Centro de Investigación y de Estudios Avanzados del IPN

Departamento de Ingeniería Eléctrica
Sección Computación

Kernel de Tiempo Real para el Control de Procesos

Tesis que presenta

Oscar Miranda Gómez

Para obtener el grado de:

Maestro en Ciencias

en la Especialidad de

Ingeniería Eléctrica Opción Computación

Asesores de la Tesis:

Dr. Pedro Mejía Alvarez
Dr. Alberto Soria López

México, DF Marzo del 2004.

Resumen

En esta tesis presentamos el análisis, el diseño y la estructura de un Kernel de Tiempo Real. Este Kernel se ejecuta sobre MS-DOS y trabaja con tareas periódicas, desalojables y en ambientes de un solo procesador. Las ventajas de este sistema operativo de tiempo real son principalmente su tamaño, su modularidad, y la facilidad de elegir el tipo de planificador a utilizar. Esta última característica nos permite integrarle nuevos planificadores experimentales sin tener que modificar su estructura. El Kernel de Tiempo Real es pequeño (de tamaño 20.5Kb), lo cual permite su implementación en ambientes empotrados, y a pesar de estar implementado sobre MS-DOS, tiene poca dependencia ya que no realiza llamadas al sistema ni hace uso del BIOS ni de los dispositivos del sistema.

El Kernel realiza funciones que todo sistema en tiempo real requiere, como son: a) manejo de interrupciones externas, b) ejecución de tareas concurrentes, c) comunicación y sincronización entre procesos mediante las primitivas de acceso a buzones y semáforos, d) planificación de tareas periódicas y aperiódicas, mediante las políticas de planificación Rate Monotonic y Earliest Deadline First (EDF), y e) manejo de tiempos. El Kernel de Tiempo Real fue probado bajo una aplicación de control de motores de corriente directa en donde los motores fueron controlados por posición y por tiempo, utilizando controladores de tipo PD. En esta aplicación se planificó la ejecución del Kernel utilizando planificación Rate Monotonic y EDF.

Abstract

In this thesis work, we introduce the analysis and design of a real-time kernel. This kernel is executed on top of MS-DOS and executes using periodic and preemptable tasks on one processor. The advantages of this kernel are mainly its small size, its modularity and the flexibility provided to choose different scheduling policies. The kernel allows the integration of new schedulers in its architecture.

The size of the kernel is small (20.5 KBytes), which allows its implementation in embedded platforms. Although the kernel was developed on top of MS-DOS, it uses few resources from the OS (only the timer interrupts).
The main features of the Real-Time Kernel lies in its functionality. It contains primitives for a) concurrent task execution, b) interrupt handling, c) task synchronization and communication, d) time handling and e) Rate Monotonic and EDF scheduling policies.

Our Real-Time Kernel has been tested using a realistic industrial setup. It was tested for controlling the position and timing of four DC Motors. In our testing examples, the Rate Monotonic and EDF were used.

Agradecimientos

Al CONACYT por el apoyo económico que me brindó durante el transcurso de mi maestría.

Agradecimiento especial a mis asesores, al Dr. Pedro Mejía Alvarez, por haberme apoyado y aconsejado durante todo el desarrollo de la tesis, y al Dr. Alberto Soria López, por sus consejos y ayuda en todo lo relacionado al control automático.

A los doctores de la sección de computación: Dr. Francisco Rodríguez, Dr. Carlos Artemio Coello, Dr.José Oscar Olmedo, Dr. Jorge Buenabad , Dr. Sergio Chapa, Dr. Arturo Díaz, Dr. Luis Gerardo de la Fraga, Dra. Xiaoou Li, Dr. Guillermo Morales y Dr. Adriano de Luca, por compartir sus conocimientos y ser partícipes de mi vida académica.

Agradezco también a los auxiliares del laboratorio Gerardo Castro Zavala y José de Jesús Meza Serrano su valiosa asesoría para la utilización y conexión del material utilizado para la implementación de las pruebas del Kernel en tiempo real. Sin su apreciable ayuda me hubiera sido más difícil concluir con éxito el presente trabajo.

A mis padres por el apoyo que me dieron en todo momento y por guiarme en el camino de la educación.

A mis amigos, su amistad y compañerismo hizo más placentera mi estancia en el CINVESTAV.

Y a Sofía Reza por el apoyo administrativo a lo largo de la maestría.

Índice general

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

En la actualidad el avance de la tecnología ha permitido una miniaturización en los dispositivos electrónicos y un incremento en la capacidad de procesamiento de estos dispositivos. Estos dispositivos se encuentran en general empotrados dentro de aplicaciones tales como teléfonos celulares, PALM's, computadoras industriales, robots, satélites, automóviles y también en distintos aparatos electro-domésticos.

La mayoría de estos dispositivos contienen un procesador y un pequeño sistema operativo empotrado el cual es capaz de controlar todo el hardware de manera eficiente. Debido a la naturaleza de estos dispositivos empotrados, en ocasiones se les demanda no sólo un correcto y eficiente funcionamiento sino también un estricto cumplimiento de requerimientos temporales. A estos sistemas se les conoce como sistemas de tiempo real empotrados.

En un sistema de tiempo real, el principal componente lo constituye el sistema operativo o Kernel. Un sistema operativo de tiempo real se ejecuta por lo general sobre un plataforma embebida (que puede ser un microcontrolador, un DSP o cualquier procesador convencional). Este sistema operativo debe ser capaz de controlar todos los recursos de hardware de la plataforma en que se encuentra y también de administrar todos los tiempos de ejecución de las tareas de tiempo real. Normalmente este sistema operativo no maneja discos, memoria cache, DMA o complejos sistemas de redes de comunicaciones, debido a que su ejecución debe ser predecible (debe ser posible estimar con exactitud sus tiempos de ejecución).

1.1. Motivación

En esta tesis, la motivación de construir un kernel desde su etapa inicial se debe a que se desea obtener el completo conocimiento y control del Sistema Operativo. El objetivo final es obtener un kernel lo suficientemente pequeño y con la capacidad de poder ser instalado en sistemas empotrados que contienen muy poca memoria, por ejemplo, aquellas plataformas que sólo tienen capacidad de memoria de entre 16kb hasta 64kb.

La mayoría de los kernels que existen han sido diseñados para trabajar con determinadas plataformas y por lo tanto sus primitivas de control se encuentran muy ligadas al hardware. Si se quisiera trabajar con estos kernels, entre los posibles problemas que se presentarían se encuentran:

1. la necesidad de conseguir (además del sistema operativo) la plataforma para la cual se encuentra diseñado lo cual podría ser muy caro y en el peor de los casos poco común.

2. muchos sistemas operativos de tiempo real, en especial los comerciales, no distribuyen el código fuente por lo que es casi imposible adaptarlos o portarlos a otras plataformas para las que no fueron diseñados.

Con el kernel que se diseñó, estos problemas se solucionan ya que se tiene toda la documentación y el conocimiento sobre el kernel y por lo tanto su modificación o migración a otro tipo de plataformas no consumiría mucho tiempo. Sería sólo necesario para conocer a detalle la plataforma a la que se quiere portar sin tener que preocuparse por la estructura del kernel.

Por otro lado, esta tesis forma parte de un proyecto de investigación llamado "Ambiente de Desarrollo de Sistemas en Tiempo Real Orientado a Control de Procesos", el cual está formado por tres proyectos de tesis, una doctoral y dos de maestría:

- Un Proyecto para el diseño de Sistemas de Tiempo Real, basado en UML y Simulink (Tesis doctoral).

- Un Kernel de Tiempo Real (Tesis de maestría).

- Sistema Visualizador Gráfico de Procesos en Tiempo Real (Tesis de maestría).

El objetivo de este proyecto es crear un ambiente de diseño, en donde los procesos de tiempo real se programen de forma visual, se genere automáticamente el código de cada tarea y éste se ejecute en el kernel de Tiempo Real.

1.2. Trabajo Propuesto

Se propone el diseño de un kernel de tiempo real capaz de manipular y controlar procesos concurrentes utilizando distintos mecanismos de planificación (existentes o experimentales). Se requiere que el kernel no sea muy grande, es decir, que cuente sólo con las primitivas básicas, con el objetivo de poder portarlo a distintas plataformas sin tener que realizarle muchos cambios.

Actualmente, la mayoría de los Sistemas en Tiempo Real que existen (tanto experimentales como comerciales), han sido diseñados para plataformas específicas. El kernel que se propone diseñar estará en un principio pensado para ejecutarse en una máquina con procesador del tipo INTEL 80x86, sin embargo debido a su tamaño y a que no contendrá ningún tipo de controladores de hardware innecesario tales como CD-ROM, Impresoras, Diskettes, etc. se podrá fácilmente portar a cualquier plataforma con tan sólo modificar los archivos de configuración del kernel en base al código o instrucciones que maneje la nueva plataforma o sistema empotrado.

1.3. Trabajos Relacionados

Existen en la actualidad un gran número de sistemas operativos de tiempo real comerciales y experimentales. Entre los sistemas operativos comerciales que existen, analizaremos los siguientes: VxWorks [1] y RT-Linux [2]. Estos sistemas operativos incluyen una amplia gama de servicios tales como manejo de procesos, manejo de memoria, manejo de tiempos y manejo de dispositivos e interfaces con distintas plataformas de hardware. Por lo general estos sistemas operativos son de gran tamaño, y están diseñados para cubrir una amplia gama de aplicaciones y servicios para sistemas embebidos. Por otro lado, existen los sistemas operativos experimentales, de los cuales los más conocidos en la actualidad son: EMERALDS de la Universidad de Michigan [4], S.H.A.R.K de la Escuela Superior Santa Ana de Pisa Italia

[5] y SPRING de la Universidad de Massachusetts en Amherts [6]. Estos sistemas operativos incluyen distintos métodos de planificación, son de tamaño medio a pequeño (512kb - 128kb), cuentan con interfaces gráficas y son bastante flexibles en su diseño. En varios de estos sistemas operativos se incluyen técnicas desarrolladas por dichas Universidades, tales son los casos de SPRING que se basa en la *técnica de planificación del mejor esfuerzo (best effort scheduling)* o EMERALDS que utiliza una *técnica de planificación llamada CSD (Combined Static/Dynamic Scheduler)* que combina los algoritmos de planificación Rate Monotonic y Earliest Deadline First.

1.4. Sistemas Operativos de Tiempo Real Comerciales

1.4.1. VxWorks

Inicialmente VxWorks fue un ambiente de desarrollo en red para VRTX y PSOSytem. Sus creadores fundaron una empresa llamada "Wind River Systems" donde crearon su propio microkernel. El resultado obtenido fue VxWorks, el cual fue diseñado para funcionar sobre arquitecturas cliente-servidor. VxWorks apareció oficialmente en 1987 como un microkernel orientado a objetos, rápido, eficiente y altamente escalable. Para que Wind River trabaje en conjunto con VxWorks, se cuenta con un ambiente integrado de desarrollo de aplicaciones empotradas llamado "Tornado". Tornado es un ambiente de diseño abierto para ser ajustable y extendido por el desarrollador. Este ambiente viene con un extenso conjunto de herramientas como compiladores, depuradores, herramientas para medir el desempeño de los datos en tiempo real, etc. [8].

Arquitectura

El corazón del sistema VxWorks es el Wind kernel. Este microkernel soporta un amplio rango de servicios de tiempo real como la multitarea, la planificación, la sincronización, la comunicación entre tareas y el manejo de memoria. Como se puede apreciar en la Figura 1.1, el sistema VxWorks cuenta en la capa más baja con librerías y primitivas que son dependientes del hardware y en la parte alta todas aquellas que son independientes. Estas últimas son implementadas como procesos y realizan funciones de red, de entrada y salida, de manejo

de archivos, etc. VxWorks (en especial la versión 5.3.1) puede ser configurado para usarse desde un sistema empotrado pequeño con fuertes restricciones de memoria, hasta complejos sistemas donde otras funciones son necesarias.

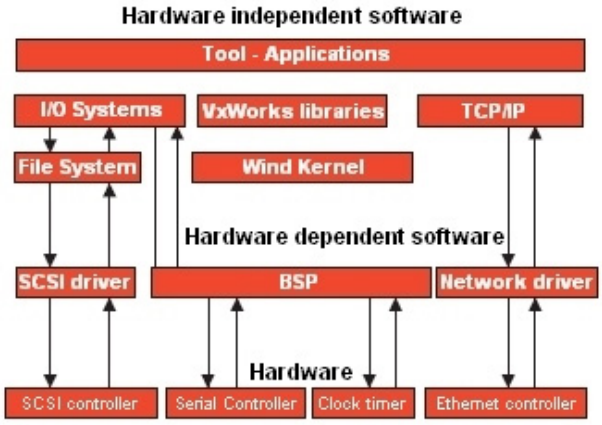

Figura 1.1: Arquitectura de VxWorks

Tareas o Procesos

VxWorks provee un ambiente multitarea básico, maneja prioridades de hasta 256 niveles (donde 0 es el de mayor prioridad) y los primeros 50 niveles están reservados para el sistema. El número máximo de tareas está limitado por la cantidad de memoria disponible.

Memoria

Todo el sistema y todas las tareas comparten el mismo espacio de memoria, lo que significa que aplicaciones defectuosas o mal diseñadas podrían accidentalmente accesar a los recursos del sistema y comprometer la estabilidad del sistema entero. Por otro lado, la empresa Wind-River Systems provee un componente adicional llamado VxVMI que necesita ser comprado por separado. Este componente (VxVMI) permite que cada proceso tenga su propia memoria virtual privada.

Mecanismos de Planificación

Cuenta con dos algoritmos de planificación, el algoritmo de planificación Round Robin y un planificador basado en prioridades. Este último algoritmo puede funcionar en dos modalidades:

1. *Con prioridad fija*, en donde cada proceso tiene una prioridad asignada desde el principio y no cambia durante toda su ejecución en el kernel.

2. *Con prioridad variable*, en donde una tarea se empieza a ejecutar con una prioridad y ésta tiende a decrecer conforme el tiempo de ejecución va aumentando [9].

Manejo de Interrupciones

Para lograr responder lo más rápido posible a interrupciones externas, las rutinas de servicio de interrupciones se ejecutan en un contexto especial (fuera de cualquier hilo de contexto), de esta manera se asegura que ninguna tarea al cambiar de contexto las involucre.

Mecanismos de Comunicación entre Tareas

VxWorks cuenta con eficientes mecanismos de comunicación entre tareas que les permiten coordinar sus acciones dentro del sistema de tiempo real. Para un simple intercambio de datos se puede utilizar el mecanismo de memoria compartida, pero si se desea una comunicación entre tareas y el CPU, es necesario utilizar las tuberías (pipes) o las colas de mensajes. Este último mecanismo está formado por un complejo entramado de colas donde los procesos ponen todo lo que quieren escribir o comunicar a otras tareas.

Mecanismos de Sincronización entre Tareas

Los mecanismos de sincronización con los que cuenta VxWorks son:

- *Sockets y Llamadas a Procedimientos Remotos:* Utilizados para mantener una comunicación en red transparente.

- *Señales:* Para el manejo de excepciones.

- *Semáforos:* Para controlar los recursos críticos del sistema. VxWorks cuenta con diferentes tipos de semáforos: binarios, contadores y de exclusión mutua con herencia de prioridad.

1.4.2. RT-Linux

RT-Linux (Real Time Linux) surgió a partir del trabajo de Michael Barabanov y Victor Yodaiken en la Universidad de Nuevo México. Actualmente continúan activamente con su desarrollo desde su propia empresa (FSM Labs) desde la que ofrecen soporte técnico. RT-Linux se distribuye bajo la "GNU Public License" y recientemente Victor Yodaiken ha patentado la arquitectura original en la que se basa RT-Linux [10]. RT-Linux funciona sobre arquitecturas PowerPC, i386, y está en desarrollo la versión para Alpha. A partir del código de Yodaiken, se está desarrollando otro proyecto liderado por P. Mantegazza llamado: "Real Time Application Interface" RTAI. Las primeras versiones de RT-Linux ofrecían un API (Aplication Programming Interface) muy reducido sin tener en cuenta ninguno de los estándares de tiempo real: POSIX Real-Time extensions, PThreads, etc. A partir de la versión 2.0, Victor Yodaiken decide reconvertir el API original a otro que fuera "compatible" con el API de POSIX Threads.

Arquitectura

Es un pequeño kernel que coexiste con el kernel de Linux, permitiéndole a éste operar funciones de tiempo real en un ambiente predecible y de baja latencia. RT-Linux se basa en la idea de máquinas virtuales para poder ejecutar un sistema operativo de tiempo compartido estándar y un gestor de tiempo real en la misma máquina. En sí, RT-Linux no es código independiente, sino que parte de su código es un parche sobre el código de Linux y otra parte son módulos cargables. RT-Linux se sitúa entre el hardware y el propio sistema operativo, creando una máquina virtual para que Linux pueda seguir funcionando [Figura 1.2]. RT-Linux toma el control de todas las interrupciones e implementa un gestor de interrupciones por software.

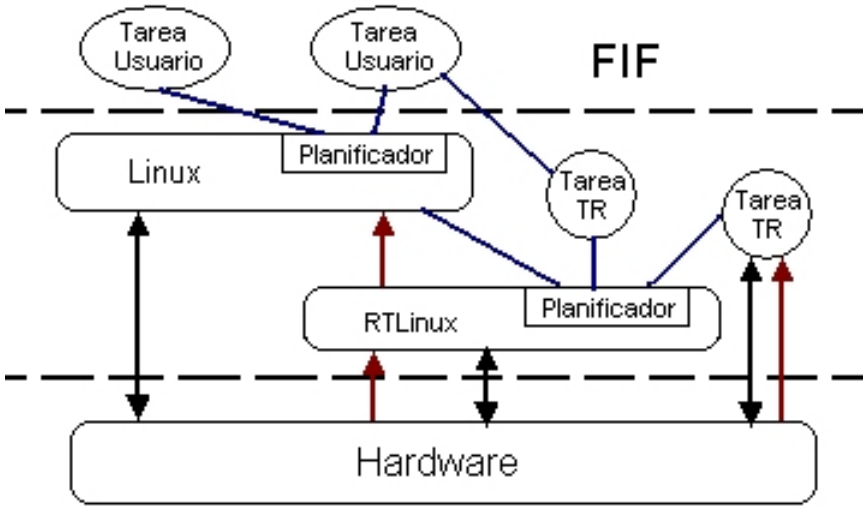

Figura 1.2: Arquitectura de RT-Linux

Tareas o Procesos

Todas las tareas son definidas estáticamente ya que no soporta asignación de memoria dinámica. Los procesos de tiempo real se clasifican en hilos (threads) ligeros y pesados, cada uno con su propio espacio de memoria. El contexto de un proceso de tiempo real sólo consiste de registros enteros, lo cual asegura un rápido cambio de contexto. Actualmente sólo se soportan tareas de tiempo real periódicas. Todas las tareas comparten el mismo espacio de memoria que el núcleo, por lo que pueden acceder a todas las variables y funciones de éste; aunque no es recomendable ya que se podría producir interbloqueos.
Las tareas de tiempo real (RT-Tasks) no pueden hacer uso de las llamadas al sistema de Linux y se ejecutan en modo supervisor; lo que significa que pueden ejecutar cualquier instrucción de procesador y tienen acceso a todos los puertos de entrada y salida (E/S).

