sched_setaffinity` y `SYS_sched_getaffinity`.

2.3. Se ale

Tener un sistema con varios procesadores implica que tarde o temprano se va a querer que se comuniquen entre ellos, ya sea para sincronismo o para compartir datos entre hilos. La comunicación más común y la que implementa la arquitectura IA-32 de Intel es a través de interrupciones [6]. En Linux la forma más sencilla para comunicar procesos y/o threads, y la que más

se parece a las interrupciones hardware, es a través de señales. Las señales no son más que unos mecanismos que permiten informar a los procesos de eventos que han sido provocados, por ellos mismos o por otros procesos. Es comparable con la gestión de interrupciones lógicas. Cuando llega una señal a un proceso el sistema interrumpe la ejecución normal de éste para ejecutar una función asociada a esa señal.

Como se ha visto anteriormente (sección 2.1.2) cada hilo que es creado va a compartir la misma tabla de manejadores de señal, por lo que ante la misma señal todos los hilos van a ejecutar la misma función. Así mismo, van a heredar del proceso llamante la máscara de señal, que podrá ser modificada para la habilitación y deshabilitación de las señales de forma independiente para cada hilo.

Para la gestión de la máscara de señales de cada hilo de Linux, MaRTE OS proporciona una interfaz para poder interactuar con las funciones pertenecientes al sistema operativo Linux sin tener conflicto con las mismas funciones que implementa MaRTE OS para la gestión de la máscara de las tareas internas. La gestión de la máscara de los hilos de Linux desde MaRTE OS se realiza con la interfaz que proporciona la función `linux_sigaction`, que permite especificar y/o examinar la acción asociada a la señal indicada, y la función `linux_sigprocmask`, que se utiliza para enmascarar y desenmascarar las señales asociadas al proceso en ejecución. En MaRTE OS no se ha definido un fichero de cabecera por lo que deberá declarase la función con el atributo `extern` antes de utilizarse. Dicho atributo indica que la función se encuentra definida en otro fichero de código objeto.

```
extern int linux_sigaction(int sig, const struct sigaction *act,
                    struct sigaction *oact);
extern int linux_sigprocmask(int how, const sigset_t *set,
                    sigset_t *oset);
```

Para el envío de señales desde Linux a los hilos creados se utiliza la llamada al sistema `SYS_tgkill` que permite el envío de una señal a un hilo específico dentro de un grupo. Su formato es

```
long sys_tgkill (int tgid, int pid, int sig);
```

donde `tgid` es el identificador del grupo de threads (hilos), `pid` es el del thread específico y `sig` es la señal que se desea enviar.

2.4. Identificación de la CPU

Para identificar cada núcleo o cpu del procesador sobre el cual se está ejecutando un código existe una instrucción ensamblador proporcionada por la arquitectura IA-32 de Intel [7]. La función que se utiliza es `cpuid` y la información que devuelve depende del valor que tomen algunos registros, así como el modelo del procesador. La identificación de cada núcleo se realiza a través del identificador de procesador que proporciona la APIC. Cuando se escribe el valor `0x01` en el registro `eax` y se invoca la función `cpuid`, el identificador se almacena en el byte superior del registro `ebx`.

Para invocar la instrucción ensamblador desde un código C se utiliza la siguiente línea de código

```
asm("cpuid" : "=b" (id) : "a" (0x01) );
```

donde se diferencian tres argumentos separados por dos puntos (:). El primero es el nombre de la instrucción. El segundo son los operadores de salida donde "`=b`" indica el registro `ebx` y `(id)` hace referencia a la variable C donde se almacenará el valor para su utilización en el programa. El tercer argumento indica que el valor de entrada se escribe en el registro `eax` ("`a`") y toma el valor `0x01`.

Otra forma de identificación que se ha tenido en cuenta es la creación de un array donde guardar el identificador de los threads creados. Cada vez que se quiera conocer la cpu sobre la que nos encontramos, se deberá obtener el identificador del proceso donde estamos ejecutando el código (véase 2.1.2) y compararlo con el array, devolviendo el índice correspondiente. Este sistema nos puede ser útil en un entorno donde haya más procesos que procesadores reales y no se pueda realizar una asignación directa entre la cpu MaRTE OS y la cpu real.

Finalmente se decidió trabajar sobre la primera implementación, ya que se busca trabajar sobre un sistema multiprocesador lo más real posible y tener conocimiento real de donde se está ejecutando cada tarea.

p tu 3

Adaptación de MaRTE OS

El sistema operativo MaRTE OS está concebido para su ejecución en sistemas monoprocesador. Los sistemas multiprocesador se están extendiendo cada vez más, por lo que una conversión para que pueda aprovechar las características de un sistema multiprocesador es un hito importante en su desarrollo.

MaRTE OS es un sistema con un núcleo monolítico que utiliza la deshabilitación de las interrupciones para conseguir proteger las estructuras de datos internas del sistema operativo ante accesos concurrentes por parte de las distintas tareas. Este mecanismo ya no es suficiente en un sistema multiprocesador, puesto que, al existir concurrencia real, tareas o interrupciones ejecutadas en diferentes procesadores podrían acceder de forma simultánea a las estructuras del sistema

MaRTE OS es un sistema operativo orientado a ser ejecutado sobre plataformas x86 donde se trabaja directamente sobre el procesador y con

interrupciones. En la arquitectura linux_lib de MaRTE OS se emula el procesador y las interrupciones mediante modificaciones en la interfaz abstracta con el hardware para que utilice los mecanismos que proporciona Linux (hilos y señales) vistos en el capítulo anterior. En este capítulo se detallan las modificaciones sobre el núcleo de MaRTE OS, el cual está pensado para ser utilizado sobre cualquier arquitectura mediante su interfaz abstracta con el hardware. Por esta razón se ha optado po utilizar la nomenclatura de la interfaz abstracta con el hardware (procesadores y interrupciones) en vez de usar los mecanismos reales que se han utilizado (hilos y señales Linux).

En este capítulo se expone las modificaciones realizadas en MaRTE OS con el objetivo de aprovechar las características de los sistemas multiprocesador. Para ello se analiza la implementación de un nuevo sistema para proteger las secciones críticas. También se estudia los cambios realizados enfocados hacia la planificación del nuevo sistema y la inicialización necesaria para que todo se encuentre en el estado correcto al comenzar la aplicación.

3.1. Spinlock

El spinlock es un mecanismo de sincronización que limita el acceso a un recurso en un entorno donde hay más de una unidad de procesamiento. La característica principal de este mecanismo, respecto a otros, es que cuando el acceso al recurso se encuentra bloqueado el proceso se introduce en un lazo en el que comprueba el estado del recurso repetidamente. El proceso permanece activo y utilizando tiempo de procesamiento en la comprobación, sin embargo no está realizando ninguna tarea útil. A este tipo de mecanismos se los denomina de *espera activa*. Este mecanismo es utilizado cuando se estima que el tiempo de espera va a ser inferior al tiempo de los dos cambios de contexto que se producirían en el caso de dormir el proceso y planificar otro de inferior prioridad, ya que este segundo proceso apenas va a tener tiempo de realizar su trabajo antes de volver a ser expulsado. En caso contrario, significa estar consumiendo recursos que pueden ser utilizados para realizar otras tareas.

El mecanismo se basa en realizar una operación, normalmente una substracción (o un *test and set*), de forma atómica, es decir, la operación no puede ser interrumpida ni dividida y es vista como instantánea; y comprobar las banderas generadas por dicha operación. Generalmente estas instrucciones se ejecutan en un único ciclo de procesador, sin embargo, en un sistema multiprocesador se necesita de algún mecanismo adicional. Este lo propor-

Cuadro 3.1: *Spinlock: Pseudocódigo de las funciones lock y unlock.*

Lock	Unlock
`atomic (decremento &lock);`	
`si 0 -> FIN (flags de op anterior)`	
`compara 0 y &lock`	`&lock = 1;`
`repite si menor o igual a 0`	
`vuelve al principio`	

ciona el sufijo `LOCK` precediendo a la instrucción ensamblador. Dicho sufijo provoca que se aserte la señal `#LOCK` del procesador donde se está ejecutando la instrucción que se pretende se ejecute de forma atómica. La señal `#LOCK` asegura que mientras se encuentre asertada el procesador tiene uso exclusivo de cualquier memoria compartida [8, sección 7.1], impidiendo que las tareas del resto de procesadores puedan hacer uso de las estructuras del sistema de forma simultánea. Esta señal solo funciona con un conjunto reducido de instrucciones, la substracción se encuentra entre ellas.