Memoria

Las RT-Tasks se ejecutan en el mismo espacio de memoria que el núcleo de Linux. RTLinux utiliza funciones de Linux como *mbuff()* para compartir zonas de memoria entre los procesos de Linux y las RT-Task. Para reservar memoria, RT-Linux puede utilizar alguno de los siguientes métodos:

- ***Memoria Alta no Utilizada.-*** Consiste en forzar a Linux a utilizar menos memoria alta de la que hay instalada en la máquina. La técnica consiste en indicarle a Linux que existe menos memoria de la que en realidad hay, de esta manera, la zona alta de memoria queda libre para ser utilizada por RT-Linux.

- ***BigPhysArea.-*** Es un parche en el núcleo de Linux que reserva durante el arranque del sistema operativo, la cantidad de memoria física que se le indique.

Mecanismos de Planificación

Utiliza un simple planificador basado en prioridades. Este está implementado como una rutina que se encarga de elegir un proceso de entre varios que se encuentran listos y clasificados con prioridad "alta" y lo marca como el siguiente proceso a ejecutar. Una tarea puede abandonar voluntariamente al procesador o puede ser desalojada por otra tarea con mayor prioridad cuyo tiempo de ejecución está por iniciar [11].

Manejo de Interrupciones

En cuanto al manejo de interrupciones, RT-Linux es el que controla todo el hardware y es el único que tiene acceso directo a él. Linux por su parte se encuentra en ejecución sobre una máquina virtual creada por RT-Linux y todas las interrupciones que solicite serán atendidas por RT-Linux. RT-Linux se encargará de simular por software (a través de la máquina virtual) la interrupción solicitada. Con esto se puede ver que RT-Linux es capaz de definir nuevas interrupciones para la máquina virtual de Linux y que estas nuevas interrupciones harán "creer" a Linux que provienen de algún periférico en hardware, mientras que en realidad, las interrupciones habrán sido originadas por alguna tarea de RT-Linux.

Mecanismos de Comunicación entre Tareas

RT-Linux maneja dos tipos de mecanismos de comunicación entre tareas que son: mediante FIFOs o a través de señales [12].

- ***Señales:*** En UNIX las señales se utilizan como un mecanismo de "sincronización asíncrono", en donde las señales hacen el papel de las interrupciones. RT-Linux hace

uso extensivo de este concepto (señal/interrupción) y lo utiliza para comunicar RT-Tasks entre si, así como para modificar el estado de ejecución de las tareas, atender interrupciones hardware y disparar eventos en el núcleo de Linux.

- ***FIFOs:*** Es un mecanismo de comunicación basado en las "FIFO" de UNIX, pero adaptado a tiempo real. Las FIFO implementadas en RT-Linux no tienen relación con las FIFO clásicas de UNIX. Ambas son implementaciones distintas de un mismo método de comunicación. Se pueden utilizar para comunicar tanto hilos entre sí, como hilos con procesos Linux existan. La comunicación con los FIFO es unidireccional, estos es, se comporta como un buffer circular de forma que cada operación de lectura puede eliminar los datos leídos.

Mecanismos de Sincronización de Procesos

La sincronización es llevada a cabo mediante mecanismos tradicionales como: Mutex, Variables de Condición y Semáforos. Las *Variables de Condición* son los tipos de datos y funciones necesarias para que, junto con el Mutex, se puedan construir "monitores". Un monitor es un conjunto de funciones que operan sobre un conjunto de datos para llevar a cabo la exclusión mutua. En cuanto a los semáforos, la funcionalidad de estos no es muy compleja ya que junto con los Mutex y las Variables de Condición se puede resolver cualquier tipo de problema relacionado con la sincronización.

1.5. Sistemas Operativos de Tiempo Real Experimentales

1.5.1. EMERALDS

EMERALDS (Extensible Microkernel for Embedded ReAL-time, Distributed Systems) es un microkernel de tiempo real diseñado para sistemas empotrados de tamaño medio a pequeño con poca memoria (32-128 Kbytes) y cuyas aplicaciones deben correr sobre procesadores lentos (12-25MHz). EMERALDS actualmente está diseñado para trabajar en procesadores Motorola 68040 y su tamaño es de 13 Kbytes.

Arquitectura

EMERALDS es un microkernel de tiempo real escrito en el lenguaje C++ y gracias a su diseño es fácil instalarle nuevos protocolos de comunicación así como nuevos manejadores de dispositivos (drivers). Como se puede ver en la figura 1.3, EMERALDS maneja procesos multihilos, planificación de tiempo real, espacios de memoria protegida, paso de mensajes, semáforos y temporizadores; todo esto de manera eficiente y manteniendo el tamaño del kernel pequeño [13].

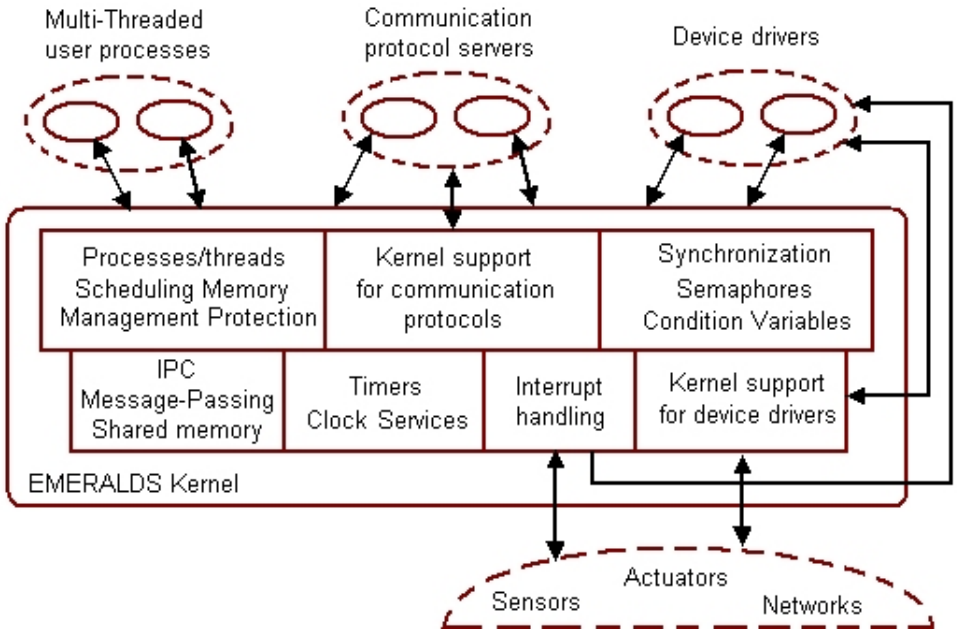

Figura 1.3: Arquitectura de EMERALDS

Tareas o Procesos

Un proceso en EMERALDS es una entidad pasiva representando una dirección protegida de memoria en la cual el hilo se ejecutará. Cada hilo tiene una prioridad especificada por el usuario.

Memoria

Las técnicas para la asignación de memoria que utiliza EMERALDS son la paginación en memoria física y la capacidad de asignar memoria virtual. Muchos sistemas operativos de tiempo real (RTOS) omiten la protección de memoria e hilos con la finalidad de reducir el tamaño e incrementar la velocidad, sin embargo en EMERALDS por el contrario se provee de la protección requerida para la tabla de páginas (sin que esto afecte en el tamaño). Para evitar que creciera el tamaño del kernel, al implementar la protección de memoria se partió del hecho que todas las aplicaciones en los sistemas empotrados se encuentran en memoria y que por lo tanto las regiones a proteger no necesitaban de algoritmos complejos.

Mecanismos de Planificación

EMERALDS provee un algoritmo de planificación basado en prioridades y tiene soporte parcial en la planificación dinámica. El usuario especifica la prioridad del hilo en forma estática (durante el tiempo de diseño) basándose en los algoritmos Rate Monotonic (RM) o Earliest Deadline First (EDF). EMERALDS por su parte utiliza un algoritmo de planificación llamado Combined Static/Dynamic Scheduler (CSD), que es una combinación de los algoritmos de planificación RM y EDF. La política de planificación de este algoritmo es la siguiente: de un 100 % de tareas en el sistema, el 50 % de tareas con prioridad alta son planificadas con EDF para evitar que pierdan sus plazos y el otro 50 % de tareas con menos prioridad, son planificadas con RM [14].

Manejo de Interrupciones

EMERALDS cuenta con un controlador de dispositivos el cual se encarga de llamar al kernel y decirle que subrutina del ISR (Interrupt Service Routines) ejecutar cuando la interrupción ocurre. Un ISR separado puede ser anexado a cada nivel de interrupción, tantas veces como exista comunicación entre los hilos del usuario y los controladores de dispositivos.

Mecanismos de Comunicación entre Tareas

Los procesos en estos sistemas tienden a intercambiar mensajes cortos y simples como la lectura de un sensor y comandos de un actuador. En EMERALDS se utiliza el tradicional

mecanismo para el intercambio de información entre tareas que es: el paso de mensajes por medio de buzones.

Mecanismos de Sincronización de Procesos

Los métodos utilizados para sincronizar los procesos en EMERALDS son: la exclusión mutua y los semáforos. Éstos funcionan de la misma manera que en otros sistemas operativos, por ejemplo: si un hilo intenta adquirir un semáforo que esta siendo ocupado en ese momento, ese hilo será bloqueado y se agregará a la cola de hilos esperando por ese semáforo. Cuando el hilo que ocupaba el semáforo lo libere, el hilo con mayor prioridad en la cola podría ser desbloqueado.

1.5.2. S.H.A.R.K.

SHARK (Soft and HArd Real-time Kernel) es un kernel de investigación para ayudar en la implementación y prueba de nuevos algoritmos de planificación, tanto para el CPU como para otros recursos. El kernel puede ser usado para validar fácilmente las ejecuciones de los algoritmos de planificación producidos en laboratorios de investigación. Este método permite a los desarrolladores enfocar su atención en el diseño de los algoritmos, ahorrándoles el tiempo de implementación de las nuevas soluciones.

Arquitectura

Con el propósito de realizar independencia entre aplicaciones y algoritmos de planificación, SHARK esta basado sobre un kernel genérico, el cual no implementa un algoritmo de planificación en particular, sino que relega las decisiones de planificación a entidades externas llamadas “módulos de planificación”. De manera similar, el acceso a recursos compartidos es coordinado por “módulos de recursos” [15].

El kernel provee las primitivas sin especificar los algoritmos que residen en los módulos externos [ver Figura 1.4]. Actualmente SHARK tiene dos tipos básicos de módulos:

1. Módulos que implementan algoritmos de planificación periódicos y servicios especiales a políticas aperiódicas llamados *Módulos de Planificación.*

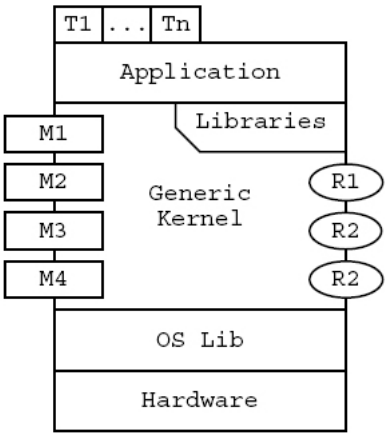

Figura 1.4: Arquitectura de SHARK

2. Módulos que manejan recursos compartidos (Hardware y Software) llamados *Módulos de Recursos*.

Tareas o Procesos

Las tareas son definidas como funciones en C estándar. Una tarea es descrita por un identificador unico de proceso y un número secuencial. Las tareas pueden ser periódicas ó aperiódicas. Las tareas periódicas son activadas automáticamente por el kernel, mientras que las tareas aperiódicas pueden ser activadas por una llamada explicita al sistema ó por la ocurrencia de un evento. Las tareas pueden tener diferentes niveles de criticidad, como son: **HARD** (las tareas críticas en el sistema), **SOFT** (aquellas tareas que pueden perder algunos plazos) y **NRT** (para aquellas tareas que no son de tiempo real).

Memoria

El kernel de SHARK cuenta con un conjunto de funciones estándar para la asignación de memoria que son provistas por las librerías del lenguaje C. Las más utilizadas son:

- ***calloc():*** la cual asigna memoria para un arreglo de n elementos,
- ***realloc():*** la cual cambia el tamaño de memoria asignado y
- ***free():*** la cual libera el espacio de memoria que haya sido asignado por calloc o realloc.

El kernel también provee un asignador de memoria basado en FLUX OS kit, el cual divide la memoria en tres regiones: 1) Memoria menor a 1MB, 2) Memoria entre 1MB y 16MB, 3) Memoria por debajo de los 16MB, y por último 4) Memoria por encima de los 16MB [16].

Mecanismos de Planificación

Como se mencionó anteriormente, este kernel no tiene implementado un algoritmo de planificación específico ya que la planificación de las tareas se realiza a través de los módulos de planificación. Actualmente algunos de estos módulos contienen los siguientes algoritmos de planificación: Rate Monotonic, Earliest Deadline First, Round Robin y Slot Shifting.
Cuando el kernel genérico tiene que desempeñar una decisión de planificación, los módulos se encargan de "preguntar" por la tarea a ejecutar. Primero se va con el módulo con más alta prioridad, si el módulo no tiene alguna tarea lista, entonces se le preguntará al módulo que le siga en prioridad y así consecutivamente. De esta manera cada módulo maneja su lista privada de tareas preparadas y el kernel genérico planifica la primer tarea encontrada en el módulo con mayor prioridad.

Manejo de Interrupciones

Cuando una interrupción se presenta, se crea un código al cual se le transfieren los datos para responder a la interrupción. Este código puede ejecutarse en dos modos diferentes:

1. El código puede ser encapsulado por completo en una función, para ser ejecutado inmediatamente con la llegada de la interrupción (sobre el contexto de la tarea en ejecución).

2. El código puede ser encapsulado por completo en una tarea, la cual será activada con la llegada de la interrupción y se planificará en base a su prioridad junto con las demás tareas.

El primer método es apropiado cuando la interrupción necesita un tiempo de respuesta rápido e inapropiado cuando el tiempo de computo no es corto, debido a que toda la planificabilidad puede ser afectada severamente; esto es porque el algoritmo no toma en cuenta el tiempo de ejecución del manejador de interrupciones. El segundo método por el contrario, está integrado perfectamente con el mecanismo de planificación del kernel, sin embargo puede causar

retardos considerables en la transferencia de datos ya que espera su turno, como cualquier otra tarea, para poder ser atendida.

Mecanismos de Comunicación entre Tareas

El intercambio de mensajes se lleva a cabo mediante puertos de comunicación. Cada puerto es único e identificable mediante un nombre simbólico. SHARK ofrece tres tipos de puertos:

- ***STREAM:*** Facilita la comunicación uno-a-uno y puede ser abierto tanto por la tarea receptora como por la emisora.
- ***MAILBOX (Buzón):*** Este mecanismo facilita la comunicación muchos-a-uno. Fue pensado para ser usado en el modelo cliente/servidor. Este puerto de comunicación sólo puede ser abierto por la tarea receptora, la cual tiene que recibir datos desde las tareas emisoras.
- ***STICK:*** Facilita la comunicación uno-a-muchos. Se pretende que sea usado para el intercambio de mensajes periódicos, para lo cual, la información más reciente es la más relevante. Este mecanismo puede ser abierto sólo por la tarea emisora y las tareas receptoras tendrían que conectarse a él.

Los dos primeros tipos de puertos implementan comunicación síncrona, mientras que los puertos STICK permite implementar la comunicación asíncrona. Los puertos STREAM usan semáforos para sincronizar el acceso. Los puertos STICK sólo usan el semáforo de exclusión mutua y los puertos MAILBOX utilizan ambos tipos de semáforos.

Mecanismos de Sincronización de Procesos

SHARK cuenta con los siguientes mecanismos de sincronización:

- ***Semáforos:*** La interfaz de los semáforos en SHARK es similar a la de los semáforos POSIX, con la diferencia de que SHARK agrega algunas otras primitivas que permiten incrementar/decrementar el contador del semáforo por más de una unidad de tiempo con la finalidad de evitar la inversión de prioridad.

- ***Mutexes:*** Un Mutex puede ser considerado como un semáforo binario inicializado en 1. De esta manera, una sección critica puede ser especificada usando sólo las primitivas *mutexblock()* y *mutexunblock()*.

- ***Variables de Condición:*** Las variables de condición tienen la ventaja de permitir a una tarea bloquearse a si misma en la espera de un evento.

1.5.3. Spring

Spring surgió como un proyecto de investigación en la universidad de Massachusetts en el año de 1990, pero fué hasta 1992 (con su segunda versión) cuando el kernel empezó a tener logros significativos. Spring es un kernel de tiempo real en donde sus creadores aseguran que provee algunos de los soportes básicos requeridos para la siguiente generación de sistemas operativos, especialmente en el conocimiento de las restricciones de tiempo. Spring surge de la necesidad de construir sistemas operativos de tiempo real predecibles y flexibles ya que la siguiente generación de sistemas de tiempo real podrían ser largos, complejos, distribuidos, adaptables, podrían contener muchas restricciones de tiempo, operar sobre ambientes no determinísticos y evolucionar sobre un largo sistema de por vida.

Arquitectura

Spring fue desarrollado para que pudiera trabajar sobre un sistema distribuido llamado *SpringNet* [ver Figura 1.5]. SpringNet es físicamente un sistema distribuido compuesto de una red de nodos multiprocesador en donde cada nodo ejecuta el kernel Spring. Cada nodo contiene uno o mas procesadores de aplicación, un procesador de sistema y un subsistema de entrada/salida.

El procesador de sistema (SP) es el encargado de administrar las funciones, particularmente las de planificación de tareas. El ó los procesadores de aplicación (AP), son los encargados de ejecutar las tareas garantizadas por el planificador. Por último, el subsistema de entrada/salida (I/O) es dividido en partes por el kernel Spring y es el que maneja los dispositivos no-críticos y lentos de entrada/salida así como a los sensores rápidos. La arquitectura de

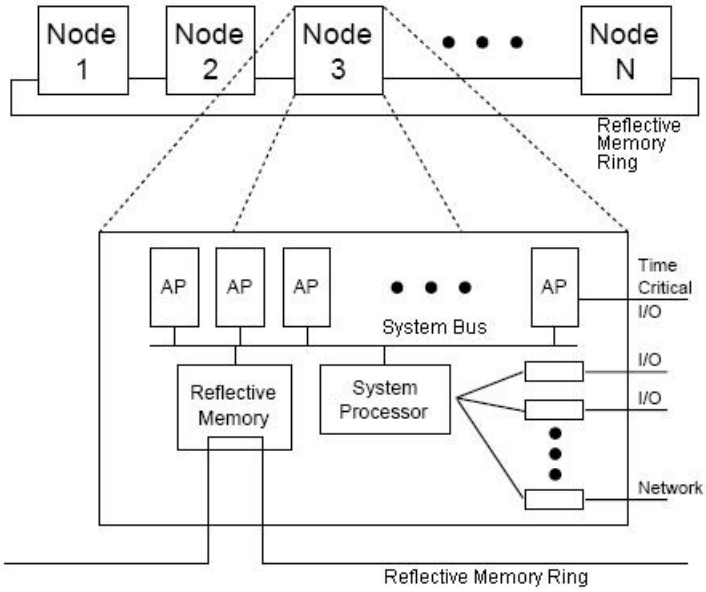

Figura 1.5: Arquitectura de Spring

Spring incorpora las ideas del nuevo paradigma propuesto por los desarrolladores del kernel, basado en la idea de la planificación del "mejor esfuerzo" (Best-Effort Scheduling)[17].