El funcionamiento es sencillo (ver cuadro 3.1). Se tiene una variable que ha de ser inicializada al valor '1'. Es importante inicializar siempre el valor del spinlock. Cuando se quiere entrar en una sección protegida con dicho spinlock se decrementa su valor de forma atómica en una unidad y, mediante las banderas del procesador, se comprueba si el valor de la variable es cero. Si dicho valor es cero se permitirá continuar con la sección protegida, en cuyo final se restaurará el valor de la variable del spinlock. En el caso de que al decrementar el valor de la variable, éste no fuera cero, significa que hay algún proceso ejecutando código en una sección protegida, por lo que el programa se introduce en un lazo en el que constantemente esta comprobando el valor de la variable. Cuando se detecta que la variable toma un valor positivo, se sale del lazo volviendo al comienzo. Si al decrementar obtiene el resultado de cero pasará a ejecutar su sección protegida, en caso contrario, es que otro proceso se ha adelantado y volverá al lazo a la espera de que se vuelva a liberar el spinlock. Después de ejecutar la sección protegida correspondiente, el programa desbloquea el spinlock fijando nuevamente el valor de la variable a '1'.

Las banderas del procesador utilizadas son `ZF` (*Zero Flag*) y `SF` (*Sign Flag*), la primera se aserta cuando el resultado es nulo y la segunda toma el valor del bit más significativo del resultado, que es el bit de signo del tipo entero

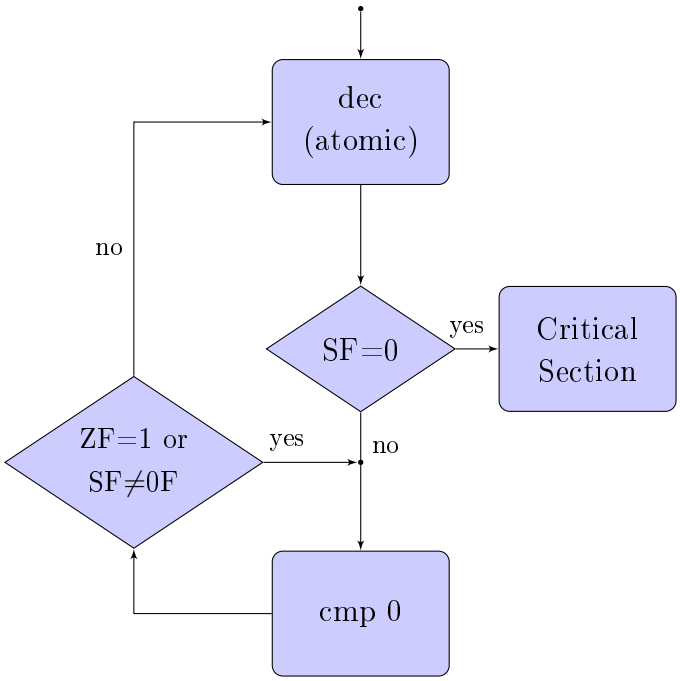

Figura 3.1: *Diagrama de flujo de la función que bloquea el spinlock*

con signo (está asertado con valores negativos). Se puede ver el pseudocódigo de las funciones lock y unlock en el cuadro 3.1 y el diagrama de flujo de la función lock en la figura 3.1.

La primera comprobación del valor del spinlock se realiza sobre el valor de la bandera de signo (SF, indica el signo resultante de la última operación) como resultado de la operación de decrementar. Si se implementara como otra instrucción, podría producirse alguna interrupción o que otro procesador esté ejecutando las mismas instrucciones para entrar en una sección protegida, por lo que no se ejecutaría de forma atómica y se produzcan inconsistencias.

Cuadro 3.2: *Spinlock: ficheros y su localización.*

Fichero	Localización
marte_spinlock.h	linux_lib_arch/include/misc/
marte_spinlock_c.c	linux_lib_arch/libmc/
marte_spinlock.ads	misc/

Para facilitar su utilización en MaRTE OS se ha implementado una librería que se basa en la implementación que realiza Linux. Se genera un fichero de cabecera (.h) donde se definen un conjunto de macros y tipos en lenguaje ensamblador para ser utilizado de forma estática. Para la interfaz en Ada se han tenido que transformar las macros en funciones con el objetivo de que puedan ser importadas. La localización de dichos ficheros en MaRTE OS se

Cuadro 3.3: *Spinlock: interfaz C.*

```
#include <misc/marte_spinlock.h>
typedef struct marte_spinlock_t;
static inline
 int  marte_raw_spin_is_locked (raw_spinlock_t *l);
static inline
 void marte_raw_spin_lock (raw_spinlock_t *lock);
static inline
 void marte_raw_spin_unlock (raw_spinlock_t *lock);
static inline
 void marte_raw_spin_init (raw_spinlock_t *lock);
```

muestra en el cuadro 3.2.

El fichero `marte_spinlock.h` se muestra en el cuadro 3.3. Para su uso en primer lugar se define la variable que se va a utilizar como spinlock (de tipo `marte_spinlock_t`) y que debe ser inicializado antes de ser utilizada (con `marte_raw_spin_init`). Posteriormente será tomado y liberado el spinlock de forma ordenada. Hay que tener especial cuidado en tomar y liberar el lock siempre de forma correcta, pues se corre el peligro de que la aplicación quede bloqueada. Se provee también una función adicional que devuelve el valor del spinlock (`marte_raw_spin_is_locked`).

3.2. Cola de area ejec able

En la versión monoprocesador de MaRTE OS la tarea que se encuentra en ejecución en un momento dado se identifica a través de la variable `Running_Task`, así mismo, todas las tareas que se encuentran en estado activo (y que incluye a la tarea que se encuentra en ejecución) se encuentran ordenadas por prioridad en una cola denominada `Ready_Queue`. La primera tarea de la cola se corresponde con la tarea más prioritaria y, normalmente, también con la tarea que se encuentra en ejecución. Para decidir si debe realizarse un cambio de contexto el planificador únicamente ha de comparar ambas tareas. En caso de que la tarea en ejecución sea diferente de la que se encuentra en la cabeza de la cola se ha de retirar a la tarea que se encuentra en ejecución y situar a la nueva tarea más prioritaria. Las tareas únicamente abandonan la cola

cuando dejan de estar activas, y por lo tanto, dejan de ser planificables. Por otro lado se define una tarea, denominada *idle* y que no realiza trabajo útil, que se guarda en la cola en una posición reservada para ella, de forma que cuando no haya tareas ejecutables sea esta tarea la que ocupe el procesador.

En esta modificación de MaRTE OS para SMP el objetivo es que MaRTE OS tenga tantas tareas en ejecución como procesadores disponga el sistema realizando los menos cambios posibles en su estructura. Para conocer de forma rápida cuales son las tareas en ejecución se ha implementado un array con tantos identificadores de tarea como procesadores se vayan a tener. Debido a esta modificación se ha tenido que modificar la filosofía de planificación anterior, en la que la tarea que se encontraba primera en la cola es la que tiene que ejecutarse. En la nueva versión las tareas que se encuentran en ejecución se van a extraer de la cola y se guardarán en el array. Mantener estas tareas en la cola dificultaría enormemente la planificación debido a la dificultad de localizar a la tarea que opta a ser ejecutada. Esta implementación nos proporciona inmediatez a la hora de identificar las tareas que se están ejecutando y cual es su posible reemplazo, sin embargo, al ser varias tareas, el algoritmo para comprobar si ha de realizarse algún cambio de contexto va a ser algo más complejo (véase la sección 3.3).

Nuevamente, al haber varias tareas ejecutándose simultáneamente, se debe tener en cuenta cual es la tarea que ha llamado al planificador, ya que es sobre la que habrá que realizar las acciones oportunas. En la versión monoprocesador de MaRTE OS se identifica de forma única a la tarea que se encuentra en ejecución. Para mantener la estructura, en todos los procedimientos donde se hace referencia a la tarea en ejecución se toma la tarea correspondiente al procesador donde se está ejecutando el procedimiento. Esto se consigue identificando el procesador donde se está ejecutando el planificador, que ha de ser el mismo que el de la función que ha provocado su ejecución (ver sección 2.4).

Respecto a las tareas idle pasa algo parecido. Se dedican a iterar en un lazo que no realiza ninguna operación y se ha optado por utilizar un mecanismo similar al de las tareas en ejecución. Se implementa un array donde se guardan los identificadores de las diferentes tareas idle que van a ser utilizadas y cuando llega el momento de planificar se selecciona la tarea correspondiente al procesador en el que se está ejecutando el planificador.

En la figura 3.2 se observa un esquema de los dos arrays de tareas y de la cola de tareas listas para ser ejecutadas. La tarea idle de la cola de tareas se corresponde con una de las tareas que se encuentran almacenadas en el

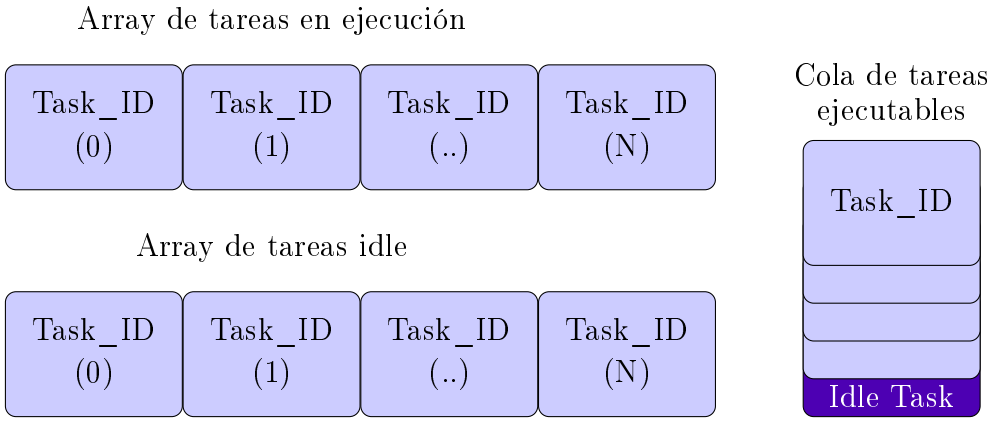

Figura 3.2: *Tareas en ejecución y preparadas para ser ejecutadas.*

array de tareas idle. Esta tarea es siempre la última en la cola y se utiliza para que el planificador detecte que no existen más tareas activas en la cola y planifique la tarea idle correspondiente al procesador en el que se encuentre.

3.3. Algoritmo de planificación

MaRTE OS ordena las tareas que se encuentran preparadas para ser ejecutadas en una estructura compuesta por un array colas. Cada cola del array hace referencia a una prioridad diferente. Las tareas se almacenan dentro de la cola de su prioridad en orden FIFO y mediante un mapa de bits se detecta de forma rápida cual es la cola más prioritaria que contiene tareas. Desde el punto de vista del sistema operativo, dicha estructura es tratada como si fuera una única cola de tareas ordenadas por prioridad, de forma que cuando se comprueba si es necesario realizar un cambio de contexto, se extrae la información de la tarea más prioritaria de la estructura y se compara con la actual tarea que se encuentra en ejecución. Normalmente, la tarea más prioritaria se corresponde con la tarea que se está ejecutando, y en caso contrario es porque se requiere un cambio de contexto. Mientras la tarea permanezca activa no se extrae de la cola, como se ha visto en la sección 3.2.

Como se ha comentado anteriormente, en la implementación multiprocesador de MaRTE OS se utiliza un array para almacenar las tareas que se encuentren en ejecución. Al haber varias tareas en ejecución de forma simultánea la cabeza de la cola no es un referente para decidir si se ha de realizar el cambio de contexto. Para resolver esta problemática se extraen de la cola las tareas que se encuentran en ejecución. Ahora la tarea que se

encuentra como cabeza de la cola es la tarea más prioritaria que opta a ser planificada.

El algoritmo de planificación que se utiliza en este trabajo ha sido escogido debido a su simplicidad y a su parecido con el algoritmo de la versión monoprocesador de MaRTE OS, por lo que no ha sido necesario realizar grandes cambios conceptuales de su estructura. El algoritmo utiliza la prioridad de las tareas para ordenarlas en una única cola global. Las tareas van a poder ser ejecutadas en cualquiera de los procesadores. El algoritmo comprueba el estado de las tareas en ejecución para decidir si debe realizar un cambio de contexto, buscando si alguna se encuentra suspendida y/o bloqueada, si la tarea que se encuentra en la cola posee mayor prioridad que alguna de las que se encuentran en ejecución o si algún procesador no tiene una tarea activa asignada para ejecutar. Si la tarea que debe ser expulsada se encuentra en un procesador diferente del que se está ejecutando el planificador, el algoritmo envía una interrupción a dicho procesador para que realice el cambio de contexto.

El planificador puede ser llamado por múltiples motivos que no le son indicados, por lo que su cometido es descubrir qué tarea puede planificar y en qué procesador. De esto se encarga el procedimiento cuyo diagrama de flujo se muestra en la figura 3.3. Dicho algoritmo en primer lugar inicializa las variables que utiliza. Da valor a la variable `cpuid` con el identificador del procesador sobre el que se está ejecutando e inicializa a falso la variable `IPIPend` correspondiente al procesador para indicar que la interrupción, si la ha habido, ya está atendida y la variable `CSReq` que indica si es necesario un cambio de contexto local. Se inicializa a falso también la variable `IPIReq` que señaliza cuando se ha de realizar el envío de una interrupción a otro procesador, el envío se realiza una vez analizado el sistema completo.

Después de la inicialización se procede a comprobar si es necesario realizar un cambio de contexto y en qué procesador. El procedimiento `Search_Target_CPU`, cuyo funcionamiento se describe en detalle más adelante, se encarga de analizar las tareas que se encuentran en ejecución en todos los procesadores y comparar su prioridad con la primera tarea de la cola. Si encuentra una situación en la que se deba producir un cambio de contexto, indica el procesador sobre el que debe ser realizado. Si, por el contrario, no encuentra dicha situación devuelve el valor `-1`, de esta forma se indica que el estado de los procesadores es el idóneo, por lo que no hace falta actuar y finaliza (no va a realizar más comprobaciones). En el caso de que indique un procesador se comprueba si se trata del procesador local o de uno remoto. En el caso de ser un procesador remoto se indica que ha de ser enviada una interrupción

para que se realice al final.

Cuando el cambio de contexto es local hay que tener en cuenta que debido al filtro implementado es posible que alguna interrupción haya sido filtrada (que se hayan producido varios eventos y todos intenten planificarse en el mismo procesador), pudiendo darse el caso que la segunda tarea de la cola debiera ser planificada también. Por ello se marca la variable que indica que se debe realizar el cambio de contexto local, se extrae la cabeza de la cola y se vuelve a realizar, en caso de que la nueva tarea no sea del tipo idle, una búsqueda para comprobar si tiene más prioridad que alguna de las tareas que se encuentran en ejecución. Si la búsqueda es positiva, se marca la variable que indica que ha de ser enviada la señal (`IPIReq`).

Internamente el procedimiento `Search_Target_CPU` comprueba en primer lugar si la tarea que se encuentra en ejecución en el procesador local se encuentra en el estado activo, o, por contra, se encuentra suspendida o bloqueada y debe abandonar el procesador. También se comprueba si, siendo la tarea que se encuentra en ejecución una tarea de tipo idle, la primera tarea de la cola no lo es y por lo tanto ha de ejecutarse. En ambas situaciones la función retorna el identificador del procesador local para indicar un cambio de contexto local. Si esta situación de cambio de contexto directo no se produce, se ha de analizar las tareas de todos los procesadores para comprobar si alguna de ellas es de tipo idle o tiene una prioridad menor que la tarea que se encuentra en la cola. En el caso de encontrarse alguna tarea de tipo idle, se para la búsqueda y se indica ese procesador para planificar. Si, por el contrario, se encuentra alguna tarea con menor prioridad que la de la cola, se indica el número de procesador de la menor de todas y en caso de igualdad, aquella que está en el procesador con un identificador más bajo (primera según el orden de la búsqueda). En el caso de la segunda llamada que realiza el algoritmo de la figura 3.3 al procedimiento se omiten las operaciones sobre el procesador local, puesto que va a ser ocupado por una tarea que posee mayor prioridad, ya que se encontraba por delante en la cola.

Para finalizar, si la variable `IPIReq` fue marcada en el transcurso de la búsqueda y se comprueba de que el procesador indicado en `targetcpu`, que es el que deberá recibir la interrupción, no tiene ninguna pendiente, se procede al envío de una interrupción al procesador indicado en la variable `targetcpu`. En el caso de tener interrupciones pendientes, se ignora y no se envía. Será el planificador el que detecte si se producen varios casos de planificación.