Tareas o Procesos

Una tarea consiste de código reentrante con datos locales y globales, una pila, un descriptor de la tarea y un bloque de control de tarea. Cada tarea adquiere recursos antes de iniciar su ejecución y éstos son liberados una vez que su ejecución termine. Las tareas son parte de una única aplicación y los tipos de tareas que pueden ocurrir en una aplicación de tiempo real son conocidas de antemano, por lo tanto, pueden ser inicializadas para determinar sus características. Esta información podría ser utilizada en tiempo de ejecución [17]. Basados en la importancia y requerimientos de tiempo, en Spring se encuentran definidas tres tipos de tareas: tareas críticas, tareas esenciales y tareas no esenciales.

Memoria

Las primitivas para el manejo de memoria son utilizadas para crear recursos definidos como pilas, bloques de control de tareas, descriptores de tareas, datos locales y globales, puertos,

discos virtuales y memoria no segmentada. Las tareas requieren de un máximo número de segmentos de memoria de cada tipo, pero en su correspondiente tiempo de activación una tarea podría requerir menos segmentos de los que podría necesitar después. Para solucionar esto, Spring asigna todos los segmentos requeridos cuando la tarea inicia su ejecución y los deja residentes en memoria. Todos los procesos tienen páginas pre-asignadas, cargadas en tiempo de ejecución y mantenidas residentes en memoria, por lo que no es necesario implementarle un método para bloquear memoria [20].

Mecanismos de Planificación

En Spring el mecanismo de planificación no está basado en prioridades como en la mayoría de los sistemas operativos de tiempo real. El algoritmo de planificación de Spring integra la planificación del CPU y la de recursos para proveer garantías. Este mecanismo de planificación se llama "Best-Effort" y tiene las siguientes características: es dinámico, planifica tareas expulsivas y no expulsivas, con precedencia, niveles de importancia y tolerancia a fallos. En el algoritmo "Best-Effort", planificar un conjunto de tareas para encontrar una planificación factible es un problema de búsqueda y se resuelve a través de una búsqueda en un árbol. Un vértice del árbol es una planificación parcial y una hoja o un vértice terminal es una planificación completa. En el peor caso, encontrar una planificación óptima requeriría de una búsqueda exhaustiva. El planificador puede ser utilizado en línea y fuera de línea. Este algoritmo de planificación ha sido implementado en software como parte del kernel Spring y en hardware en un chip VLSI [6].

Manejo de Interrupciones

Las interrupciones que son generadas por los dispositivos I/O son atendidas por el sistema de entrada/salida. Sin embargo, existen otras interrupciones generadas por algún ambiente no determinístico y que podría afectar la predecibilidad del kernel. Para evitar esto, las tareas que se ejecutan sobre los procesadores de aplicación son aisladas de cualquier interrupción no predecible y sólo los procesadores de sistema junto con los subsistemas I/O pueden ser afectados por estas interrupciones. Si de manera indirecta alguna interrupción llegara a afectar a tareas que se ejecutan sobre los procesadores de aplicación, sería responsabilidad del

"algoritmo de garantía" atender la petición.

Mecanismos de Comunicación entre Tareas

La comunicación entre procesos (IPC) en Spring ha sido diseñada con el fin de ofrecer una comunicación predecible. Por lo tanto, las primitivas de comunicación tienen límites en el tiempo de ejecución y la comunicación síncrona no permite que se produzcan bloqueos no predefinidos. En Spring la comunicación se realiza a través de puertos, los cuales son objetos de memoria protegida que contienen unidades de datos llamados mensajes. Los procesos pueden comunicarse con algún otro proceso colocando mensajes en los puertos y a su vez obteniendo mensajes de los mismos.

Mecanismos de Sincronización de Procesos

La predecibilidad es una de las preocupaciones que más se tiene en cuenta en el diseño de un sistema de tiempo real. Como un esfuerzo por acercarse a esta predecibilidad, Spring ha limitado cada aspecto y primitiva del kernel (incluyendo a los mecanismos de sincronización) para cumplir con las exigencias de la siguiente generación de kernels multiprocesadores de tiempo real. Entre las características de los mecanismos de sincronización implementados, destaca la inclusión de "semáforos sólidos" que forman parte de los nuevos paradigmas de los sistemas de tiempo real propuestos por Spring y por lo tanto, han sido diseñados para poder funcionar eficientemente sobre los sistemas de tiempo real multiprocesador.

1.6. Objetivos de la Tesis

1.6.1. Objetivos Generales

En cuanto a su utilidad, el objetivo es diseñar un kernel en tiempo real sólido y capaz de interactuar con el mundo exterior para controlar los eventos que puedan ocurrir dentro de los sistemas en tiempo real. Además de tener la propiedad de permitir la monitorización y el control de todos los procesos.

En cuanto a lo académico, se requiere que sea pequeño para facilitar su migración en plataformas empotradas y permitir experimentar con él en este tipo de plataformas (PALMS,

PDAS). Además, debe permitir incluir nuevos mecanismos de planificación con la finalidad de poder estudiar otros algoritmos que existan e incluso probar aquellos que se diseñen en el CINVESTAV.

1.6.2. Objetivos Particulares

- Obtener el mejor tiempo de respuesta posible que pueda ofrecer el Timer de la máquina. (reprogramación del PIT)
- Evitar que el kernel se extienda (en tamaño) con la finalidad de que permita ser transportado a plataformas empotradas las cuales suelen tener demasiadas restricciones (memorias y dispositivos de almacenamiento secundarios).
- Integrar el Kernel desarrollado con un visualizador de procesos para fines estadísticos.
- Lograr un conocimiento profundo sobre el tema y poder documentarlo de tal forma que pueda servir en un futuro, para el traslado del kernel en algún sistema empotrado.
- Modularizar el kernel de tal forma que no dependa de un solo mecanismo de planificación. Con este se logra tener un kernel que permite experimentar con otro tipo de planificadores que ya existen o incluso con algunos que se pudieran diseñar en un futuro.

1.7. Organización de la Tesis

La escritura de la tesis se desarrolló de la siguiente manera:

En el Capítulo 2 se define el concepto de sistemas de tiempo real, se mencionan los elementos y características de estos, así como las aplicaciones y clasificaciones más comunes de los sistemas de tiempo real. En este capítulo también se explica qué es un sistema operativo de tiempo real y se definen otros conceptos necesarios para la comprensión de la tesis, como el de proceso y quantum. Como los procesos o tareas son el objetivo a controlar dentro del kernel, es necesario saber bien su comportamiento y por lo tanto en este capítulo se explican los estados y transiciones que puede tener un proceso y se muestra el contenido del Bloque

de Control de Procesos (BCP). Para finalizar este capítulo, se muestran y definen los componentes que todo sistema operativo en tiempo real debe contener.

El Capítulo 3 está dedicado por completo a explicar los mecanismos de planificación en los sistemas de tiempo real. Allí se clasifican y muestran las restricciones de las tareas de tiempo real, y se define el problema de planificación. En este capítulo además, se clasifican las políticas de planificación con base en sus prioridades y se dan dos ejemplos por clasificación. Se explica Rate Monotonic y Deadline Monotonic para planificadores basados en prioridades estáticas y se explica Earliest Deadline First y Least Laxity First para planificadores basados en prioridades dinámicas.

En el Capítulo 4 se encuentra todo lo referente al Kernel desarrollado en esta tesis. Se muestra la arquitectura que se utilizó para desarrollar el Kernel, se presenta el BCP utilizado para las tareas del Kernel y cuál es el comportamiento que van a tener dentro del mismo. Se explica cómo configurar el Kernel para que se adapte a las necesidades del usuario y además, se detalla paso a paso cada una de las primitivas y manejadores que lo integran. En este capítulo se encuentra también la estructura general que deben de tener las tareas o procesos creados por el usuario.

El Capítulo 5 contiene los resultados obtenidos de un ejemplo de aplicación implementado al Kernel. Esta aplicación está compuesta por 4 motores y requiere que sean controlados en "posición" mediante técnicas de control. Como antecedentes, este capítulo cuenta con una introducción a la teoría de control y un listado de los tipos de retroalimentación más utilizados en los sistemas de control. Como conclusión, el capítulo describe las pruebas realizadas al kernel y el comportamiento obtenido al trabajar con tareas de tiempo real no simuladas.

Por último, el Capítulo 6 contiene las conclusiones a las que se llegaron con este trabajo de tesis y además, presenta algunas extensiones o modificaciones que se le podrían hacer al kernel como trabajo a futuro con la finalidad de extenderle el campo de aplicación.

Capítulo 2

Sistemas de Tiempo Real

2.1. Definición de Sistemas 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 [21]. Para que el funcionamiento del sistema sea correcto no basta con que las acciones del sistema sean correctas, sino 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 (ver figura 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 aún más complejos y se utilicen en el control de estaciones espaciales, en sistemas integrados

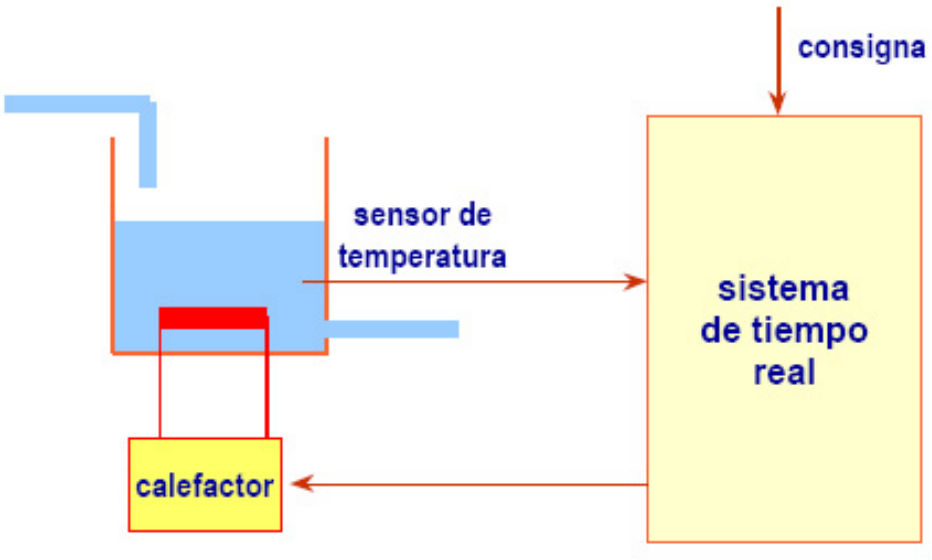

Figura 2.1: Sistema de Tiempo Real.

de visión/robótica/Inteligencia Artificial y en entornos peligrosos como plantas químicas, nucleares, etc.

2.2. Aplicaciones de los Sistemas de Tiempo Real

En el campo de los sistemas de tiempo real se distinguen tres tipos principales de aplicaciones principalmente:

- ***En el control de procesos industriales:*** Este tipo de aplicación se empezó a llevar a cabo desde la década de los 60 y actualmente, es normal el uso de microprocesadores para este tipo de aplicaciones. El objetivo es conseguir que una determinada variable siga una evolución prefijada. Esta variable puede ser: temperatura, presión caudal, nivel dentro de un depósito, velocidad o posición de un motor, etc. La computadora debe ser capaz de generar señales que permitan conseguir el objetivo. Para esto, se parte de una variable a controlar, el valor ideal para esta variable y un algoritmo de control.

- ***En la manufacturación:*** El uso de las computadoras en la manufacturación es esencial para reducir el costo de la producción y aumentar la productividad. Por lo general

en este tipo de aplicación la computadora tiene integrado el diseño del producto a fabricar y se encarga de coordinar las tareas a realizar por los distintos componentes del sistema como son, las máquinas de construcción, las cintas transportadoras, etc.

- ***En la comunicación y aplicaciones militares:*** Este tipo de aplicaciones inicialmente surgen del campo militar. Sin embargo actualmente existe un gran número de aplicaciones que tiene características similares; por ejemplo la monitorización de pacientes en centros médicos, el control del tráfico aéreo y los informes bancarios remotos. Todos estos sistemas contienen un complejo juego de módulos de vigilancia, dispositivos que recogen información y procedimientos que permiten tomar decisiones.

2.3. 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

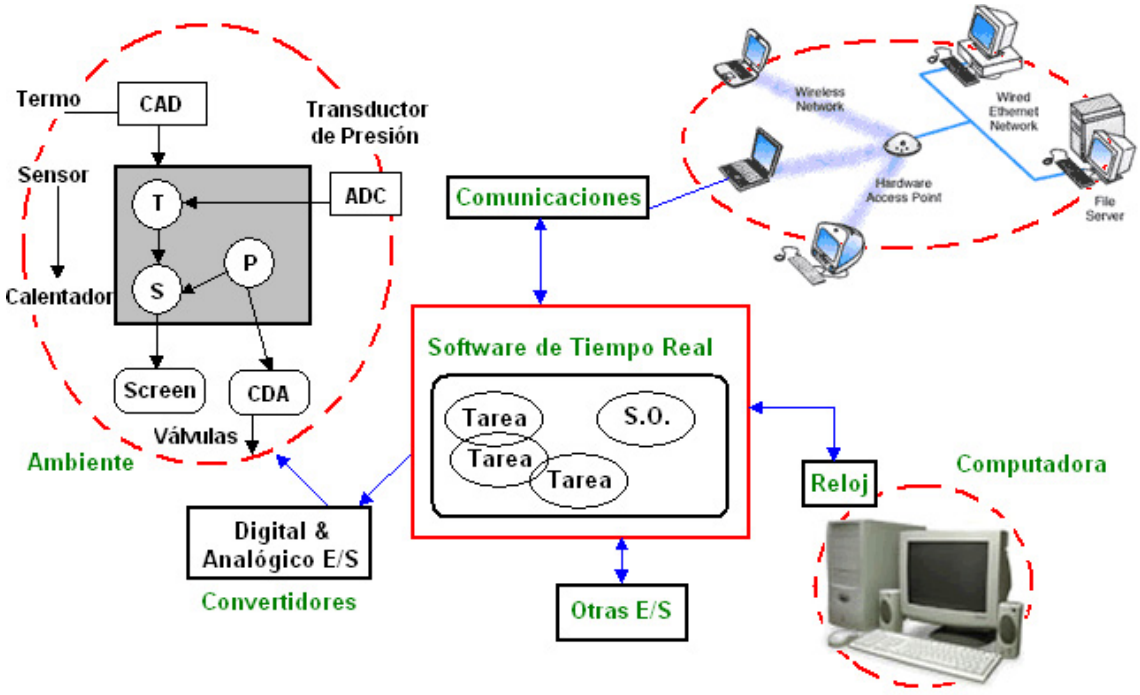

Figura 2.2: Elementos de un Sistema de Tiempo Real.

posible configurar los tiempos de ejecución de las tareas en tiempo real, y por tanto la planificación del sistema, ni tampoco sería posible saber si las tareas cumplen con sus restricciones temporales. En una aplicación que involucra a varias computadoras es necesario que los relojes de tiempo real se encuentren sincronizados, a fin de que todos ellos lleven la misma cuenta de tiempo.

- **Software de Tiempo Real:** El software de tiempo real está compuesto de 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, de tareas, de memoria, de 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. Existe otro software dentro del sistema, como son las librerías, los drivers de dispositivos o los controladores de comunicaciones.

- **Comunicaciones:** En el sistema de tiempo real pueden existir distintas computadoras que interactúan entre sí. Entre ellas existe un medio físico de comunicación (hardware) y un protocolo de comunicaciones (software) 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 aplicaciones 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 otras entradas y salidas principalmente aquellas con las que interactúan con el usuario, como son el teclado, el ratón, y otros dispositivos de interacción con el usuario.

2.4. 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 del entorno 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 pero 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, éstos 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.

2.5. 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, por los flujos de ejecución de los que está compuesto, etc.). La clasificación más conocida es aquella en la que distinguen a los sistemas de tiempo real con base en sus requisitos temporales y de fiabilidad. Esta clasificación separa los sistemas de tiempo real en sistemas críticos y en sistemas acríticos.

2.5.1. 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 alguna situación de fallo de alguno de 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

2.5.2. Sistemas de Tiempo Real Acríticos (Soft Real Time Systems)

Son sistemas en los que es importante el cumplimiento de los requisitos temporales, pero si de alguna manera alguno de ellos no pudiera cumplirse el sistema no produciría un fallo.

[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.

Por ejemplo, en los sistemas multimedia de tiempo real las 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 vídeo 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

2.6. ¿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. En resumen,

se puede decir que la principal responsabilidad de un sistema operativo de tiempo real es la de producir resultados exactos y garantizar el cumplimiento de unos plazos (deadline) predefinidos.

2.7. Proceso

2.7.1. 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 [22].
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* que controla el sistema operativo con un contador de programa que especifica la siguiente instrucción a ejecutar y el conjunto de recursos asociados.

2.7.2. 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.

2.7.3. 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 los tareas dentro del conjunto de tareas.
- **Latencia (L_i):** $L_i = D_i - f_i$, representa el retraso 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. Un proceso con gran importancia es ejecutado con mayor frecuencia que otro que no es tan importante con la finalidad de evitar que el proceso importante pierda su plazo.

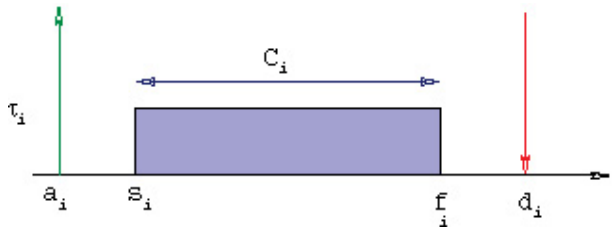

Figura 2.3: Parámetros de un Proceso de Tiempo Real

2.7.4. 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 asigna al proceso recursos, se reserva un espacio en memoria para cargar código, datos y 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.

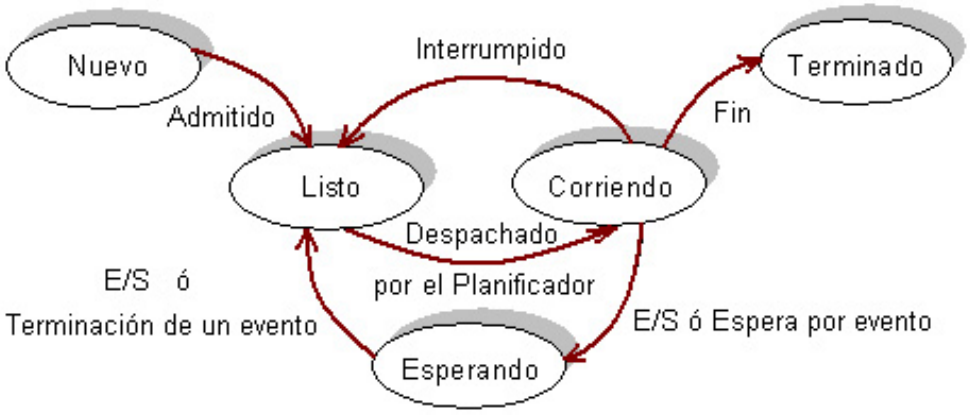

Figura 2.4: Diagrama de estado de los procesos

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.