Al enviar una interrupción al procesador correspondiente (`targetcpu`), ese procesador verá interrumpido lo que está realizando y planificará la tarea de

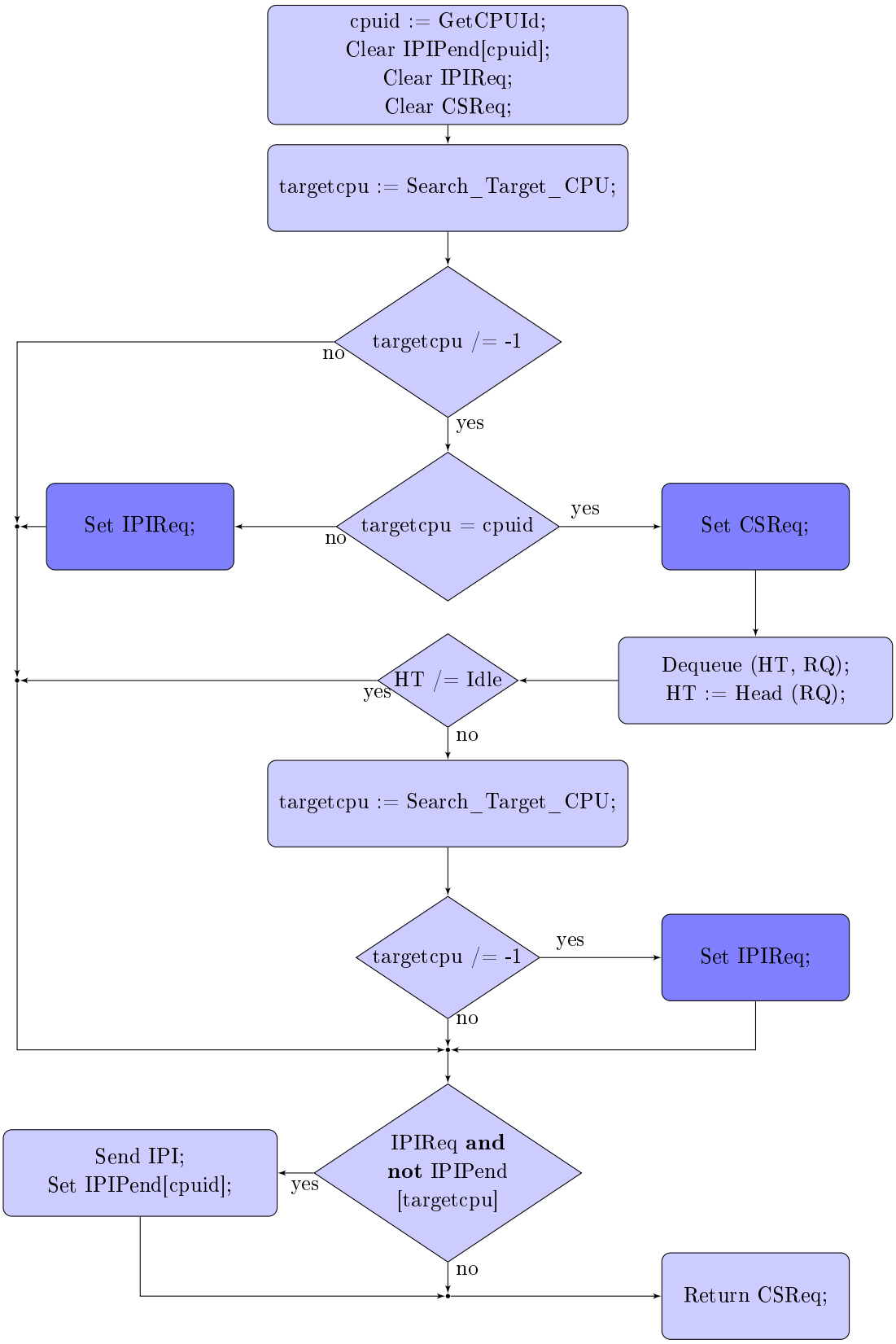

Figura 3.3: *Diagrama de flujo del algoritmo que decide el cambio de contexto.*

mayor prioridad que se encuentra esperando en la cola. Al enviar la interrupción no se envía ninguna información adicional para que el procesador sepa lo que tiene que hacer, por lo que el procesador que recibe dicha interrupción específica volverá a ejecutar el algoritmo de planificación completo (tal y como se muestra en la figura 3.3) y si no se ha producido ningún cambio en el estado del sistema planificará la tarea que se encuentra en la cabeza de la cola. Recordar que aunque en MaRTE OS se habla de interrupciones, éstas se emulan mediante señales Linux en la arquitectura linux_lib.

Al volver a ejecutar la planificación, este procesador puede detectar que otro procesador se encuentra ocioso, o que existe algún procesador ejecutando alguna tarea de menor prioridad de la que va a ser planificada, por lo que podrá enviar una nueva interrupción a otro procesador.

Para que los procesadores no envíen múltiples interrupciones a un mismo procesador antes de que sea atendida, y puesto que muy probablemente el motivo del envío múltiple de las interrupciones sea el mismo, se ha implementado un filtro para que hasta que no sea atendida una interrupción no puedan serle enviadas más. El filtro es implementado a través de un array booleano (`IPIPend`), con un elemento por cada procesador, donde se marca aquellos procesadores a los que ya se les ha enviado la interrupción. Esta marca permanecerá activa hasta que sea atendida por el manejador. Cuando el planificador detecta que el procesador objetivo tiene una señal pendiente, desecha la interrupción que iba a enviar.

En la figura 3.3, que muestra el diagrama de flujo del algoritmo, se encuentra resaltado los momentos en los que se indica que se va a realizar un cambio de contexto local (`Set CSReq;`) y donde se realiza la indicación para que se envíe la interrupción que va a provocar la planificación en el procesador escogido (`Set IPIReq;`).

3.4. Con r cci n de eccione cri ica

Con una única unidad de procesamiento la inhibición de las interrupciones es suficiente para tener la seguridad de que la sección critica no va a ser interrumpida por algún evento u otra tarea, y así es como se encuentra implementado en MaRTE OS. En un sistema multiprocesador lo normal será que cada procesador gestione de forma independiente sus interrupciones. Así por ejemplo, en los multiprocesadores de Intel cada procesador posee una APIC encargada de la gestión de las interrupciones para ese procesador. És-

tas están comunicadas con una APIC para entrada/salida. Lo mismo ocurre en nuestra implementación sobre Linux con la máscara de señales.

Esto obliga a implementar un mecanismo para que el kernel de MaRTE OS no se ejecute de forma simultanea en dos procesadores diferentes, puesto que podría derivar en una inconsistencia grave en las variables que se manipulan en el kernel durante la planificación.

La implementación seleccionada para impedir el acceso simultaneo a las secciones críticas del kernel de MaRTE OS ha sido una combinación de deshabilitación de las interrupciones con spinlocks. La deshabilitación de interrupciones (bloqueo de señales) evita que una tarea ejecutando una sección crítica sea expulsada por un evento externo que debe atenderse en ese mismo procesador. Como se ha visto anteriormente, las interrupciones únicamente se deshabilitan en el procesador en el que se ejecuta la instrucción, por lo que , dado el caso de que no sean especificas, pueden llegar a ser atendidas por otro procesador.

Por otro lado, la utilización de los spinlock es debida a que estando ejecutando parte del kernel no es recomendable realizar suspensiones y/o cambios de contexto, pues la idea es entrar y salir de la sección crítica lo antes posible. Cosa que permite la espera activa.

En MaRTE OS se define como sección crítica del kernel al trozo de código que se ejecuta desde que se deshabilitan las interrupciones y se toma el spinlock hasta que finaliza liberando el spinlock y habilitando de nuevo las interrupciones. Las instrucciones dentro de la sección crítica pueden acceder de forma exclusiva a las estructuras de datos del kernel. En general son todas las operaciones relacionadas con la gestión de las tareas, y cuyo máximo exponente es el procedimiento `Do_Scheduling` que se encarga de comprobar si es necesario realizar un cambio de contexto para ejecutar una tarea más prioritaria. Por lo tanto todas estás operaciones van protegidas como se ha explicado anteriormente.

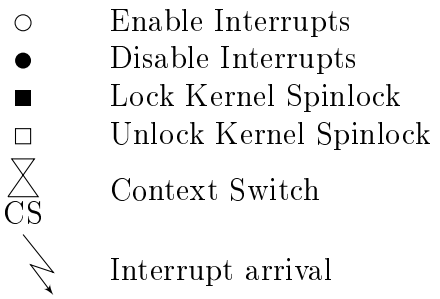

Figura 3.4: *Sección crítica: Leyenda.*

A la hora de trabajar con secciones críticas y con bloqueos hay que tener mucho cuidado con las situaciones que se pueden generar, ya que un error en

su uso puede provocar el bloqueo del programa. A continuación se muestra el estudio realizado de todas las posibles situaciones que se pueden producir en MaRTE OS en su versión multiprocesador. En la figura 3.4 se muestra una leyenda con el significado de los iconos que son utilizados en los diagramas posteriores.

El cambio de contexto es el elemento principal de la planificación, se produce cuando la tarea en ejecución cede su puesto en el procesador, de forma voluntaria u obligada, a otra tarea. Durante el cambio de contexto se guarda el estado actual del procesador en el stack de la tarea correspondiente, para en un futuro poder restaurar dicho estado y poder continuar el trabajo de la tarea justo donde se dejó; y es sustituido por el estado de la nueva tarea que va a ser ejecutada. Esta operación se llama desde dentro de la sección critica del kernel, pues una interrupción o corrupción de los datos puede ser catastrófica. Sin embargo, una vez que se produce el cambio de contexto, el kernel deja de tener el control del procesador, que es cedido a la tarea. Puesto que salir de la sección crítica antes de realizar el cambio de contexto no es viable, se ha de dejar que la nueva tarea realice la liberación de la sección crítica después del cambio de contexto, es decir, la salida de la sección crítica se realiza en el contexto de la nueva tarea.

Un cambio de contexto básico es el que se muestra en la figura 3.5 donde una tarea, `T1`, se encuentra ejecutando en el procesador. En un momento dado, el código de la tarea ejecuta, por ejemplo, una operación que la suspende (como puede ser la operación `nanosleep`). La operación de suspensión realiza operaciones en el kernel de MaRTE OS, llama a la operación de entrada en la sección crítica denominada `Enter_Critic_Section` (`ECS` en el diagrama) que es la que se encarga de enmascarar las señales (deshabilitar interrupciones) y tomar el spinlock. Una vez dentro de la sección crítica se realizan todas las instrucciones correspondientes a la operación que ha sido llamada. Si la operación conlleva un cambio de estado de alguna tarea al finalizar se llama a la operación de planificación `Do_Scheduling` (`DoSched`). Como se ha visto en la sección anterior `Do_Scheduling` se encarga de comprobar si es necesario realizar el cambio de contexto. Si este se realiza, es el contexto de la nueva tarea, `T2`, el que se encarga de salir de la sección crítica mediante la llamada a `Leave_Critic_Section` (`LCS`) y continua con su ejecución.

Figura 3.5: *Sección crítica: Cambio de contexto básico.*

Para que todo funcione correctamente es necesario que las tareas recién creadas estén envueltas en una pequeña porción de código que se encarga de que cuando se planifica por primera vez la tarea ésta libere la sección crítica del kernel antes de comenzar a ejecutarse. Al tratarse de la primera vez que se ejecuta la liberación de la sección crítica ha de realizarse mediante un procedimiento ligeramente diferente llamado `Leave_Critic_Section_From_Start_Task` (`LCSfST`). Así mismo, una vez finalizada una tarea se ha de liberar las estructuras de control que ha utilizado e indicar al kernel que existe un procesador libre. Estas operaciones se realizan, nuevamente, desde la envolvente, donde se entra en la sección crítica y se realizan los procesos de liberación de recursos (`Terminate_Running_Task`) que finaliza llamando al procedimiento de planificación. En la figura 3.6 se muestra como al comenzar la ejecución de una tarea se libera la sección crítica y al finalizar se vuelve a tomar para ejecutar la liberación de recursos.

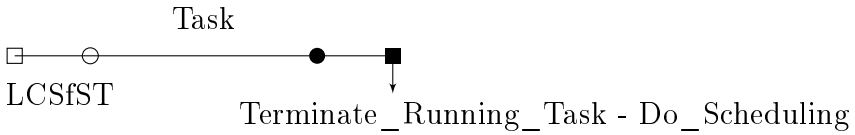

Figura 3.6: *Sección crítica: Envolvente de tarea.*

Si el sistema no tuviera eventos, las tareas se ejecutarían de forma ordenada y esperando a que finalizase la tarea anterior. De esta forma, al finalizar la tarea, ésta toma la sección crítica y el planificador se encarga de poner a otra en ejecución, que va a liberar la sección crítica para ejecutarse sin problemas. Se ejecutarían de forma cíclica y ordenada tareas y secciones críticas.

Sin embargo, la realidad no es tan sencilla, los cambios de contexto se pueden agrupar en dos grupos:

- Los cambios de contexto locales son aquellos que se producen en el mismo procesador donde se ha invocado el planificador al producirse algún evento, como pueden ser la suspensión de una tarea, la recepción de una interrupción, etc.
- Los cambios de contexto remotos son aquellos en los que el planificador envía una interrupción a otro procesador indicándole que existen tareas más prioritarias que la que actualmente está ejecutando. Dicho procesador realizará un cambio de contexto local.

Un cambio de contexto local se produce por ejemplo cuando la tarea en ejecución realiza una operación que requiera una llamada al planificador

(cambio de prioridad, cesión voluntaria, suspensión, etc.). Esta tarea realizará una llamada a las operaciones correspondientes del planificador, y por lo tanto, requerirá la entrada en una sección crítica. El planificador en función de la operación escogida y las tareas que se encuentran en la cola de preparadas escoge si mantener a la tarea en ejecución porque es más prioritaria que las que se encuentran en la cola, o por el contrario, la tarea que se encuentra en la cola es más prioritaria que la tarea que se encuentra en ejecución y por lo tanto se planifica. Esta tarea puede haberse ejecutado con anterioridad, por lo que continuará desde donde fue expulsada o puede ser una tarea recién creada y que todavía no se ha ejecutado, por lo que comenzará por ejecutar el código envolvente. En la figura 3.7 se observan los tres posibles casos de planificación local y ordenada. Los diagramas muestran respectivamente una situación sin cambio de contexto, un cambio de contexto a una tarea nueva, y un cambio de contexto a una tarea iniciada.

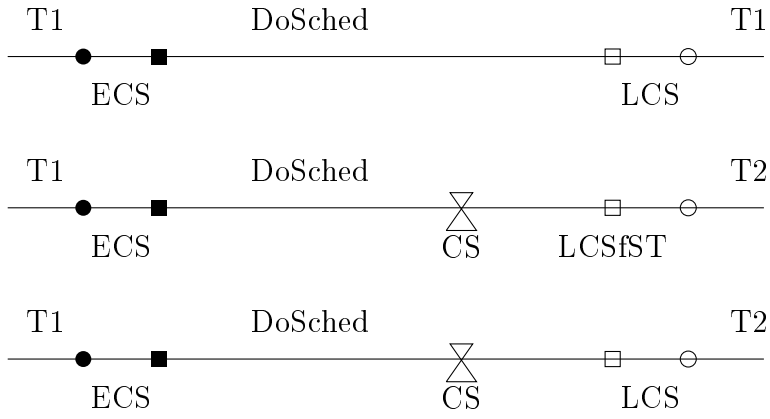

Figura 3.7: *Sección crítica: Planificación local y ordenada.*

Aunque una de las posibles operaciones para que una tarea deje el procesador es `yield`, en la que la tarea cede el procesador a otras posibles tareas de igual prioridad; lo normal es que la tarea se suspenda por un tiempo. El sistema operativo programa un temporizador para el instante de activación que lanzará una interrupción al expirar. El manejador de dicha interrupción activará la tarea e indicará al kernel que hay una tarea que se despierta para que revise si ha de hacerse un cambio de contexto.

Cuando una tarea ejecuta una operación de suspensión que hace uso de la sección crítica del kernel, dicha operación se encuentra envuelta por la toma y liberación del spinlock del kernel. Sin embargo, si la tarea es expulsada por dicha operación no puede realizar la liberación del spinlock. Es la tarea que la reemplaza la que lo va a liberar, ya sea mediante su envolvente, como se ha visto anteriormente, o porque al completar la operación por la que fue

expulsada se libera el spinlock.