2.7.5. 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 cómputo (Quantum) o cuando algún proceso de mayor prioridad le quita el procesador.
- ***E/S o Espera por evento:*** Ocurre cuando el proceso necesita esperar a algún dispositivo de entrada/salida o 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.

2.7.6. 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 como se muestra en la figura 2.5, contiene muchas piezas de información asociadas con un proceso específico.

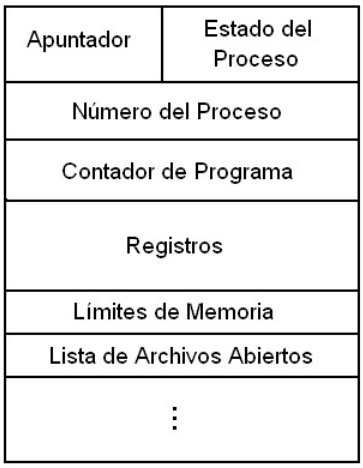

Figura 2.5: Bloque de Control de Procesos (PCB)

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, de segmentos, registros límites, etc. 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, límites de tiempo, números de procesos, etc.
- ***Información del estado de I/O:*** Aquí se encuentra la lista de dispositivos de entrada/salida (I/O) asignados al proceso, la lista de archivos que ha abierto, etc.

2.8. Componentes de un Sistema Operativo de Tiempo Real

2.8.1. 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 [23].

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, puede 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. En este contexto se encuentran incluidos los datos en el PCB del proceso que se está ejecutando. El *cambio de contexto* ocurre cuando un proceso es interrumpido de su ejecución para que otro proceso pueda utilizar el procesador (ver figura 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.

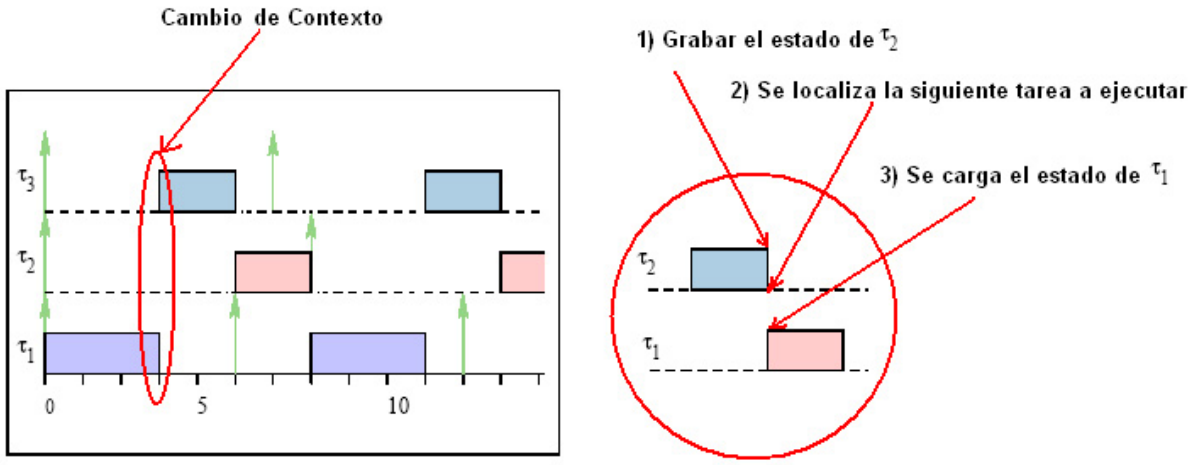

Figura 2.6: Cambio de Contexto.

El cambio de contexto es considerado como *sobrecarga* (overhead) ya que durante el tiempo que éste se lleva a cabo, el sistema queda incapacitado para realizar alguna operación útil. El tiempo que tarda en ejecutarse el cambio de contexto puede variar de una máquina a otra[2], dependiendo de la velocidad de la memoria, el número de registros que contenga y la existencia de instrucciones especiales (tales como alguna instrucción que permita cargar o almacenar todos los registros de una sola vez) [22].

2.8.2. 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 allí 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:

[2]El tiempo que tarda en realizarse el cambio de contexto puede variar entre 1 y 1,000 microsegundos dependiendo de las características de la máquina.

- Decidir qué procesos cargar en memoria cuando esté ésta 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 por lo que no es necesario tener un administrador de memoria complejo.

2.8.3. Manejador de Reloj

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

- Proveer información para el calendario de procesos (planificadores).
- Mantener actualizada la fecha y la hora del día.
- Manejar 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.

2.8.4. Mecanismos de Sincronización y Comunicación

Es otro mecanismo básico que todo kernel debe proveer. 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 [22].

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 tiene prohibida 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 [22].

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)*[3] 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 (ver figura 2.7).
- ***Signal(S):*** Es una operación que incrementa el valor de su argumento semáforo (S)

[3]Las primitivas señal y espera originalmente fueron definidas como P y V (señal y espera en holandés) por Dijkstra, quien fue el que inventó el concepto de semáforo.

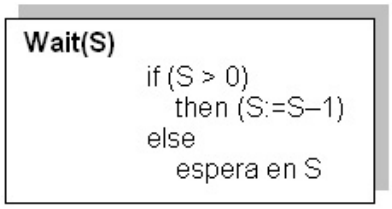

Figura 2.7: Operación WAIT.

siempre y cuando, no haya procesos bloqueados en la cola de semáforos; si los hay, en lugar de incrementar desbloquea al proceso. La operación SIGNAL también es indivisible (ver figura 2.8).

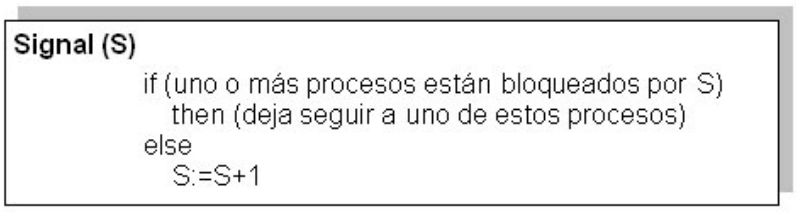

Figura 2.8: Operación SIGNAL.

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 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.

Tipos de Semáforos

Por su granularidad, los semáforos pueden ser:

- ***Semáforos Binarios:*** Los valores que pueden tomar los semáforos binarios son cero y uno debido a que el recurso que controlan, sólo un proceso lo puede poseer. Si un proceso

solicita un recurso controlado por un semáforo binario y es el primero en pedirlo, se le asigna el recurso. Por otro lado si ya estaba un proceso utilizando ese recurso, al nuevo proceso que lo solicitó se le bloquearía hasta que el recurso haya sido liberado. La definición en la operación WAIT es la misma y en la operación SIGNAL el cambio consiste en lo siguiente: Si hay procesos suspendidos, despierta uno, si no, el valor de S es igual a 1.

- ***Semáforos Contadores:*** Los semáforos contadores son útiles cuando hay que asignar un recurso a partir de un conjunto de recursos idénticos. El semáforo tiene como valor inicial el número de recursos contenidos en el conjunto. Cada operación WAIT decrementa en uno el semáforo, indicando que se ha retirado un recurso del conjunto y que lo está utilizando algún proceso. Cada operación SIGNAL incrementa en uno el semáforo, lo que indica la devolución de un recurso al conjunto y que el recurso puede ser asignado a otro proceso. Si se intenta una operación WAIT cuando el semáforo tiene valor o, el proceso deberá esperar hasta que se devuelva un recurso al banco mediante una operación SIGNAL.

Por otra parte, sin importar el tipo de semáforo que sea, los semáforos pueden ser con bloqueo o sin bloqueo.

- ***Semáforos con bloqueo:*** Este tipo de semáforos, dependiendo de la condición en WAIT, pueden llegar a bloquear un proceso hasta que el recurso por el que esperan sea liberado. El mecanismo para bloquear al proceso puede ser de dos maneras, utilizando FIFOs o sólo marcándolos como bloqueados. En este segundo método la operación SIGNAL desbloquea un proceso al azar (sin saber cuál es) mientras que cuando se utilizan FIFOS, la operación SIGNAL siempre desbloquea al proceso que lleva más tiempo en la cola.

- ***Semáforos sin bloqueo:*** Estos semáforos se conocen también como *semáforos con espera activa* y se caracterizan por su operación WAIT, ya que ésta es un ciclo el cual se encuentra constantemente revisando el valor de S en lugar de bloquear el proceso (ver figura 2.9).

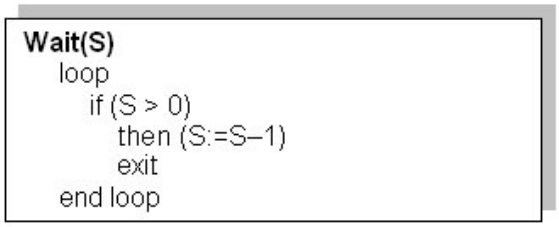

Figura 2.9: Operación WAIT en semáforos sin bloqueo.

Comunicación entre Procesos

Para que los procesos puedan comunicarse es necesario que tengan 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, que 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). Recibir un mensaje de cualquier proceso; la variable id es utilizada para almacenar el nombre del proceso con el que se tuvo comunicación.*

La desventaja en ambos esquemas (simétricos y asimétricos) se encuentra en el momento de cambiar el nombre de un proceso. Tendría que ser necesario examinar en las definiciones

de todos lo procesos aquellos instantes en donde se haga referencia al nombre que tenía antes el proceso para poderlo cambiar por el nuevo nombre.

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 [7], 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* y *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.

2.8.5. 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 éstos 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.

Capítulo 3

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 más 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.

3.1. 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 [27].

3.1.1. Clasificación de las Tareas de Tiempo Real

Con base en los parámetros mostrados en la sección 2.7.3 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 en 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 (ver figura 3.1).
- **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 término 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 (ver figura 3.1).

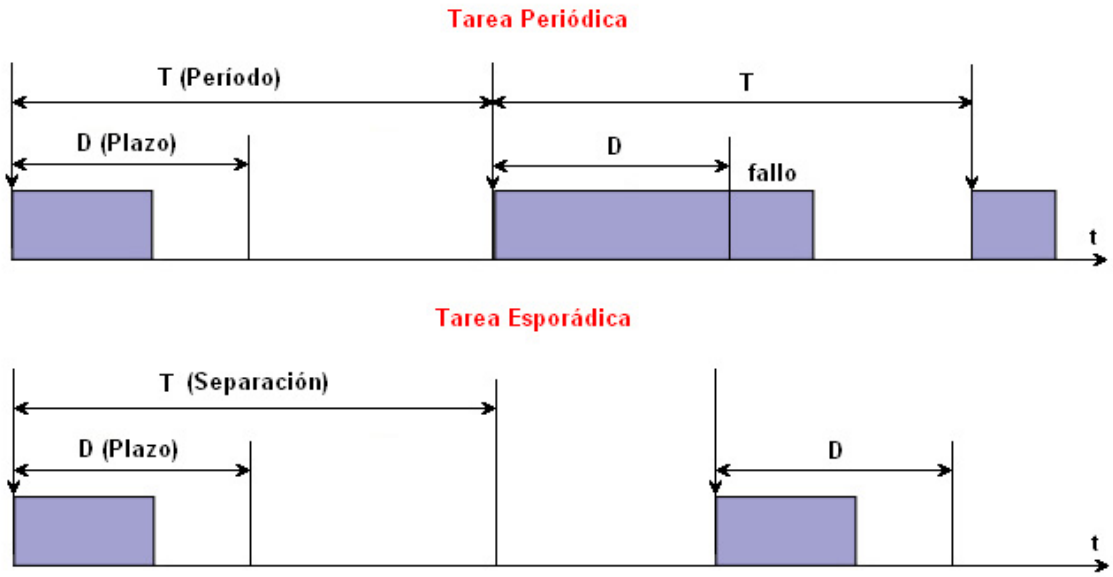

Figura 3.1: Tareas Periódicas y Esporádicas.

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 tareas de mayor prioridad. Por otro lado, las tareas no expulsables no podrán ser interrumpidas sino hasta que éstas 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.

3.1.2. 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 cómputo 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 3.2 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.

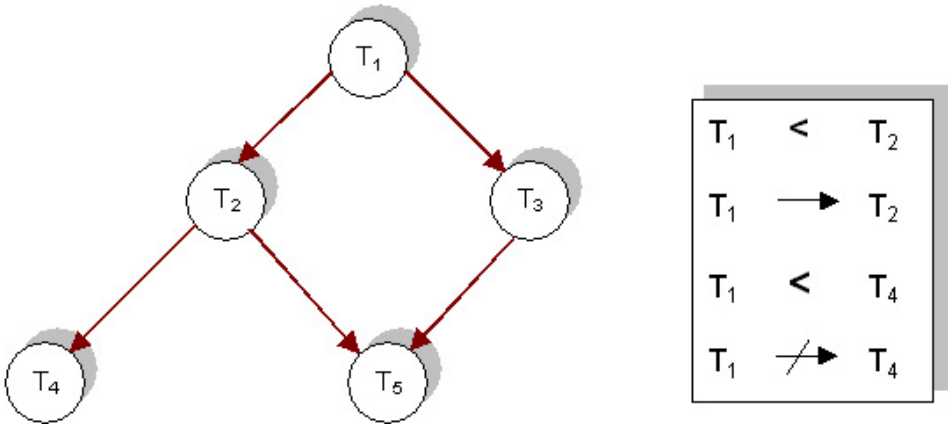

Figura 3.2: Relación de precedencia para cinco tareas.

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, 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 es llamado *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 accediendo a este recurso. Esta situación puede motivar pérdidas 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 que 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.

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.

3.2. 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 [23]. 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 un solo 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).

3.3. Clasificación de Políticas de Planificación

En los sistemas de tiempo real existen muchas estrategias de planificación de tareas. Sin embargo, éstas pueden ser clasificadas en dos grandes grupos: los planificadores cíclicos y los planificadores basados en prioridades (ver figura 3.3).

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 planificadores cíclicos es la poca flexibilidad en el momento de modificar parámetros (añadir o borrar tareas), ya

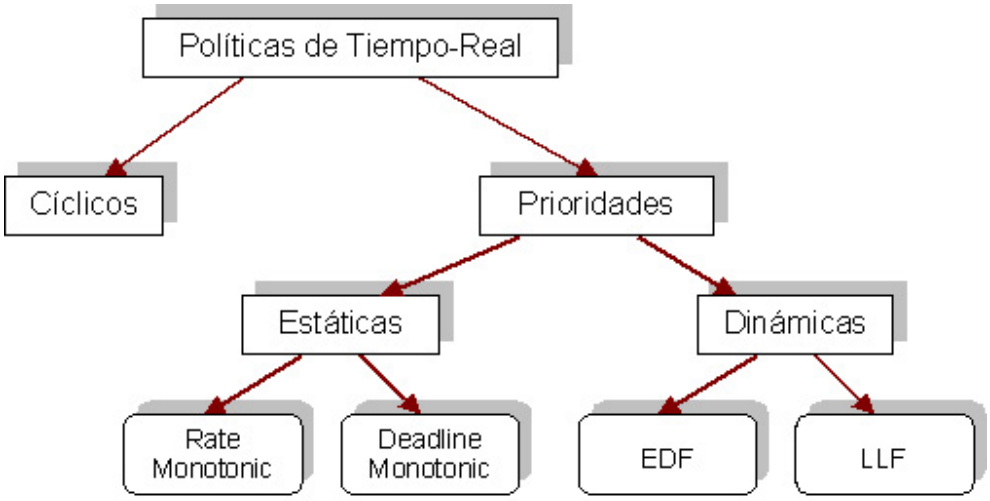

Figura 3.3: Clasificación de los algoritmos de planificación para sistemas de tiempo real.

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*).

3.4. 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 3.1, $T_M = 100$.

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,

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

Tabla 3.1: Conjunto de tareas periódicas

$m \leq \{D_i\}$

- 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 cumplir 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 3.1 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 pueden 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 tienen 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 3.1 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 3.4 se presenta gráficamente la ejecución cíclica para el conjunto de tareas presentado en la tabla 3.1.

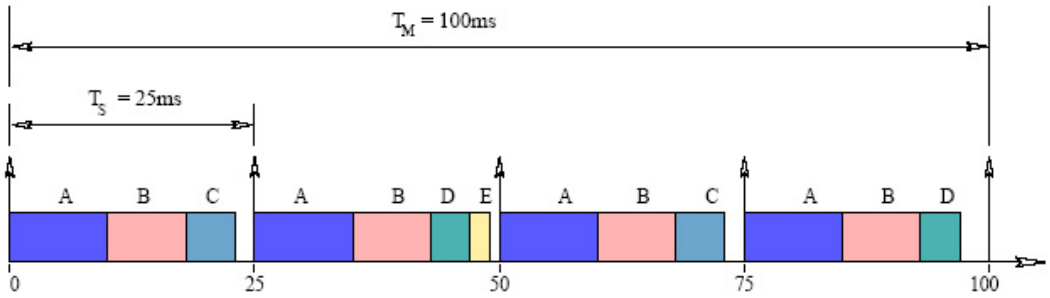

Figura 3.4: Planificación Cíclica.

3.5. 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 línea*. 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.

3.5.1. 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 [24] quienes en 1973 propusieron el algoritmo Rate Monotonic (RM), además demostraron que:

Teorema 3.5.1 *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 [24], los autores proporcionan un *control de admisión* [1] 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:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1) \tag{3.1}$$

Teorema 3.5.2 *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 mínimo* $n(2^{1/n} - 1)$ *cumpla la siguiente condición:*

$$U \leq U_{min} = n(2^{1/n} - 1) \tag{3.2}$$

[1] 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.

La tabla 3.2 describe el comportamiento de la expresión 3.2 para distintos valores de n.

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

Tabla 3.2: Utilización mínima garantizada para n tareas

Como se puede apreciar en la tabla 3.2, al aumentar el número de tareas la utilización mínima garantizada converge en:

$$U_{min} = \ln 2 \approx 0{,}6931 \tag{3.3}$$

Ejemplo: Consideremos el conjunto de tareas[2] τ_1 (30,10), τ_2 (40,10) y τ_3 (50,12) mostrado en la Figura 3.5. El factor de utilización de estas tres tareas es:

$$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{C_3}{T_3} = \frac{10}{30} + \frac{10}{40} + \frac{12}{50} = \frac{494}{600} \approx 0{,}82 \tag{3.4}$$

La utilización total de este conjunto de tareas es mayor que la utilización mínima garantizada para tres tareas establecida por la condición 3.2 (0,82 $>$ 0,7788). Siguiendo la ejecución de las tareas, presentada en la Figura 3.5, es posible observar que la tarea τ_3 del conjunto pierde su plazo de respuesta en el tiempo t=50. Esta pérdida de plazo se debe a que en la

[2]Generalmente en un modelo simple de tareas, una tarea se puede representar como un par ordenado (T_i,C_i), debido a que el plazo de respuesta es considerado igual al período. Es decir, $D_i = T_i$.

primera instancia de ejecución sólo puede ejecutarse por 10 unidades de tiempo. Siendo que su tiempo de cómputo es de $C_3 = 12$, dos unidades de tiempo quedan sin ser ejecutadas. Note que la tarea τ_3 posee la menor prioridad del conjunto utilizando el esquema de asignación de prioridades del algoritmo RM.