En el momento en el que se produce una interrupción, estas son deshabilitadas por hardware en la arquitectura x86, sin embargo, en las arquitecturas linux y linux_lib se han de deshabilitar mediante software a través del manejador de interrupciones de MaRTE OS. Al igual que las tareas, el manejador de interrupción (`IH` en el diagrama) definido por el usuario es envuelto por un conjunto de instrucciones que permiten al kernel gestionar diferentes aspectos de la interrupción, siendo uno de ellos el acceso a la sección crítica. Las interrupciones son pequeños trozos de código que se ejecutan con alta prioridad en atención a eventos importantes del sistema. Puesto que los manejadores de los eventos suelen interactuar con el kernel las interrupciones se ejecutan dentro de una sección crítica. Al finalizar el manejador de la interrupción, ésta puede finalizar de dos formas. Por un lado, si se ha producido alguna llamada al planificador ésta no ha sido atendida por encontrarse ejecutando el manejador, por lo que se llama al planificador al finalizar la interrupción (`Do_Scheduling`). En el caso de no necesitarse el planificador se finaliza la interrupción (`End_Interrupt`) y se continua con la tarea que fue interrumpida. En el diagrama de la figura 3.8 se puede ver lo comentado.

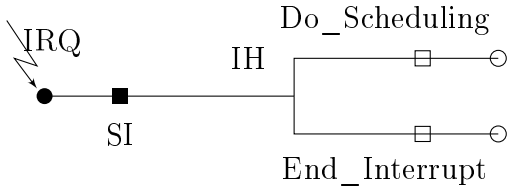

Figura 3.8: *Sección crítica: Manejador de interrupción.*

Al finalizar el manejador de la interrupción se libera el spinlock del kernel y se habilitan de nuevo las interrupciones. Cuando la interrupción finaliza de forma normal la tarea continua ejecutando. Sin embargo, cuando se produce la llamada al planificador y se realiza un cambio de contexto, aunque la interrupción se da por concluida todavía no ha salido de su envolvente y es la tarea que toma el procesador la que realiza la liberación de la sección crítica. La tarea que fue interrumpida se encuentra bloqueada en la envolvente de la interrupción y cuando sea ejecutada nuevamente finalizará las operaciones que quedan, que son la liberación del spinlock y la habilitación de interrupciones. Estas operaciones permitirán liberar los recursos del kernel que tomó la tarea a la que sustituye en el procesador. Se explica con más detalle en la figura 3.10 donde se muestra el proceso completo.

En la figura 3.9 se muestra diferentes formas de actuar del sistema ante la llegada de una interrupción. La llegada de una interrupción interrumpe la

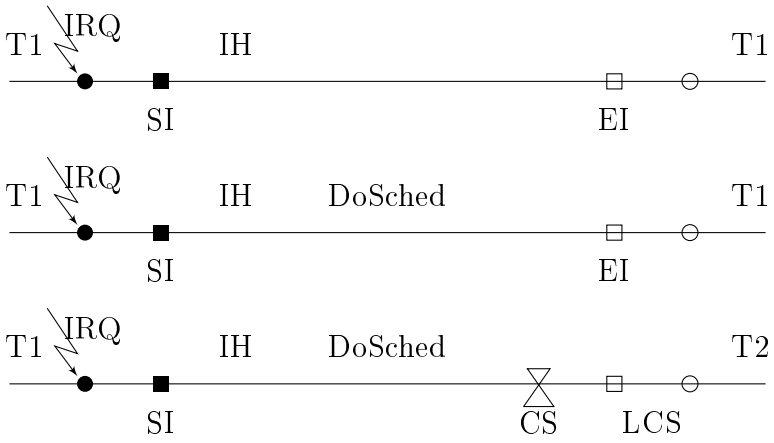

Figura 3.9: *Sección crítica: Llegada de una interrupción.*

ejecución de la tarea `T1` generando un entorno de sección crítica (`SI`, `Start_Interrupt`) y ejecutando el manejador de la interrupción. A la hora de finalizar la interrupción es cuando se observan las diferencias. Como se ha comentado anteriormente, si no se requiere la ejecución del planificador éste no se ejecuta y se finaliza la interrupción (`EI`, `End_Interrupt`) continuando la ejecución de `T1`. Si se requiere planificación existe el caso de que no se requiera realizar el cambio de contexto y se puede continuar con la tarea `T1`, o que sea requerido y se cambie el contexto a la tarea `T2`. La tarea `T2` puede ser nueva o haber ejecutado ya parte de su código, se comportan prácticamente igual como ya se ha visto anteriormente.

Como ejemplo de una situación algo más compleja en la figura 3.10 se muestra el escenario completo en el que una tarea, T1, es interrumpida por la llegada de una interrupción que provoca que tenga que ser ejecutado su manejador, IH (*Interrupt Handler*), y que el estado del sistema cambie. Al existir una tarea más prioritaria (T2) que la que se encontraba ejecutándose se produce el cambio de contexto correspondiente. Al cabo de un tiempo T2 finaliza las operaciones que debe realizar y abandona el procesador, dejándolo libre a la siguiente tarea de la cola, en este caso suponemos que vuelve a ser T1, que realiza los últimos procedimientos de salida del manejador de interrupción y continua ejecutando el código escrito por el usuario.

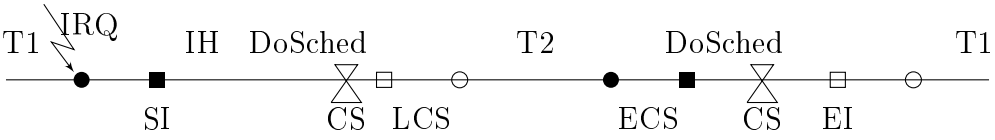

Figura 3.10: *Sección crítica: Proceso completo de la llegada de una interrupción.*

Aunque la interrupción como tal finaliza antes de ejecutarse el primer cambio de contexto, el puntero del contador de programa perteneciente a la tarea expulsada (`T1`) permanece apuntando a funciones que se encuentran dentro del procedimiento envolvente del manejador de interrupción, y aunque lo único que va a hacer es ir saliendo de la función de planificación, de la envolvente de la interrupción, y salir de la sección critica, el código pertenece a la interrupción. En la figura 3.10 se aprecia como el código que finaliza la interrupción (`EI`) se ejecuta al volver a planificar la tarea que fue expulsada

También se debe considerar que son varios procesadores los que van a estar ejecutando instrucciones, lo que supone que podrán darse cambio de contexto remotos. En este tipo de cambios de contexto, como se ha visto anteriormente, los procesadores se van a comunicar mediante interrupciones (señales en la arquitectura linux_lib). La comunicación entre los procesadores se limita a enviar una interrupción cuando es la tarea del otro procesador la que debe ser expulsada. Esta interrupción provoca que se vuelva a ejecutar el planificador en el otro procesador. Éste comprobará que existe una tarea en la cola que tiene mayor prioridad que la que está ejecutando actualmente y procederá al cambio de contexto. El cambio de contexto es un cambio de contexto local, por lo que no hay variaciones con lo ya visto. Lo único que cambia es la forma de invocar al planificador, a través de una interrupción específica.

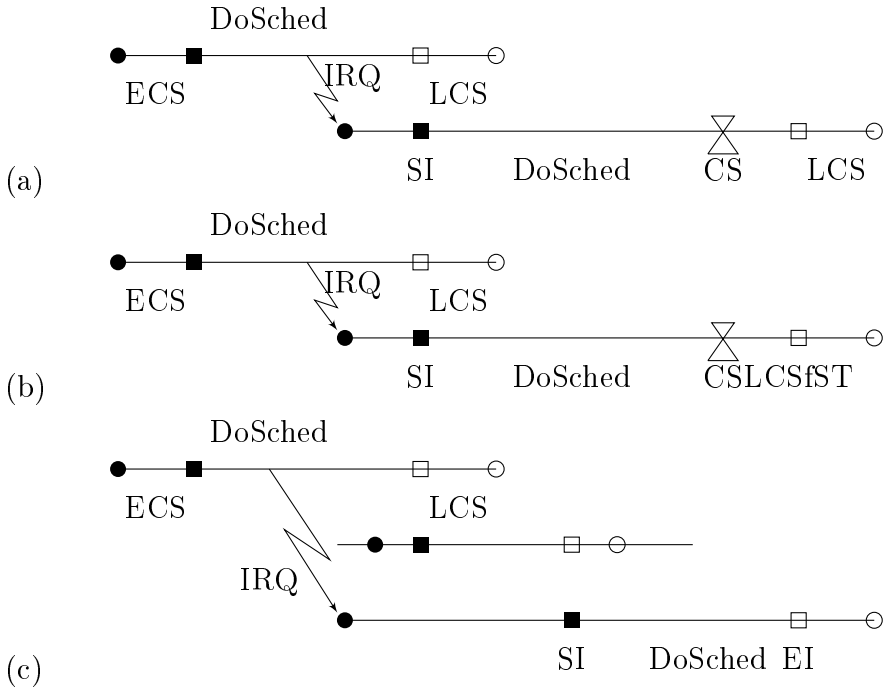

Figura 3.11: *Sección crítica: Interrupción entre procesadores.*

En el proceso de planificación, una vez que se ha comprobado que no existe necesidad de un cambio de contexto local, el algoritmo comprueba si existe alguna tarea entre las que se encuentran en ejecución que posea una prioridad menor que la que se encuentra en la cola. Si se da el caso, el planificador envía una interrupción al procesador correspondiente. En la figura 3.11 se muestran tres posibles situaciones, donde la más común va a ser la (a), en la que el procesador inicial descubre que hay que realizar un cambio de contexto en otro procesador y le envía una interrupción. El diagrama (b) muestra el caso de que la tarea a ejecutar sea nueva. El diagrama (c) sin embargo, es algo más complejo. Muestra una situación en la que se ejecuta otra sección crítica en un tercer procesador en el intervalo desde que se realiza el envío de la interrupción hasta que se entra en la sección crítica del planificador en el procesador remoto. Ésta sección crítica modifica el estado de las tareas involucradas en el cambio de contexto señalizado (cambio de la prioridad de alguna de ellas, cesión del procesador, la tarea que estaba esperando para a ejecutarse, etc.), provocando que cuando se ejecute el planificador en el procesador remoto, el cambio de contexto ya no sea necesario. Esta situación se produce por una condición de carrera para entrar a la sección crítica.

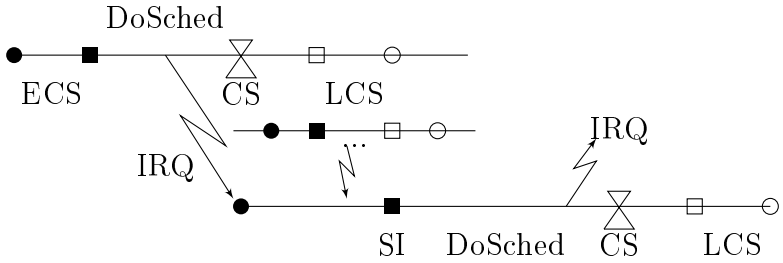

Figura 3.12: *Sección crítica: Doble cambio de contexto.*

Al igual que en el diagrama (c) de la figura 3.11 se puede dar el caso que entre se que envía la señal y esta es atendida se puedan producir otras entradas en la sección crítica. Debido al filtro implementado puede darse el caso de que en dos procesadores diferentes se produzcan eventos de planificación y ambos se remitan a al mismo procesador, pues es el que tiene la tarea de prioridad más baja. Este procesador recibirá una única interrupción debido al filtro y realizará el cambio de contexto correspondiente. Suponiendo que estos eventos sean de tareas que se despiertan, la segunda tarea (de menor prioridad) se queda sin planificar. Es por ello, como se explica en la sección 3.3, que cuando se planifica una tarea en el procesador local se revisa el estado de la cola por si fuera necesario realizar un nuevo cambio de contexto en otro procesador. Esta situación se muestra en la figura 3.12.

3.5. Inicialización

La creación de los hilos Linux y los spinlock que utiliza el kernel de MaRTE OS se hace necesaria antes de la inicialización de las tareas de usuario. En MaRTE OS estas operaciones se realizan en el paquete `marte-kernel-initialization.adb`.

Por un lado se ha de inicializar el spinlock del kernel, esto se realiza al comienzo del paquete que confecciona la inicialización del sistema para evitar problemas en su posible utilización.

Por otro lado, al finalizar la inicialización normal de MaRTE OS para sistemas monoprocesador y una vez instalados los manejadores y desenmascaradas las señales (puesto que la inicialización se realiza con éstas enmascaradas), se inicializa el nuevo sistema multiprocesador de MaRTE OS para linux_lib. La inicialización multiprocesador consiste en la asignación exclusiva del hilo principal al procesador en el que se esté ejecutando para posteriormente crear el resto de los hilos necesarios que serán utilizados como procesadores. Tras crear cada hilo, con las características indicadas en la sección 2.1.2, se le asigna a un procesador diferente. Una vez los hilos se encuentran activos y fijados cada uno a un procesador diferente se envía una señal a cada uno de ellos para que planifique sobre ese hilo la tarea idle que se corresponde con ese procesador. Es por esto que las señales deben encontrarse desenmascaradas al realizar la inicialización multiprocesador. Las tareas idle que se planifican tienen su estructura de datos registrada y el stack preparado para ser ejecutadas tras el primer cambio de contexto. Una vez que ya hay una tarea idle ejecutándose en cada procesador, MaRTE OS puede realizar cambios de contexto sobre él de forma normal.

3.6. HAL

Para una mejor utilización de la funcionalidad que se ha ido introduciendo como consecuencia de generar un sistema multiprocesador, se ha extendido la interfaz abstracta con el hardware (HAL) que posee MaRTE OS. Esta extensión de la interfaz define un conjunto de funciones en el nuevo paquete `HAL.SMP` que facilita la gestión de los diferentes procesadores. Actualmente se encuentra implementado el procedimiento de inicialización comentado en la sección 3.5. También se han definido dos funciones que se utilizan en el proceso de inicialización para indicar y comprobar, respectivamente, que los

procesadores se van inicializando correctamente, de forma que no se continua con la inicialización hasta que todos los procesadores han sido inicializados. Las dos funciones que más van a ser utilizadas son la función que permite conocer el procesador sobre el que se está ejecutando (sección 2.4), que va a ser utilizada principalmente en el paquete del planificador para identificar sobre que procesador ha de actuar, y la función que permite el envío de una interrupción a otro procesador (2.3). Estas dos últimas funciones son importadas desde C, lenguaje desde el que también pueden ser utilizadas.

```
package Marte.HAL.SMP is

   pragma Preelaborate;

   function Cpuid return Integer;
   pragma Import (C, Cpuid, "marte_smp_cpuid");
   pragma Inline (Cpuid);

   function Send_IPI (cpu : in Integer; sig : in Integer) return Integer;
   pragma Import (C, Send_IPI, "marte_smp_send_ipi");
   pragma Inline (Send_IPI);

   procedure CPU_Ready (cpu : Integer);

   function Is_Ready (cpu : Integer) return Boolean;
   pragma Inline (Is_Ready);

   procedure Initialize;

end Marte.HAL.SMP;
```

La función que permite el envío de interrupciones entre procesadores, en la arquitectura linux_lib envía una señal al hilo que se encarga de implementar el procesador. Se utiliza una señal libre, no utilizada por la versión monoprocesador de MaRTE OS y que no pertenece al conjunto de señales de usuario.

p tu 4

Pruebas y evaluación

Uno de los puntos clave de la implementación realizada es el cambio de contexto entre diferentes tareas, así como el reto de la utilización de todos los procesadores. Para ello durante la implementación se han utilizado diferentes programas para comprobar que se estaba utilizando todos los procesadores disponibles y que la planificación se estaba realizando de forma correcta.

La utilización de los diferentes procesadores se ha comprobado realizando lecturas del identificador del procesador como se ha visto en la sección 2.4. Inicialmente comprobando donde se ejecuta cada uno de los hilos de Linux generados, y posteriormente viendo que las tareas generadas se ejecutan en los diferentes procesadores disponibles.

La prueba básica es la de proporcionar carga a todos los procesadores y a la vez generar cambios de contexto para comprobar que el planificador realiza correctamente su trabajo. La carga hace referencia al tiempo de procesador que consume cada tarea, a mayor carga mayor será el tiempo necesario para

Cuadro 4.1: *Tiempo medio de ejecución de la prueba.*

Sistema	**Tiempo (μs)**
monoprocesador	603,039
multiprocesador	162,650

completar la tarea. Por otro lado, el número de tareas es el número de instancias de la tarea que se van a ejecutar en el sistema, van ha tener relevancia en la carga del sistema y en la cantidad de cambios de contexto que se van a producir.