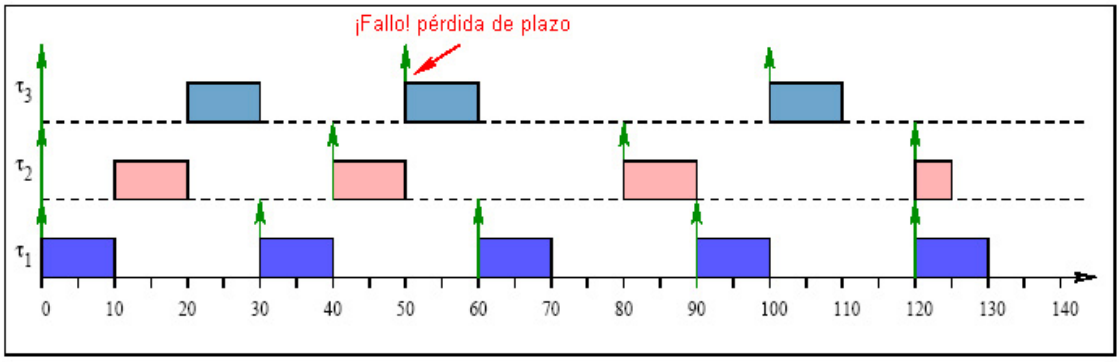

Figura 3.5: Conjunto de tareas que no satisface la Ecuación 3.2 y por lo tanto no es planificable.

En la Figura 3.5 se representa el conjunto de tareas mediante una gráfica de Gantt. Las flechas de la gráfica nos indican los instantes donde las tareas solicitan tiempo de procesador. Los rectángulos sombreados muestran la cantidad de tiempo de cómputo utilizando por la tarea.

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

Tabla 3.3: Conjunto de tareas, la utilización total cumple con la condición 3.2

Por otro lado, consideremos el conjunto de tareas mostrado en la tabla 3.3. Como se puede observar, la utilización total del conjunto de tareas no excede la cota proporcionada por la condición 3.2, es decir, U=0.775 $<$ 0.778 $= U_{min}$. Por lo cual se garantiza que este conjunto de tareas no perderá ningún plazo de respuesta. La Figura 3.6 muestra gráficamente el comportamiento del conjunto de tareas.

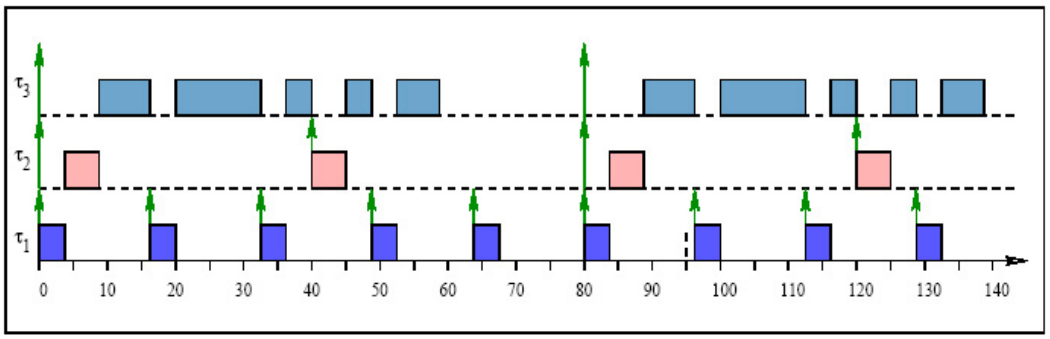

Figura 3.6: Conjunto de tareas que satisface 3.2 y por tanto es planificable.

Es importante observar que la tabla 3.4 describe un conjunto de tareas cuya utilización del procesador es del 100 %, es decir, no satisface la condición 3.2, 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 3.7 muestra la ejecución de las tareas sin que éstas hayan perdido 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 3.4: Conjunto de tareas armónico, es planificable

Condición de Planificabilidad Suficiente y Necesaria

Lehoczky et al. [25] desarrollaron una condición de planificabilidad necesaria y suficiente para un conjunto de tareas que es ejecutado en un sistema uniprocesador. Esta condición se define en el teorema 3.5.3

Teorema 3.5.3 *Sea* $\Gamma = \{\tau_1, \tau_2,...,\tau_n\}$ *un conjunto de n tareas periódicas, con* $T_1 \leq T_2 \leq T_3 \leq ... \leq T_n$.

τ_i *es planificable usando el algoritmo Rate Monotonic si y sólo si*

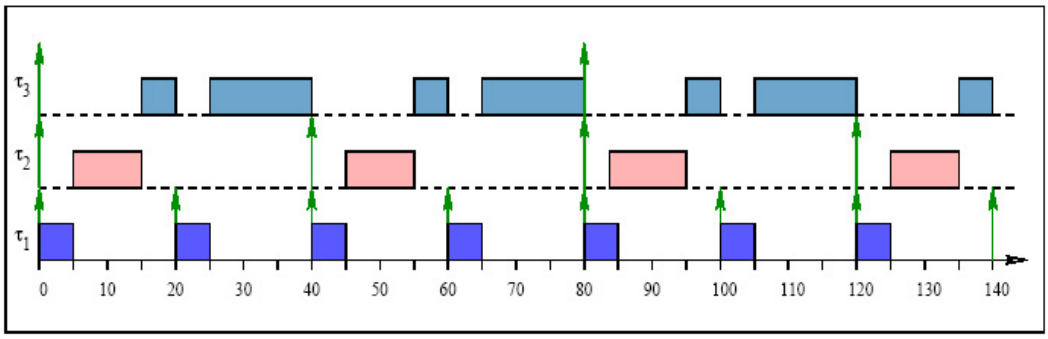

Figura 3.7: Conjunto de tareas que no satisface 3.2. Sin embargo, es planificable.

$$L_i = min_{\{t \in S_i\}} W_i(t)/t \leq 1. \tag{3.5}$$

El conjunto entero de tareas es planificable bajo el algoritmo Rate Monotonic si y sólo si

$$L = max_{\{1 \leq i \leq n\}} L_i \leq 1. \tag{3.6}$$

donde:

Los elementos de S_i son los puntos de planificación en los que la tarea i es planificada. Es decir, los instantes de tiempo en que se cumple el plazo de respuesta y los tiempos de arribo de las tareas de mayor prioridad que i.

$$S_i = \{k \cdot T_j | j = 1, ..., i; k = 1, ..., \lfloor T_i/T_j \rfloor\}. \tag{3.7}$$

$W_i(t)$ mide la cantidad de tiempo de procesador requerido por i tareas, del instante o al instante t.

$$W_i(t) = \sum_{j=1}^{i} C_j \cdot \lceil t/T_j \rceil. \tag{3.8}$$

$$L_i(t) = W_i(t)/t. \tag{3.9}$$

Esta condición de planificabilidad se realiza a partir del instante crítico. El instante crítico es el instante de tiempo en el cual todas las tareas presentan su máximo tiempo de respuesta. Este instante se presenta cuando todas las tareas comienzan a ejecutarse al mismo tiempo. La condición del teorema 3.5.3 es exacta. Si el conjunto de tareas no es planificable bajo esta condición, se debe a que éste no es planificable (bajo ningún algoritmo de planificación), por lo que en este caso, al menos una tarea perderá su plazo de respuesta.

Ejemplo: La tabla 3.5 muestra un conjunto de tareas periódicas el cual será planificado con el algoritmo de planificación RM, Podemos observar que el factor de utilización total es de 0.928, que es mayor que U(3). Por tanto, la condición 3.2 no nos permite asegurar nada sobre si se pueden garantizar los plazos de las tres tareas. Sin embargo, el factor de utilización de las dos primeras tareas (τ_1 y τ_2) es igual a $0.678 < \mathrm{U}(2) = 0.828$, por lo que podemos asegurar que τ_1 y τ_2 terminan siempre dentro de sus plazos de respuesta.

Tarea	T_i	C_i	*Utilización*
1	7	3	0.428
2	12	3	0.250
3	20	5	0.250
			0.928

Tabla 3.5: Conjunto de Tareas que no satisface 3.2.

Se aplica la condición 3.7 para comprobar si el plazo de respuesta de τ_3 está garantizado. Para ello, se examina la ejecución del sistema en el intervalo de tiempo [0,20], suponiendo que $t = 0$ es un instante crítico. Obteniéndose los siguientes puntos de planificación de τ_3:

$$S_3 = \{1 * 7, 2 * 7, 1 * 12, 1 * 20\} = \{7, 12, 14, 20\} \tag{3.10}$$

como se observa en la Figura 3.8, coinciden con los instantes límites de τ_1, τ_2 y τ_3 en el intervalo [0,20]. Aplicando la condición 3.5 se obtiene la cantidad de tiempo de procesador requerido por las i tareas, en el instante t, la tabla 3.6 muestra este hecho.

Puesto que, al menos en un caso ($t = 20$), se verifica la condición 3.5, podemos concluir que τ_3 está garantizada y dado que esta tarea es la de menor prioridad, el conjunto de tareas

t	$W_3(t)$
7	(3*1 + 3*1 + 5*1) = 11
12	(3*2 + 3*1 + 5*1) = 14
14	(3*2 + 3*2 + 5*1) = 17
20	(3*3 + 3*2 + 5*1) = 20

Tabla 3.6: Carga en el instante t

es planificable.

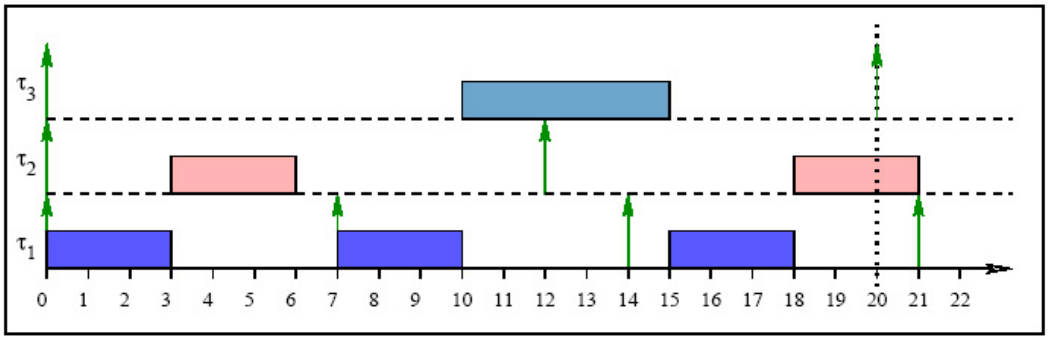

Figura 3.8: Puntos de Planificación.

3.5.2. Deadline Monotonic (DM)

En 1982, Leung y Whitehead [26] 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, a la tarea con el plazo más corto se le asigna la prioridad más alta.

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

Teorema 3.5.4 *[26] 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.

3.6. 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 fuera de 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.

y sus principales desventajas son:

- En una sobrecarga del sistema no es posible predecir qué tareas pierden sus plazos de respuesta. Lo que es más grave, es que en una sobrecarga, la pérdida de plazos de algunas tareas puede producir un efecto en cascada de pérdida 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 prioridad, lo cual introduce sobrecarga al sistema.

3.6.1. Earliest Deadline First (EDF)

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

$$U \leq 1 \tag{3.11}$$

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 proporcione 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 3.7 muestra un conjunto de tareas, cuya utilización total es del 82 % y la Figura 3.9 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 3.11

Tarea	T_i	C_i	*Utilización*
1	30	10	0.333
2	40	10	0.250
3	50	12	0.240
			0.823

Tabla 3.7: Conjunto de tareas, cuya utilización es del 82 %

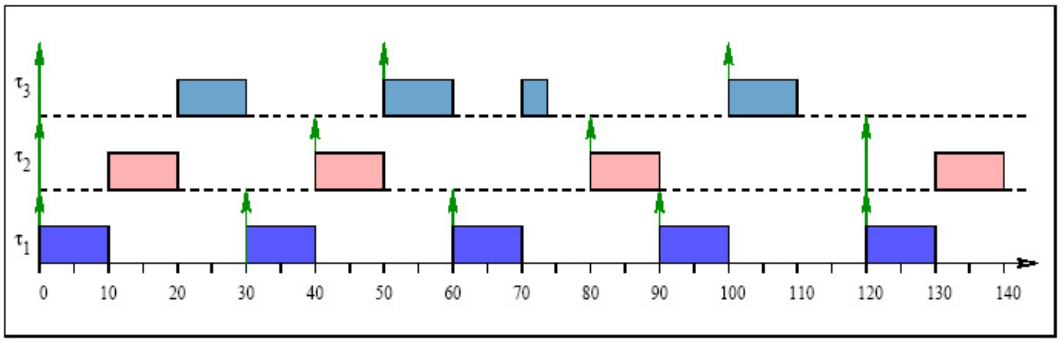

Figura 3.9: Planificación de las tareas de la tabla 3.7 bajo EDF.

3.6.2. 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.

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

$$holgura = D_i - t - C_i(t) \tag{3.12}$$

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=(70,2)$\}$. 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 $t = 2$ por la tarea τ_3 que se ejecuta desde el tiempo 2 al 4. Durante este intervalo, la holgura de τ_1 decrece de 3 a 1. (En el tiempo 4, el resto del tiempo de ejecución de τ_1 es 1, es decir 6-4-1=1). La Figura 3.10 muestra la planificación del conjunto de tareas bajo LLF.

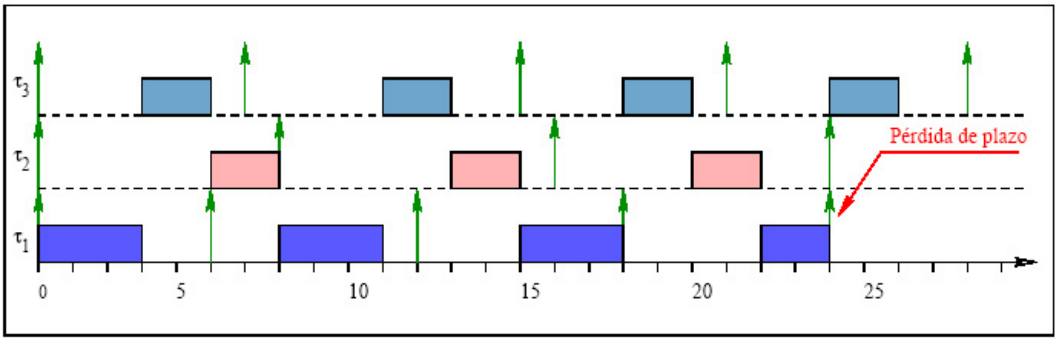

Figura 3.10: Planificación de tareas bajo el algoritmo LLF.

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 en el algoritmo LLF sí se necesita de dicho parámetro temporal. Esto es una desventaja debido a que la holgura es calculada con base en el tiempo máximo de

cómputo, es decir, la determinación de la holgura es inexacta ya que el algoritmo asume el peor caso para calcular la holgura.

Como se puede observar, los algoritmos con manejo de prioridad dinámica tienen cierta ventaja sobre los algoritmos de prioridad fija. La ventaja radica en el hecho en que la cota límite (máxima) es del 100 % para cualquier conjunto de tareas.

Capítulo 4

Diseño del Kernel de Tiempo Real

En la actualidad el avance de la tecnología ha permitido que cada vez más, los aparatos electrónicos sean de menor tamaño y capaces de realizar cualquier tarea de manera aparentemente "inteligente". La mayoría de estos aparatos contiene un procesador y un pequeño sistema operativo empotrado el cual es capaz de controlar todo el hardware de manera eficiente. Sin embargo, no es suficiente con que operen correctamente, sino que además, se requiere que éstos sean seguros y que respondan a eventos en un tiempo de respuesta acotado. En esta tesis, se desarrolló un kernel de tiempo real. Como ya se había mencionado antes, un Kernel es el módulo central de cualquier sistema operativo. Es la parte que se carga primero y permanece en la memoria principal. Normalmente, el kernel es el responsable por la administración de la memoria, los procesos, las tareas y los discos.

En este capítulo se describe la filosofía de diseño que se llevó a cabo para implementar un kernel de tiempo real para el control de procesos, así como la implementación de cada una de las partes que comprende al sistema, tal es el caso de las primitivas y los manejadores (en donde se menciona para cada objeto la estructura que utilizan). También se explican las consideraciones que se deben tener para inicializarlo y algunos otros puntos generales que ayudarán a comprender mejor cómo fue diseñado.

4.1. Arquitectura General

En la Figura 4.1 se presenta la arquitectura del Kernel en tiempo real. En el nivel más bajo se encuentran los distintos dispositivos que controla el Kernel. En el siguiente nivel están las distintas funciones del kernel, conocidas como *primitivas*. En la parte superior, se encuentran los procesos del usuario quienes interaccionan con el Kernel.

El Kernel soporta una arquitectura con un solo procesador, en el cual los procesos se ejecutan concurrentemente. El Kernel se diseñó en el lenguaje de programación C, propios de la familia de procesadores Intel 80x86.

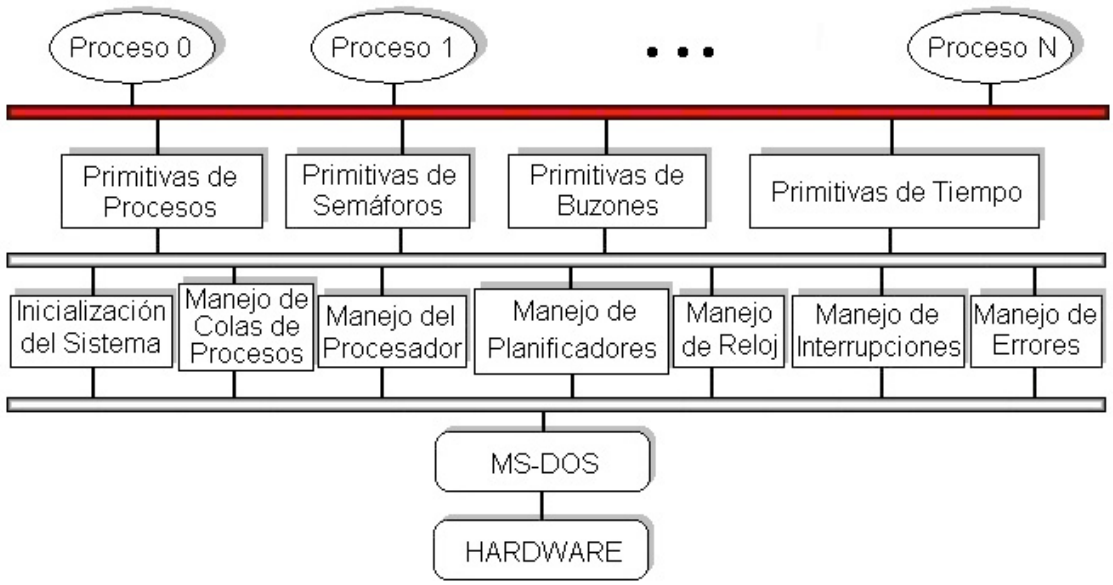

Figura 4.1: Arquitectura del Kernel.