Una de las pruebas que se han realizado mide las diferencias entre las versiones de MaRTE OS, mono- y multiprocesador, ante un conjunto de tareas que repetidamente ejecutan una porción de código para después dormirse, provocando que el procesador esté constantemente en uso y realizando cambios de contexto. En el cuadro 4.1 se muestran los tiempos empleados para la ejecución de 30 tareas. Las tareas se componen por un bucle que itera 100 veces ejecutando una porción de código de aproximadamente unos $160\mu s$ y una suspensión de $50\mu s$. Como se puede apreciar la diferencia es aproximadamente cuatro veces, pues estamos utilizando cuatro procesadores, lo que indica que el reparto de tareas por procesador se hace correctamente.

Otra de las pruebas evalúa el tiempo que tarda el nuevo sistema en realizar un cambio de contexto. Estas medidas no son tan relevantes como lo serían si se ejecutarían en un sistema de tiempo real, pero permiten apreciar diferencias entre el comportamiento de ambos sistemas, mono y multiprocesador. Así mismo, las medidas permiten tener una estimación de los tiempos del cambio de contexto y comprobar que éstos no son desorbitados ni impiden la utilización práctica del sistema. Probar las aplicaciones sobre esta arquitectura nos permiten analizar su comportamiento en una situación complicada de forma que cuando se ejecute el programa en un sistema empotrado la mayoría de situaciones conflictivas ya hayan sido probadas y analizadas.

Esta prueba está compuesta por una tarea principal de prioridad alta, varias tareas de relleno de prioridad media (al menos tantas como procesadores) y una de prioridad baja. Las tareas de relleno sirven para mantener a los procesadores ocupados y que la tarea de prioridad baja se mantenga en la cola sin ser ejecutada. La tarea principal realiza una lectura del reloj del sistema, guardando su valor en una variable global, para acto seguido elevar la prioridad de la última tarea a un valor alto, por encima de las tareas de relleno, de forma que pase a ser ejecutada. Esta tarea lee nuevamente el reloj y calcula el tiempo empleado por el sistema para realizar el cambio de con-

Cuadro 4.2: *Tiempo medio del cambio de contexto.*

Sistema	**Tiempo (μs)**		
	mínimo	medio	máximo
monoprocesador	2	3,039	7
multiprocesador (local)	6	10,650	53
multiprocesador (remoto)	9	26,031	569

texto. Posteriormente, vuelve a reducir su valor de prioridad, de forma que es expulsada, y espera para repetir el proceso. Repitiendo esta operación un número importante de veces se obtienen los valores medios presentados en el cuadro 4.2. Como se puede observar el tiempo empleado en realizar el cambio de contexto aumenta ligeramente al tener que realizar más cálculos, frente al caso monoprocesador. Dependiendo del valor que toma la prioridad de la tarea principal, si es mayor o menor que la prioridad de las tareas de relleno, se tienen dos casos diferentes de planificación. Si la prioridad de la tarea principal es mayor, al realizar el cambio de prioridad de la tarea más baja, esta se tiene que planificar sobre uno de los procesadores que contienen las tareas de relleno, que son las de más baja prioridad entre las que se encuentran en ejecución. En este caso, el cambio de contexto se produce de forma remota y será necesario el envío de una interrupción, por lo que el proceso se ralentiza. Sin embargo, si la prioridad de la tarea principal es menor, va a ser ésta la que sea expulsada y por lo tanto se tendrá un cambio de contexto local.

Cuando el cambio de contexto es remoto conlleva un envío de señal y el tiempo empleado aumenta considerablemente, cosa normal puesto que los cálculos se realizan en los dos procesadores. Sin embargo, este aumento del tiempo se debe en gran parte al envío de la señal Linux, que como se puede apreciar en el cuadro 4.3 puede ser bastante variable, que comparando cno los tiempos del cuadro 4.3 puede verse que supone un 38 % del tiempo medio total de cambio de contexto.

Cuadro 4.3: *Tiempo de envío de una señal en Linux.*

Sistema	**Tiempo (μs)**		
	mínimo	medio	máximo
Linux	1,682	10,036	504,229

p tu 5

Conclusiones y trabajo futuro

Una vez finalizado el proyecto se realiza el balance sobre el trabajo realizado y las dificultades encontradas, y se detallan aspectos que se salen de los objetivos de este proyecto y que constituirán líneas de trabajo futuro.

5.1. Concl ione

Como se ha visto a lo largo de este proyecto el objetivo principal, que era realizar una implementación de MaRTE OS que aproveche los diferentes procesadores de un sistema multiprocesador, se ha cumplido. Con la adaptación realizada en este proyecto se dispone de un sistema operativo capaz de planificar un conjunto de tareas entre múltiples procesadores de forma que se aprovechan los recursos disponibles en los sistemas multiprocesador.

Este nuevo sistema operativo podrá ser utilizado como base de desarrollos futuros, y ,aunque sobre Linux, presenta gran parte de la problemática de un sistema SMP real.

Uno de los aspectos que ha marcado el trabajo ha sido la dificultad a la hora de depurar la implementación. El depurador *gdb* soporta la depuración de programas con varios hilos de ejecución, y si es ejecutado se puede observar como son lanzados los hilos que se crean en la inicialización y que se asignan a los procesadores. Sin embargo, al intentar depurar la inicialización de las tareas idle sobre los hilos Linux creados, MaRTE OS no es capaz de completar el cambio de contexto sobre el hilo Linux y se queda bloqueado. La mayoría de la depuración se ha realizado mediante escrituras en pantalla.

5.2. Trabajo f ro

El trabajo realizando en este proyecto no es más que el comienzo de un nuevo área de trabajo e investigación con MaRTE OS.

- Aunque se ha conseguido realizar una implementación multiprocesador de MaRTE OS falta integrar el trabajo realizado con la distribución oficial para que pueda ser utilizado por terceras personas.
- El fin último que ha motivado este trabajo es poder disponer de una versión multiprocesador de MaRTE OS que se ejecute en una máquina desnuda multiprocesador. Para ello es necesario realizar un port a la arquitectura x86 y/o ARM multiprocesador.
- Esta versión de MaRTE OS incluye un algoritmo de planificación sencillo y funcional. Una vía de trabajo puede ser utilizar la plataforma que se ha implementado para probar otros algoritmos de planificación y trabajar con la afinidad de tareas a nivel de MaRTE OS.
- El estado del sistema operativo ya permite realizar investigaciones interesantes. Por ejemplo, la implementación de la interfaz multiprocesador propuesta para la futura revisión del estándar Ada (Ada 2012) con el objetivo de validar su utilidad y completitud.

Bibliografía

[1] Mario Aldea Rivas and Michael González Harbour. MaRTE OS: An Ada Kernel for Real-Time Embedded Applications. In Ada-Europe-2001, Leuven, Belgium, editor, *International Conference on Reliable Software Technologies*, pages 125–134, May 2001.

[2] Mario Aldea Rivas and Michael González Harbour. *Planificación de Tareas en Sistemas Operativos de Tiempo Real Estricto para Aplicaciones Empotradas*. PhD thesis, University of Cantabria, November 2002.

[3] MaRTE OS Web Page. *http://marte.unican.es*.

[4] Paul E. McKenney. Smp and embedded real time. *Linux Journal*, Issue #153, January 2007.

[5] ARM. *The ARM Cortex-A9 Processors Withepaper*, Sept. 2009.

[6] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 1: Basic Architecture*, February 2008.

[7] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 2A: Instruction Set Reference, A-M*, November 2007.

[8] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 3A: System Programming Guide, Part 1*, November 2007.

[9] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 2B: Instruction Set Reference, N-Z*, November 2007.

[10] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 3B: System Programming Guide, Part 2*, July 2008.

Acrónimos

API Application Programming Interface

APIC Advanced Programmable Interrupt Controller

ARM Advanced RISC Machines Ltd.

CPU Central Processing Unit

CS Context Switch

ECS Enter Critic Section

ECSfI ECS from Interrupt

ECSfST ECS from Start Task

FIFO First In First Out

GCC GNU Compiler Collection

GNU GNU is Not Unix (proyecto con el objetivo de crear un sistema operativo completamente libre)

GNAT GNU NYU Ada Translator

HAL Hardware Abstraction Layer

IPI InterProcessor Interruption

LCS Leave Critic Section

LCSfI LCS from Interrupt

NUMA Non-Uniform Memory Access

NYU New York University

PID Process Identification Number

POSIX Portable Operating System Interface; la X viene de UNIX como seña de identidad de la API. ("Interfaz para Sistemas Operativos portables basados en UNIX")

SF Sign Flag

SMP Symmetric Multi-Processing

TID Thread Identification Number

UMA Uniform Memory Access

ZF Zero Flag