4.1.1. Estructura General de las Primitivas

Todas las primitivas en el kernel, aún cuando realizan funciones diferentes, tienen la siguiente estructura general[1]:

```
Nombre de la primitiva([parámetros])
```

[1]Todo lo que se encuentra dentro de "[ ]" es opcional y puede o no, contenerlo la primitiva.

```
{  Deshabilitar interrupciones
   Validar los parámetros
   Si hay error en los parámetros entonces
      Se llama al manejador de errores y excepciones;
   Si no hubo error
      Se ejecuta la función de la primitiva;
   [ Si algún proceso se alistó o bloqueó entonces
      Llamar al manejador del cpu
   ]
   Habilitar interrupciones
}
```

Como se muestra en el pseudocódigo, en cada primitiva del kernel, lo primero que se hace es deshabilitar las interrupciones. Este paso es muy importante ya que todas las primitivas deben ser atómicas debido a que las operaciones que realizan son importantes y no se deben interrumpir. El siguiente paso, es verificar los parámetros que recibe la primitiva para detectar los posibles errores que pudieran ocurrir. Si se encuentra algún error, se manda a llamar al manejador de errores enviándole como parámetro el identificador o código de error. Si no hubo errores, se ejecuta la primitiva con la seguridad de que todo está en orden. Algunas primitivas como las de manejo de tiempo o buzones necesitan llamar al manejador del procesador (CPU), ya que algún proceso es bloqueado o ingresado a la cola de procesos listos y es necesario que sean tomados en cuenta para la planificación. Antes de salir de la primitiva, es necesario volver a habilitar las interrupciones para que los procesos puedan ser interrumpidos cuando se les termine su tiempo de ejecución.

4.1.2. Estructura de los Procesos o Tareas

Las tareas en el kernel se definen de la misma forma en que se crean las funciones en C estándar. El código de una tarea típica se muestra en la Tabla 4.1. Ahí podemos apreciar que se pueden declarar variables locales (de manera opcional) así como código que sólo se desea ejecutar una sola vez al iniciar la tarea. Todas las tareas tienen un ciclo infinito "*while (1)*", este es el cuerpo de la tarea y aquí va el código que deseamos que se ejecute periódicamente.

Dentro de este ciclo infinito, de manera opcional se puede utilizar la primitiva *Fin_de_Ciclo* la cual se puede utilizar en aquellas tareas en las que se desea que el cuerpo de la tarea sólo se ejecute una vez por cada período.

```
1.  void NombreTarea()
2.    {  // Variables Locales
3.       [ int a,b,c;
4.         // Inicialización de variables
5.         a=10;
6.       ]
7.       ...
8.       while(1)
9.        { // Cuerpo de la tarea
10.         ...
11.         ...
12.         [Fin_de_Ciclo();]
13.       }
14.   }
```

Tabla 4.1: Estructura de las Tareas o Procesos.

4.1.3. Prioridades

El kernel maneja varios niveles de prioridad, el número de prioridades máximo se puede cambiar en el archivo de configuración del kernel llamado CONFIG.H (ver sección 4.4.1). [2] Las prioridades son estáticas y el usuario es quien le asigna las prioridades a las tareas, a excepción de la *primer tarea* que tiene la prioridad 0, y la *última tarea* que tiene la prioridad más baja en el Kernel.

4.1.4. Procesos o Tareas

Inicialmente el Kernel está configurado para trabajar con un máximo de 32 procesos. Sin embargo, el usuario, con base en sus necesidades, puede aumentar el número de los procesos a controlar por el Kernel. En cuanto al manejo de procesos, el Kernel trabaja con procesos estáticos, esto es, que desde la inicialización del sistema los procesos ya se encuentran definidos y no cambian durante la vida de éste. Dentro del Kernel, existen dos procesos importantes llamados: *primero y último* los cuales son activados al iniciar el Kernel.

[2] Por defecto ha sido inicializado con 15 niveles de prioridad en donde 0 es la máxima prioridad y 14 la menor.

Primer Proceso

El proceso primero es la primera tarea que se ejecuta en el kernel, tiene la mayor prioridad (*0*) y es la encargada de activar a los demás procesos o tareas que se van a ejecutar en el sistema. Este proceso antes de terminar su ejecución es eliminada del sistema y no se vuelve a ejecutar jamás en el Kernel (ver Tabla 4.2).

```
1.  void PrimerT()
2.    {  register int proc;
3.       //Activa todas las tareas en el kernel
4.       for (proc=1;proc<=NTAREAS;proc++)
5.        { activa(proc); }
6.       Elimina_Proceso(); //se elimina de la cola de procesos listos el mismo
7.       while(1)
8.        {   }
9.    }
```

Tabla 4.2: Código del Proceso "Primera Tarea".

Último Proceso

El proceso último es una tarea que tiene la finalidad de sólo ejecutarse cuando el kernel no tiene otra tarea por ejecutar. La prioridad de este proceso es la más baja que existe dentro del kernel (MAXPRIO-1) y como se puede apreciar en la Tabla 4.3, este proceso no tiene más que un ciclo infinito (que es el cuerpo de cualquier proceso) el cual sólo consume tiempo de procesador mientras no hay tareas listas para ejecutarse.

```
1.  void UltimaT()
2.    {
3.       while(1)
4.        {   }
5.    }
```

Tabla 4.3: Código del Proceso "Última Tarea".

4.1.5. BCP de los Procesos en el Kernel

Como se vio en la sección 2.7.6, cada proceso tiene asociada una estructura llamada Bloque de Control de Proceso (BCP) la cual contiene la información necesaria para el control del proceso. La estructura del BCP en el Kernel está representada en la Figura 4.2, en el cual se puede apreciar que la información está ordenada de la siguiente manera:

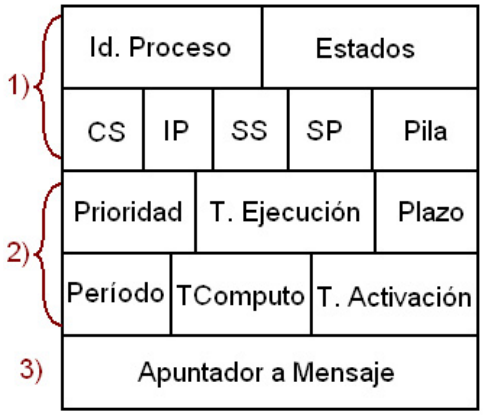

Figura 4.2: Bloque de Control de Procesos en el Kernel.

1) Información del proceso: Aquí se encuentra almacenado el identificador del proceso, el *estado* en que se encuentra durante la ejecución del Kernel (listo, suspendido, etc.), la dirección y el segmento de código inicial del proceso (*CS e IP*), el segmento y el puntero a los datos en la pila (*SS y SP*), y la *pila* que le corresponde al proceso. En la pila se almacena el contenido de sus registros (cs, ip, bp, di, si, ds, es, dx, cx, bx, ax y flags) así como las direcciones de las instrucciones que ejecuta el proceso. Los registros incluidos en el stack son indispensables para que el Kernel pueda llevar a cabo el *cambio de contexto*.

2) Información para el planificador: Aquí se encuentran los datos necesarios para que el proceso pueda ser planificado. Contiene información como la prioridad, el tiempo de cómputo, el plazo, el tiempo de activación y el tiempo de ejecución. Estos datos son necesarios para que las políticas de planificación puedan calcular cuando le corresponde ejecutarse a cada proceso.

3) Información de memoria utilizada en buzones: Sólo contiene un apuntador a una dirección de memoria reservada para almacenar un mensaje del buzón en caso de que el proceso sea introducido a la cola de procesos bloqueados por buzón (cpbb).

Físicamente, el bloque de control de procesos en el Kernel está representado por la estructura de la Tabla 4.4.

```
1.  struct BCP {
2.    unsigned estado;
3.    unsigned int Cs,Ip;
4.    unsigned char pila[tam_pila];
5.    unsigned prioridad;
6.    unsigned long tcc; // Tiempo de Computo Consumido
7.    unsigned long TA;  // Tiempo de Activación
8.    unsigned long Periodo,TComputo;
9.    unsigned long P; // Plazo
10.   struct MENSAJE PtrMje;
11. }PROCESO[MaxPro];
```

Tabla 4.4: Estructura del Bloque de Control de Procesos (BCP).

4.1.6. Estados de los Procesos en el Kernel

Dentro del Kernel, cada proceso puede encontrarse en cualquiera de los siguientes estados, tal como se describe en la Figura 4.3:

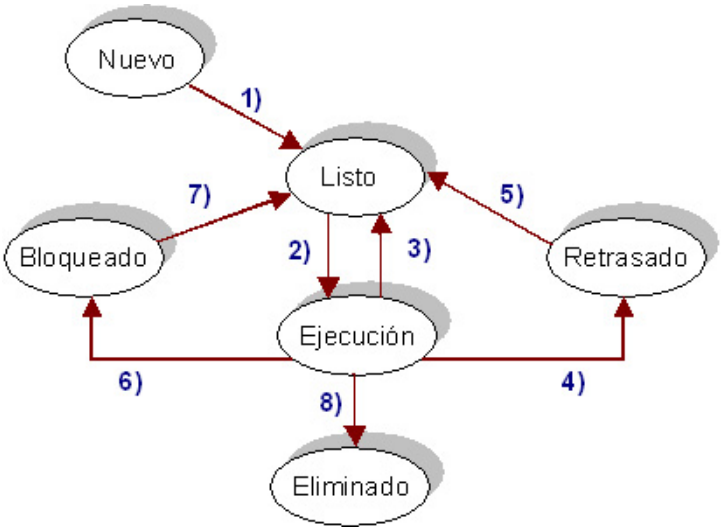

Figura 4.3: Estados de los Procesos en el Kernel.

Nuevo: Este estado es con el que empiezan los procesos al iniciar el Kernel. Esto indica que existe el proceso pero que no ha sido activado y, por lo tanto, no se encuentra listo para su ejecución.

Listo: En este estado, el proceso se encuentra listo para ejecutarse y espera a que se le asigne el procesador.

Ejecución: En este estado, el proceso tiene el control del procesador.

Bloqueado: En este estado, el proceso se encuentra bloqueado en algún recurso, como puede ser un semáforo, o un buzón.

Retrasado: En este estado el proceso se encuentra bloqueado, esperando por un tiempo determinado antes de regresar al estado de listo.

Eliminado: En este estado se encuentran los procesos que ya han terminado su ejecución y que ya no van a volver a utilizarse dentro del Kernel.

4.1.7. Transiciones entre Estados

Los procesos cambian de estado por alguna de las siguientes razones (ver Figura 4.3):

1. el proceso es creado y activado.
2. se le asigna el procesador.
3. se le quita el procesador debido a: un bloqueo, el alistamiento de un proceso con mayor prioridad o por el fin de su tiempo de cómputo.
4. el proceso se retrasa.
5. ocurre una vez que termina su tiempo de retraso, quedando listo para ejecutarse.
6. el proceso se bloquea por un recurso o interrupción.
7. obtuvo el recurso que esperaba y queda listo para ejecutarse.
8. el proceso se elimina a sí mismo.

4.1.8. Primitivas y Manejadores del Kernel

Para tener el control de todos los procesos y eventos que ocurren en el kernel, éste hace uso de las primitivas y los manejadores. Como se puede apreciar en la arquitectura del kernel (ver Figura 4.1), los manejadores se encuentran estrechamente relacionados con el hardware por lo que son los únicos que se pueden comunicar con él. En cuanto a las primitivas, éstas no se pueden comunicar con el hardware, pero sí con los procesos y los manejadores, esto quiere decir, que los procesos sólo pueden utilizar las primitivas, quienes a su vez se tienen que comunicar con los manejadores para poder utilizar parte del hardware.

4.2. Primitivas

4.2.1. Primitivas de Procesos

Aquí se definen las primitivas para el manejo de procesos como son: la primitiva "Activa", la primitiva "Elimina_Proceso" y la primitiva "Fin_de_Ciclo" (Figura 4.4). La estructura de

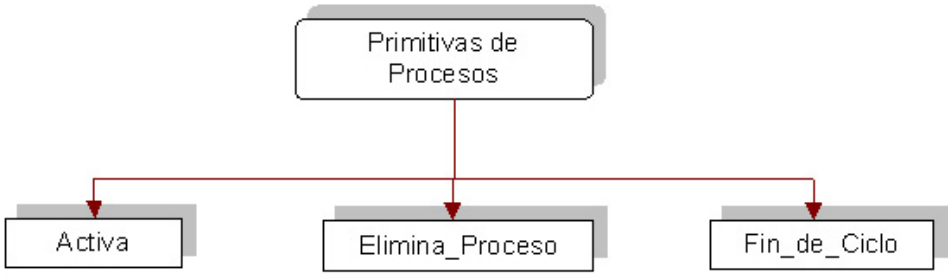

Figura 4.4: Primitivas de Procesos.

las primitivas de procesos son las siguientes:

Activa :

NOMBRE: `void activa(unsigned int num)`.

FUNCIÓN: Se encarga de activar un proceso para que se ejecute en el Kernel.

ENTRADAS: El identificador del proceso a activar (num).

SALIDAS: El proceso almacenado en la cola de listos.

Activa se encarga de indicar al Kernel que el proceso está listo para ejecutarse cambiándole su estado en el BCP a *listo* e insertándolo en la cola de procesos listos mediante la primitiva *inserta* del manejador de colas, la única condición que se pone para poder activar un proceso es, que el proceso que activa al nuevo proceso debe tener mayor prioridad (ver línea 7 de la Tabla 4.5).

Elimina_Proceso :

NOMBRE: `void Elimina_Proceso()`.

FUNCIÓN: Se encarga de eliminar al proceso que invoca esta primitiva, por lo que una vez eliminado, éste es incapaz de volverse a ejecutar en el Kernel.

ENTRADAS: El proceso en ejecución (idposeedor).

SALIDAS: El proceso eliminado del Kernel.

```
1.  void activa(unsigned int num)
2.    { //El numero del proceso debe estar dentro del rango permitido
3.      if (num>=MaxPro)
4.        ERROR(7);
5.      else
6.       { //El proceso que lo activa debe tener mayor prioridad
7.         if(PROCESO[idposeedor].prioridad <= PROCESO[num].prioridad)
8.           { //En el kernel el que tenga un numero menor es el que tiene
9.             //mayor prioridad, Ejemplo: el de prioridad=1 es mayor que
10.             //el que tenga prioridad=10.
11.             PROCESO[num].estado=LISTO;
12.            Inserta(cpl,PROCESO[num].prioridad,num);
13.          }
14.        else
15.          ERROR(8);
16.      }
17.    }
```

Tabla 4.5: Primitiva Activa.

La primitiva **Elimina_Proceso** le indica al Kernel que el proceso ha terminado de realizar todas sus funciones y que ya no va a ser útil de nuevo en el sistema. Para esto, le cambia su estado a *eliminado* en el BCP del proceso y lo saca de la cola de procesos listos mediante la primitiva *elimina* del manejador de colas. La primitiva Elimina_Proceso sólo la puede llamar el proceso en ejecución (IdPoseedor) y por lo tanto al finalizar la primitiva se invoca al cambio de contexto para que se le asigne el procesador a otro proceso (ver la línea 7 de la Tabla 4.6).

```
1.  void Elimina_Proceso()
2.    { {disable();
3.      // cambia el estado del proceso a ELIMINADO
4.      PROCESO[idposeedor].estado= ELIMINADO;
5.      Elimina(cpl,idposeedor);  //Saca el proceso de la cola de activos
6.      enable();
7.      CambiaContexto();  // Llama al cambio de contexto
8.    }
```

Tabla 4.6: Primitiva Elimina Proceso.

Fin_de_Ciclo :

NOMBRE: `void Fin_de_Ciclo()`.

FUNCIÓN: Indica al Kernel que una tarea ya cumplió con su ciclo de ejecución en un período.

ENTRADAS: Ninguna.

SALIDAS: Ninguna.

La primitiva **Fin_de_Ciclo** es utilizada para indicar al Kernel que la tarea ya cumplió con el ciclo de ejecución y que no necesita ejecutarse nuevamente el código hasta su siguiente tiempo de activación.

```
1.  void Fin_de_Ciclo()
2.    {  PROCESO[idposeedor].FC = VERDADERO;
3.       while (PROCESO[idposeedor].FC==VERDADERO)
4.         {  // Se mantiene aqui hasta que inicie su tiempo de activación }
8.    }
```

Tabla 4.7: Primitiva *Fin de Ciclo* de la Tarea.

4.2.2. Primitivas de Tiempo

El Kernel permite que un proceso pueda retrasarse por un lapso de tiempo determinado. Para lograrlo se hace uso de la primitiva de tiempo "Retrasa" (ver Figura 4.5). Las estructuras

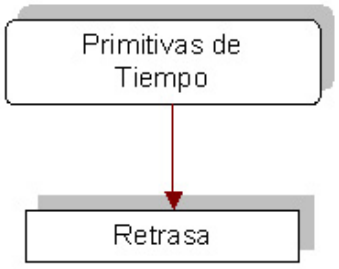

Figura 4.5: Primitivas de Tiempo.

de estas primitivas de tiempo son las siguientes:

Retrasa :

NOMBRE: `void retrasa(unsigned long tiempo)`.

FUNCIÓN: Retrasa el proceso que la llama por un tiempo determinado en milisegundos. Este proceso es sacado de la cola de listos y almacenado en la cola de procesos retrasados.

ENTRADAS: El identificador de proceso a retrasar (idposeedor) y el tiempo (tiempo) que va a durar en la cola de retrasados. El rango de retraso es de 1 hasta 4,294,967,295 milisegundos.

SALIDAS: El identificador de proceso a ejecutarse.

La primitiva **Retrasa** permite que un proceso pase al estado de retrasado por un tiempo determinado. Para lograrlo, el proceso es eliminado de la cola de procesos listos e insertado en la cola de procesos retrasados y su estado es cambiado a "retrasado". Por último, como esta primitiva es llamada por el proceso en ejecución, al terminar de ejecutarse se invoca al cambio de contexto para que otro proceso pueda hacer uso del procesador (ver Tabla 4.8).

```
1.  void retrasa(int tiempo)
2.    { disable();
3.      //Si el tiempo es menor a 1...
4.      if(tiempo<1)
5.        // tiempo invalido
6.        ERROR(6);
7.      else
8.       {//saca el proceso de la cola de activos
9.         //Saca el proceso de la cola de procesos listos
10.        Elimina(cpl,idposeedor);
11.        // y lo inserta en la cola de retrasados
12.        inserta_r(idposeedor,tiempo);
13.        PROCESO[idposeedor].estado=RETRASADO;
14.      }
15.     enable();
16.     CambiaContexto();
17.   }
```

Tabla 4.8: Primitiva Retrasa.

4.2.3. Primitivas de Semáforos

Las cuatro primitivas utilizadas en el Kernel para manejar los semáforos son: "CreaSem", "IsColSemVacia", "Espera" y "Senial", las cuales se pueden ver Figura 4.6.

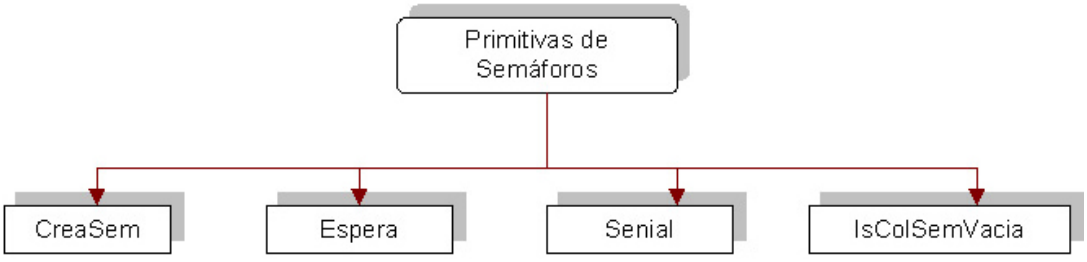

Figura 4.6: Primitivas de Semáforo.

Iniciar Semáforo: En esta primitiva se le da un valor inicial (positivo) a la variable del semáforo, con lo cual se crean una cola asociada al semáforo. En esta cola, se incluirán los procesos bloqueados por este recurso.

CreaSem :

NOMBRE: `void CreaSem(unsigned int idSem,unsigned int valor, char nomsem[])`.

FUNCIÓN: Crea un semáforo determinado, dándole un nombre y valor inicial.

ENTRADAS: El identificador del semáforo a crear (idSem), un valor inicial (valor: 0 o 1 para binarios) y el nombre del semáforo (nomsem).

SALIDAS: El semáforo creado y listo para utilizarse.

```
1.  void CreaSem(unsigned int idSem,unsigned int valor,char nomsem[])
2.    {  disable();
3.       if(idSem >= MaxSem)
4.         ERROR(9);
5.       else
6.         { if(valor>CuentaMaxima)  ERROR(10);
7.           else
8.             { if( SEMAFORO[idSem].creado == VERDADERO)
9.                 ERROR(11);
10.              else
11.               { SEMAFORO[idSem].cont=valor;
12.                 SEMAFORO[idSem].Tam=0;
13.                 SEMAFORO[idSem].creado=VERDADERO;
14.                 CopiaCadena(SEMAFORO[idSem].nombre,nomsem);  // se nombra al semáforo
15.               }
16.            }
17.         }
18.     enable();
19.    }
```

Tabla 4.9: Primitiva *Crea Semáforo*

Señal: Esta primitiva permite incrementar en una unidad, a la variable del semáforo. Si la variable del semáforo es negativa indicará que existe algún proceso bloqueado (en la cola de este semáforo). Por lo tanto si al ejecutar la primitiva señal, el contador es negativo, algún proceso bloqueado en la cola de este semáforo (el que se encuentre al principio de la cola), se pasará a la cola de procesos listos. En este caso, el Kernel provocará un cambio de contexto para verificar si el proceso desbloqueado, es de mayor prioridad que el proceso en ejecución. Si al ejecutar la primitiva señal no existen procesos bloqueados (el contador es cero o positivo), se incrementa el contador del semáforo y el proceso continuará su ejecución.

Senial :

NOMBRE: `void Senial(unsigned int idSem)`.

FUNCIÓN: Manda una señal al semáforo a desocupar. Si existen procesos bloqueados, se alista al de mayor prioridad.

ENTRADAS: El identificador de semáforo a desocupar.

SALIDAS: El semáforo libre y disponible para su uso.

```
1.  void Senial(unsigned int idSem)
2.    { unsigned int idProc;
3.      unsigned int proc;
4.      disable();
5.       if( idSem >= MaxSem)
6.        {
7.          ERROR(9);
8.        }
9.       else
10.        { if(SEMAFORO[idSem].creado == FALSO)
11.            {
12.              ERROR(12);
13.            }
14.          else
15.            {  //Si la cola esta vacia...
16.               if(IsColSemVacia(idSem))
17.               //Sólo aumenta el contador del semáforo
18.               SEMAFORO[idSem].cont++;
19.              else
20.                 //Si no, alista al de mayor prioridad
21.                { //Busca al de mayor prioridad
22.                  idProc=BuscaMayor(SEMAFORO[idSem].cpbs);
23.                  //Lo saca de la cola de procesos bloqueados por semáforo
24.                  proc=Saca(SEMAFORO[idSem].cpbs,PROCESO[idProc].prioridad);
25.                  // y lo inserta en la cola de procesos listos
26.                  Inserta(cpl,PROCESO[proc].prioridad,proc);
27.                  //Actualiza el conteo de los procesos bloqueados por semáforo
28.                  SEMAFORO[idSem].Tam--;
29.                  //Se actualiza su estado
30.                  PROCESO[proc].estado = LISTO;
31.                  //Obliga a un cambio de contexto
32.                  CambiaContexto();
33.                }
34.             }
35.        }
36.     enable();
37.    }
```

Tabla 4.10: Primitiva *Señal*

Espera: Esta primitiva permite decrementar en una unidad a la variable del semáforo. Si después de decrementar esta variable, su valor es negativo, el proceso que ejecuta esta primitiva es enviado a la cola del semáforo y sacado de ejecución. En este caso, el Kernel invocará a la rutina de cambio de contexto para ejecutar al siguiente proceso de la cola de listos. Por el contrario, si después de ejecutar la primitiva espera, la variable del semáforo contiene un valor cero o positivo, el proceso continuará su ejecución.

Espera :

NOMBRE: `void Espera(unsigned int idSem)`.

FUNCIÓN: Señaliza el semáforo a ocupar, si esta vacío le da el recurso, esto es, continúa su ejecución sin problemas, pero si estaba ocupado lo bloquea.

ENTRADAS: El identificador de semáforo a utilizar.

SALIDAS: El semáforo ocupado.

```
1.  void Espera(unsigned int idSem)
2.    { disable();
3.       if( idSem >= MaxSem)
4.         ERROR(9);
5.       else
6.         { if(SEMAFORO[idSem].creado == FALSO)
7.             ERROR(12);
8.           else
9.             { SEMAFORO[idSem].cont--;
10.              if(SEMAFORO[idSem].cont < 0) //Si la cola no esta vacía
11.                { //Bloquea el proceso que pidió el semáforo
12.                  //eliminándolo de la cola de procesos listos
13.                  Elimina(cpl,idposeedor);
14.                  //e insertándolo en la cola de procesos bloqueados por semáforo
15.                  Inserta(SEMAFORO[idSem].cpbs,PROCESO[idposeedor].prioridad,idposeedor);
16.                  SEMAFORO[idSem].Tam++; //lleva el conteo de los procesos bloqueados por semáforo
17.                  PROCESO[idposeedor].estado = BLOQUEADO; //Se actualiza su estado
18.                  CambiaContexto(); //Obliga a un cambio de contexto
19.       }}}
20.     enable();
21.    }
```

Tabla 4.11: Primitiva *Espera*

IsColSemVacia :

NOMBRE: `unsigned int IsColSemVacia(unsigned int idSem)`.

FUNCIÓN: Se encarga de revisar si la cola de procesos bloqueados por semáforo está vacía, o no.

ENTRADAS: El identificador del semáforo a revisar.

SALIDAS: FALSO si la cola tiene elementos o VERDADERO si está vacía.

En la Figura 4.7 se muestran 2 tareas haciendo acceso de un semáforo para implementar regiones críticas. El uso de estas regiones críticas permite a cada proceso accesar una variable compartida en forma exclusiva.

```
1.  unsigned int IsColSemVacia(unsigned int idSem)
2.    { disable();
3.      if(SEMAFORO[idSem].Tam > 0)
4.        return FALSO;//cola con elementos
5.      else
6.        return VERDADERO; //cola vacia
7.      enable();
8.    }
```

Tabla 4.12: Primitiva que revisa si la Cola de Semáforos está vacía.

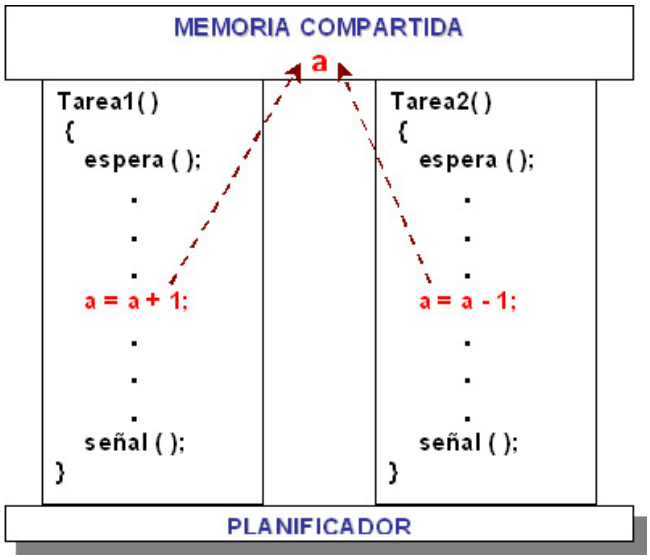

Figura 4.7: Utilización de los semáforos en el Kernel.

4.2.4. Primitivas de Buzón

Las primitivas de buzón, como se muestran en la Figura 4.8 son: "CreaBuzon", "Recibir-Mje" y "EnviarMje".

Crear Buzón: Esta primitiva permite crear la estructura de datos que define al buzón y su cola de procesos asociada.

CreaBuzon :

NOMBRE: `void CreaBuzon(unsigned int IdProceso,unsigned int IdBuzon).`

FUNCIÓN: Crea un buzón y lo inicializa indicando que ya ha sido creado y anotando el identificador del proceso que lo creó.

ENTRADAS: El identificador del buzón a crear (unsigned int IdBuzon) y el identificador del proceso que lo desea crear (IdProceso).

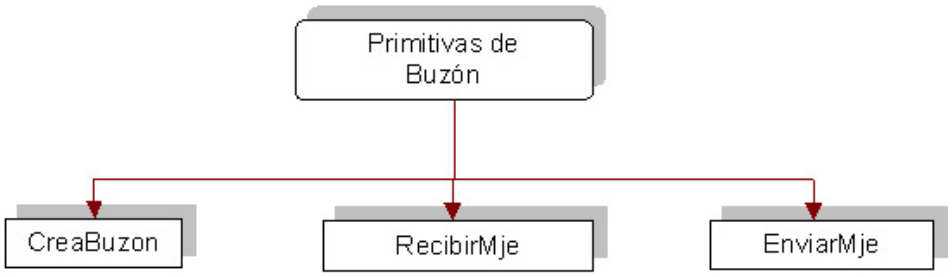

Figura 4.8: Primitivas de Buzón.

SALIDAS: El Buzón creado o en caso de fallo, un identificador con la descripción del error generado.

```
1.  void CreaBuzon(unsigned int IdProceso,unsigned int IdBuzon)
2.    { disable();
3.      if (IdBuzon >= MaxBuz)
4.        ERROR(14);
5.      else
6.        {if (dcb[IdBuzon].Creado)
7.           ERROR(15);
8.         else // No hubo errores por lo tanto se crea el buzón
9.           { dcb[IdBuzon].Creado=1;
10.            dcb[IdBuzon].BuzProceso=IdProceso;
11.          }
12.       }
13.      enable();
14.    }
```

Tabla 4.13: Primitiva *Crea Buzón*

Enviar Mensaje: Mediante esta primitiva es posible enviar un mensaje en formato ASCII al buzón indicado (el cual previamente tuvo que haber sido creado). Si al enviar un mensaje alguna tarea se encontraba bloqueada esperando mensajes, esta tarea será desbloqueada y enviada a la cola de procesos listos. Si un proceso envía un mensaje al buzón y el buzón se encuentra lleno, provocará que el proceso se incluya en la cola del buzón. Este proceso permanecerá en esta cola hasta que otro proceso reciba mensajes y libere suficiente espacio del buzón.

EnviarMje :

NOMBRE: `void EnviarMje (unsigned int IdBuzon,struct MENSAJE Mensaje)`.

FUNCIÓN: Envía un mensaje a un buzón determinado.

ENTRADAS: El identificador del buzón donde se va a almacenar el mensaje (IdBuzon) y el mensaje a enviar.

SALIDAS: El mensaje almacenado en el buzón o en caso de error, un identificador con la explicación del fallo.

```
1.  void EnviarMje (unsigned int IdBuzon,struct MENSAJE Mensaje)
2.    { unsigned int Ultimo,IdProceso,Prio,Proc;
3.      disable();
4.      if (IdBuzon >= MaxBuz) ERROR(14);//El idBuzon debe estar en el rango permitido
5.      else
6.        {if (dcb[IdBuzon].Creado==0) ERROR(16);// El buzón debe estar creado
7.         else  // Si los buzones cumplen los requisitos de arriba continúa...
8.           {IdProceso=Mensaje.ProCreo; //Obtiene el Id del proceso que creo el mensaje
9.            Prio=PROCESO[IdProceso].prioridad; //Obtiene la prioridad del proceso
10.           if (dcb[IdBuzon].UltMje==MaxMje) //Comprueba si hay espacio en la cola de buzones
11.             //No hay espacio para el mensaje por lo que se envía a la cola de Bloqueados por Buzón.
12.             {PROCESO[IdProceso].PtrMje=Mensaje; //Copia al PCB del proceso a bloquear
13.              Elimina(cpl,IdProceso); // sacar proceso de la cola de procesos listos
14.              Inserta(cpbb[IdBuzon],Prio,IdProceso); //y lo mete a cpbb.
15.              PROCESO[IdProceso].estado=BLOQUEADO;
16.              CambiaContexto(); // Llama al cambio de contexto
17.             }
18.            else // Hay espacio para el mensaje dentro del buzón.
19.             { Ultimo=dcb[IdBuzon].UltMje;  //Obtiene la posición del último mensaje
20.               dcb[IdBuzon].Mensajes[Ultimo]=Mensaje; //Copia la estructura del Mensaje Completo
21.               dcb[IdBuzon].UltMje++;
22.             }//Busca si hay procesos bloqueados por intentar recibir mensaje cuando no había
23.            Proc=BuscaMayor(cpbb[IdBuzon]);
24.            if (Proc!=0) //Si había bloqueados, saca al de mayor prioridad
25.              { IdProceso=Saca(cpbb[IdBuzon],PROCESO[Proc].prioridad);
26.                Inserta(cpl,PROCESO[IdProceso].prioridad,IdProceso); //y lo mete a cpl
27.                PROCESO[Proc].estado=LISTO;
28.                CambiaContexto(); // Llama al cambio de contexto
29.               }
30.           }
31.       }
32.      enable();
33.    }
```

Tabla 4.14: Primitiva *Envía Mensaje*

Recibir Mensaje.- Esta primitiva permite recibir mensajes del buzón indicado. Si el buzón se encuentra vacío, el proceso que invoca a esta primitiva será bloqueado en la cola respectiva. Note que la cola del buzón contendrá sólo a procesos bloqueados esperando recibir mensajes, o sólo a procesos esperando enviar mensajes al buzón. Por su implementación, los dos tipos de procesos no pueden coexistir en una misma cola.

RecibirMje :

NOMBRE: `void RecibirMje (unsigned int IdBuzon, struct MENSAJE *Mensaje)`.

FUNCIÓN: Recibe un mensaje desde un buzón determinado.

ENTRADAS: El identificador del buzón donde se va a recibir el mensaje (IdBuzon) y un puntero con la dirección donde se va a almacenar el mensaje recibido.

SALIDAS: El mensaje recibido o en caso de error, un identificador con la explicación del fallo ocurrido.

```
1.  void RecibirMje (unsigned int IdBuzon, struct MENSAJE *Mensaje)
2.    { unsigned int Ultimo,i,Proc,IdProceso,Prio;
3.      disable();
4.      if (IdBuzon >= MaxBuz) ERROR(14); //El idBuzon debe estar en el rango permitido
5.      else
6.        {if (dcb[IdBuzon].Creado==0) ERROR(16);// El buzón debe estar creado
7.         else // Si los buzones cumplen los requisitos de arriba continúa...
8.           {Prio=PROCESO[idposeedor].prioridad; //Obtienen la prioridad del proceso
9.            if (dcb[IdBuzon].UltMje==0) // Si no hay mensajes en el buzón...
10.             { // Bloquear proceso en espera del mensaje.
11.               Elimina(cpl,idposeedor); // sacándolo de la cola de procesos listos
12.               Inserta(cpbb[IdBuzon],Prio,idposeedor); //y metiéndolo a cpbb
13.               PROCESO[idposeedor].estado=BLOQUEADO;
14.               CambiaContexto(); // Llama al cambio de contexto
15.             }
16.           *Mensaje=dcb[IdBuzon].Mensajes[0]; // Hay mensajes, sacar el primero de la cola
17.           for (i=0;i<MaxMje-1;i++) // Recorrer los mensajes de la cola dcb
18.             dcb[IdBuzon].Mensajes[i] = dcb[IdBuzon].Mensajes[i+1];
19.           IdProceso=BuscaMayor(cpbb[IdBuzon]); //Busca al de mayor prioridad bloqueado en IdBuzon
20.           if (IdProceso == 0) // Comprobar si hay elementos en la cola cpbb, 0 NO HAY
21.             { if (dcb[IdBuzon].UltMje == MaxMje) //Si estaba lleno dcb pero no se pasaba
22.                 dcb[IdBuzon].Mensajes[MaxMje-1] = MjeLimpio; //limpiar la última posición
23.               dcb[IdBuzon].UltMje--; //Indicar que hay un mensaje menos
24.             }
25.           else //Hay procesos bloqueados en cpbb, por lo tanto...
26.             { Proc=Saca(cpbb[IdBuzon],PROCESO[IdProceso].prioridad); // Saca al de mayor prioridad
27.               Inserta(cpl,PROCESO[Proc].prioridad,Proc); //y lo mete a la cola de procesos listos
28.               PROCESO[idposeedor].estado=LISTO;
29.               dcb[IdBuzon].Mensajes[MaxMje-1]=PROCESO[Proc].PtrMje; //Copia el Mensaje del PCB
30.               PROCESO[Proc].PtrMje = MjeLimpio; //Limpia el Mensaje del PCB
31.               CambiaContexto(); // Llama al cambio de contexto
32.             }
33.           }
34.         }
35.      enable();
36.    }
```

Tabla 4.15: Primitiva *Recibir Mensaje*

4.3. Manejadores

4.3.1. Manejo de Registros

El manejador de registros contiene los procedimientos necesarios para controlar los registros y los datos en memoria interna. Las operaciones aquí realizadas sólo se ejecutan una vez (al iniciar el Kernel) y son fundamentales ya que preparan al sistema para que éste pueda trabajar en la máquina permitiéndole realizar correctamente los cambios de contexto entre los procesos. La Figura 4.9 nos muestra los procedimientos que contiene el manejador de registros.

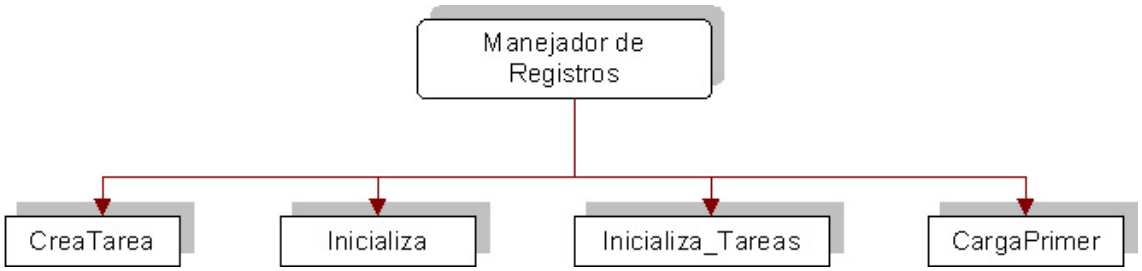

Figura 4.9: Procedimientos del Manejador de Registros.

CreaTarea :

NOMBRE: `void CreaTarea(void (*DirTarea)(),unsigned id,unsigned prio, unsigned long TC, unsigned long P)`.

FUNCIÓN: Inicia los registros de la tarea indicada, obtiene los valores del SS y SP, el CS y el IP del proceso, llena la pila con los registros de la máquina en el siguiente orden: FLAGS, CS, IP, AX, BX, CX, DX, ES, DS, SI, DI, BP y además asigna la prioridad, el tiempo de computo y el período al proceso para que este pueda ser planificado.

ENTRADAS: La dirección de la tarea a crear, su identificador, prioridad, período y tiempo de computo.

SALIDAS: Una tarea con pila y registros listos para ser utilizados.

Inicializa :

NOMBRE: `void inicializa()`.

FUNCIÓN: Se encarga de inicializar todas las estructuras de datos que se van a utilizar

```
1.  void CreaTarea(void (*DirTarea)(),unsigned id,unsigned prio,unsigned long TC, unsigned long P)
2.    {  unsigned int Iip,Ics,ssI,spI,banderas;
3.       disable();
4.       if (id>=MaxPro) ERROR(7); //se verifica que el identificador del proceso sea válido
5.       banderas=0x200; //Guarda el valor de las banderas para que permitan interrupciones
6.       ssI = _SS; spI = _SP; // Se guardan los valores de la pila principal
9.       _SS = PROCESO[id].ss = FP_SEG (&PROCESO[id].pila);
10.      _SP = PROCESO[id].sp = FP_OFF (&PROCESO[id].pila+(tam_pila-2));
11.      Ics = FP_SEG(DirTarea);  // Se obtiene la dirección IP y CS del proceso
12.      Iip = FP_OFF(DirTarea);
13.       asm {  push banderas // Se introducen todos los registro a la pila de cada proceso...
14.              push Ics //   Esto se hace debido a que la interrupción del reloj
15.              push Iip // al activarse, mete a la pila todos los registros en el
16.              push ax  // siguiente orden: AX,BX,CX,DX,ES,DS,SI,DI y BP.
17.              push bx  //   Al terminar la interrupción se sacan los registros
18.              push cx  // anteriores y además se ejecuta un IRET el cual saca
19.              push dx  // de la pila el IP, CS y FLAGS en ese orden.
20.              push es
21.              push ds
22.              push si
23.              push di
24.              push bp }
26.       PROCESO[id].ss = _SS; // Debido a que hubo push en la pila los registros SS y SP
27.       PROCESO[id].sp = _SP; //  cambiaron, por lo que es necesario guardarlos...
28.       _SP = spI; _SS = ssI;// cambia la pila por la pila inicial
30.       PROCESO[id].prioridad=prio; //Se asigna la prioridad
31.       PROCESO[id].Periodo  =  P;  // El periodo
32.       PROCESO[id].TComputo =  TC;  // y el tiempo de computo
33.       PROCESO[id].P=P; //Siguiente Periodo o TA //CAMBIAR a TA checarlo
34.     enable();
35.    }
```

Tabla 4.16: Primitiva *Crea Tarea*

en el Kernel, estas son, la cola de procesos listos, la de buzones, la de semáforos y cola la de procesos retrasados. Además crea un semáforo 0 destinado para aquellos procesos o tareas de deseen entrar en "Región Critica".

ENTRADAS: Ninguna.

SALIDAS: Las estructuras inicializadas y listas para utilizarse.

Inicializa_Tareas :

NOMBRE: `void Inicializa_Tareas(void)`.

FUNCIÓN: Inicializa la estructura de control de las tareas (BCP) poniendo su estado a "nuevo".

ENTRADAS: Ninguna.

SALIDAS: Ninguna.

CargaPrimer :

```
1.  void inicializa()
2.    {  register int i;
3.       disable();
4.       Inicializa_Tareas(); //Inicializa el PCB de las tareas
5.       InicializaCola(cpl);// Inicializa la cola de procesos listos (activos)
6.       inicializa_r();//inicializa la cola de procesos retrasados
7.       for (i=0;i<MaxSem;i++)//inicializa la cola de semáforos
8.         InicializaCola(SEMAFORO[i].cpbs);
9.       for (i=0;i<MaxBuz;i++)//inicializa la cola de procesos bloqueados por buzón
10.        InicializaCola(cpbb[i]);
11.      CreaSem(0,0,"Region Critica");//Semáforo utilizado para entrar en región critica
12.      for (i=1;i<MaxPro;i++)
13.        { PROCESO[i].tcc=0; //Tiempo de Computo Completado
14.          PROCESO[i].TA=0;  //Tiempo de Activación
15.        }
16.      enable();
17.    }
```

Tabla 4.17: Primitiva *Inicializa*

```
1.  void Inicializa_Tareas(void)
2.    { register int i;
3.      disable();
4.      for (i=0;i<MaxPro;i++)
5.        PROCESO[i].estado=NUEVO;
6.      enable();
7.    }
```

Tabla 4.18: Primitiva *Inicializa Proceso*

NOMBRE: `void interrupt CargaPrimer(void)`.

FUNCIÓN: Pone en memoria la pila de la primera tarea con el objetivo de que ésta se ejecute primero. Además, intercambia la rutina del teclado (0x09h) y el reloj (0x08h) por las propias del Kernel.

ENTRADAS: Ninguna.

SALIDAS: Ninguna.

4.3.2. Manejo del Procesador

Este manejador sólo tiene un procedimiento llamado *CambiaContexto* (ver Figura 4.10) y sus características son las siguientes:

CambiaContexto :

NOMBRE: `void interrupt CambiaContexto(void)`.

FUNCIÓN: Se encarga de asignarle el procesador a una tarea con base en la decisión del planificador.

ENTRADAS: Ninguna, al iniciar contiene el identificador del proceso a ser reemplazado.

```
1.  void interrupt CargaPrimer(void)
2.    {
3.       disable();
4.       old_IntTeclado=getvect(0x09);
5.       old_IntReloj=getvect(0x08); //Cambia al planificador basado en el timer
6.       setvect(0x09,IntTeclado);
7.       setvect(0x08,CambiaContexto);
8.       // Graba el segmento y el puntero de la pila original para que cuando
9.       // finalice el kernel, la ejecución del DOS pueda continuar donde se
10.      // quedo al iniciar el kernel.
11.      oldss=_SS;
12.      oldsp=_SP;
13.      // Cambia la pila por la pila de la primer tarea
14.      _SS=PROCESO[idposeedor].ss;
15.      _SP=PROCESO[idposeedor].sp;
16.      enable();
17.    }
```

Tabla 4.19: Primitiva *Carga Primer*

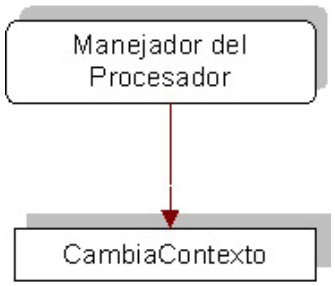

Figura 4.10: Procedimiento del Manejador del Procesador.

SALIDAS: Devuelve el identificador de la tarea que estará en ejecución.

La rutina es de tipo interrupción y es la que reemplazó a la int 0x08h dedicada a controlar la hora. Es periódica, y de manera automática se activa cada milisegundo y como se dijo arriba, es la que asigna el procesador a las tareas. El primer paso que se realiza al entrar a este procedimiento, es revisar si es necesario despertar a alguna tarea suspendida con la finalidad de que sea tomada en cuenta dentro de la planificación. El siguiente paso es encontrar la siguiente tarea a la que se va a asignar el procesador. Esto lo realiza el planificador[3], quien devuelve mediante la variable *SigTar* el identificador de la siguiente tarea a ejecutar (ver sección 4.3.4 para saber con más detalle como se planifica). Por último, existe una variable

[3]El Kernel cuenta con tres mecanismos de planificación (FIFO Round Robin, EDF y Rate Monotonic) pero está diseñado para soportar nuevos mecanismos de planificación sin que sea necesario realizar muchos cambios al código

llamada *IdPoseedor* la cual indica el proceso que tiene asignado el CPU en ese instante. IdPoseedor es comparada con SigTar y si la siguiente tarea a ejecutar (*SigTar*) es diferente a IdPoseedor, entonces se realiza el **cambio de contexto**, si eran iguales no hace nada y continúa la ejecución de la tarea.

Cambio de Contexto

Para lograr el cambio de contexto en el Kernel, es necesario guardar la pila de la tarea actual y cambiar la pila del sistema por la del nuevo proceso a ejecutar. Para guardar la pila basta con copiar las direcciones del SP (Stack Pointer) y del SS (Stack Segment) al BCP del proceso. Para cambiar la pila del sistema por la del nuevo proceso a ejecutar sólo se necesita copiar los valores del SS y SP almacenados en el BCP del nuevo proceso a la dirección SS y SP del sistema (ver Tabla 4.20).

```
1. // Revisa si es necesario realizar el cambio de contexto
2. if (SigTar!=idposeedor) //Si es diferente la tarea hace el cambio
3.   { //Guarda la pila de la tarea actual
4.     PROCESO[idposeedor].ss=_SS;
5.     PROCESO[idposeedor].sp=_SP;
6.     PROCESO[idposeedor].estado=LISTO; // se cambia a listo su estado
7.     // Cambia la pila por la de la nueva tarea\\
8.     _SS=PROCESO[SigTar].ss;
9.     _SP=PROCESO[SigTar].sp;
10.    PROCESO[SigTar].estado=EJECUCION; //Cambia su estado a CORRIENDO o EJECUCION
11.    idposeedor=SigTar; //Actualiza Idposeedor
12. }
```

Tabla 4.20: Código del Cambio de Contexto.

4.3.3. Manejo de Colas de Procesos

El Kernel está diseñado para ejecutarse sobre máquinas que contienen un solo procesador. Esto significa que únicamente un proceso puede estar ejecutándose a la vez y que los demás procesos pueden estar bloqueados por recursos, o listos esperando en una cola. Para el control de los procesos, se establecieron las siguientes colas:

1. Cola de procesos listos.
2. Cola de procesos bloqueados por semáforos.
3. Cola de procesos bloqueados por buzón (ya sea por enviar o por recibir un mensaje).

4. Cola de procesos retrasados.

Para controlar los procesos almacenados en las colas del Kernel, fue necesario crear rutinas especializadas que permitieran manipular estas colas (ver Figura 4.11). Las colas de procesos

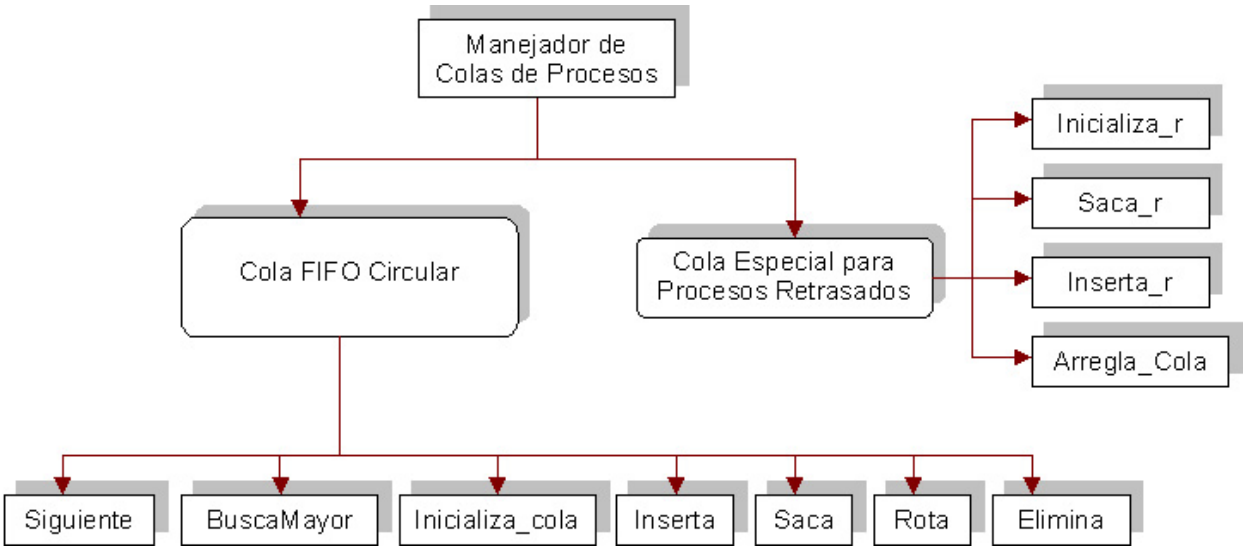

Figura 4.11: Procedimientos del Manejador de Colas de Procesos.

fueron diseñadas con el fin de manejar eficientemente la ejecución de las tareas de acuerdo a su estado. Las tres primeras colas tienen una estructura similar con múltiples niveles de prioridades, como se muestra en la Figura 4.12. La primer tarea en la cola es aquella que cuenta con la mayor prioridad en el sistema. En nuestro Kernel, asumimos que los procesos se insertan al final de la cola respectiva dependiendo de su prioridad y su arribo en la cola. En la cola de procesos retrasados es totalmente diferente, depende en su mayoría del tiempo que tienen que esperar los procesos en esta cola y no del orden de llegada o la prioridad que éstos tengan.

Todas las colas de procesos utilizadas en el Kernel a excepción de la cola de procesos retrasados, utilizan la estructura de la Tabla 4.21. Esta estructura, representa una cola cuya política de funcionamiento es denominada FIFO (First In First Out), es decir, el primer elemento en entrar es el primer elemento en salir. Dentro de la estructura se encuentran definidas algunas variables útiles para el control de la cola como son: Tam que permite saber cuantos elementos tiene la cola, $Inicial$ que contiene el índice del primer elemento en la cola,

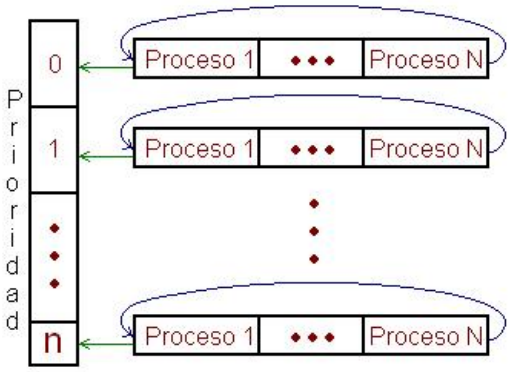

Figura 4.12: Colas de Procesos.

Final que contiene el índice del último elemento almacenado en la cola y $Proceso[MaxPro]$ que es un arreglo secuencial en donde se almacenan los identificadores de los procesos que están en la cola. La cantidad de procesos que puede tener la cola son 32 ($MaxPro$), sin embargo esta cantidad puede variar al modificar el archivo de configuración (ver sección 4.4.1).

```
1. struct COLA {
2.   unsigned int Tam; // Lleva  el conteo de los elementos en la cola
3.   unsigned int Inicial; //Un puntero al primer elemento de la cola
4.   unsigned int Final; // Un puntero al último elemento de la cola
5.   unsigned int Proceso[MaxPro]; // Lugar donde se almacena el Id
6. };                              //de los procesos que están en la cola
```

Tabla 4.21: Estructura utilizada para las colas de los procesos.

Cola de Procesos Listos (CPL)

Esta cola tiene una estructura igual a la mostrada en la Figura 4.12, y las razones por las que un proceso se puede encontrar en la cola de procesos listos son:

1. Al iniciar el sistema, ya que la mayoría de los procesos están listos para su ejecución.

2. Cuando termina su tiempo de cómputo.

3. Cuando sale de la cola de retrasados debido a que terminó su tiempo de retraso y está listo para ejecutarse.

4. Cuando sale de algún bloqueo (ya sea por semáforo o por buzón) debido a que obtuvo el recurso que esperaba y quedó listo para ejecutarse.

La manera en que está declarada la cola de procesos listos en el kernel es:

```
struct COLA cpl[MaxPrio];
```

donde COLA es la estructura general vista en la Tabla 4.21.

Cola de Procesos Bloqueados por Semáforos (CPBS)

La cola de procesos bloqueados por semáforo tiene una estructura igual a la mostrada en la Figura 4.12 y físicamente en el kernel se encuentra declarada como se ve en la Tabla 4.22. La razón de que la cola de procesos bloqueados por semáforo esté declarada dentro de

```
1. struct SEM {
2.   int cont;
3.   int Tam; //Número de procesos bloqueados
4.   char creado; // 1 VERDADERO 0 FALSO
5.   char nombre[15];
6.   struct COLA cpbs[MaxPrio]; //Cola de procesos bloqueados por semáforos
7. } SEMAFORO[MaxSem]; //Estructura completa del semáforo
```

Tabla 4.22: Declaración de la Cola de Procesos Bloqueados por Semáforo (CPBS).

la estructura del semáforo (ver línea 6 de la Tabla 4.22), es porque debe existir una cola por cada semáforo que exista en el sistema. Un proceso puede estar dentro de este tipo de cola debido a que deseaba utilizar un recurso administrado por un semáforo, y le fue negado porque ese recurso ya era utilizado por otro proceso. El proceso puede salir de esta cola sólo cuando el recurso que esté esperando se haya liberado.

Cola de Procesos Bloqueados por Buzón (CPBB)

La estructura de la cola de procesos bloqueados por buzón al igual que las otras dos colas vistas, tiene la misma estructura mostrada en la Figura 4.12 y las razones por las que un proceso se puede encontrar en la cola de procesos bloqueados por buzón son:

1. Al tratar de enviar un mensaje y no poder hacerlo debido a que el buzón está lleno (no hay espacio para el mensaje).

2. Cuando desea recibir un mensaje y el buzón se encuentra vacío (no hay mensajes en el buzón).

La cola de procesos bloqueados por buzón está declarada físicamente en el Kernel como:

```
struct COLA cpbb[MaxBuz][MaxPrio];
```

donde COLA es la estructura general vista en la Tabla 4.21.

Cola de Procesos Retrasados (CPR)

Debido a las operaciones especiales que se realizan en cuanto al tiempo de retraso, la estructura de la cola de retrasados es totalmente diferente a las otras colas manejadas en el kernel que son de tipo FIFO circular.

La manera en que trabaja la cola de retrasados es la siguiente:

1. Para el primer elemento en la cola de retrasados, éste es insertado tal y como llegó sin sufrir ningún cambio. Ejemplo: llega un proceso que tiene que esperar 8 unidades de tiempo, debido a que es el primero se coloca en la cola de retrasados sin cambiar nada (ver ejemplo 1 de la Figura 4.13).

2. Si llegara un proceso con tiempo de retraso mayor, se colocaría después de los procesos con menor retraso y el tiempo de retraso del nuevo sería la diferencia de su tiempo de retraso original menos la suma de los tiempos de retraso de todos los procesos almacenados en la cola antes que éste. Ejemplo: Primero llega un proceso con tiempo de retraso igual a 12 por lo tanto 12-8=4. Si después llega un proceso con retraso igual a 15 quedaría: 15-(8+4)=3 (ver ejemplo 2 de la Figura 4.13).

3. Si llega un proceso con tiempo de retraso menor a los que están en la cola, el nuevo proceso quedaría igual y los procesos se recorrerían a la derecha. El tiempo de retraso del proceso siguiente al nuevo tendría que ser recalculado y los demás procesos a la derecha quedarían igual. Ejemplo: llega 5, por lo tanto 8 se recalcularía 8-5=3 y los demás quedarían igual (ver ejemplo 3 de la Figura 4.13).

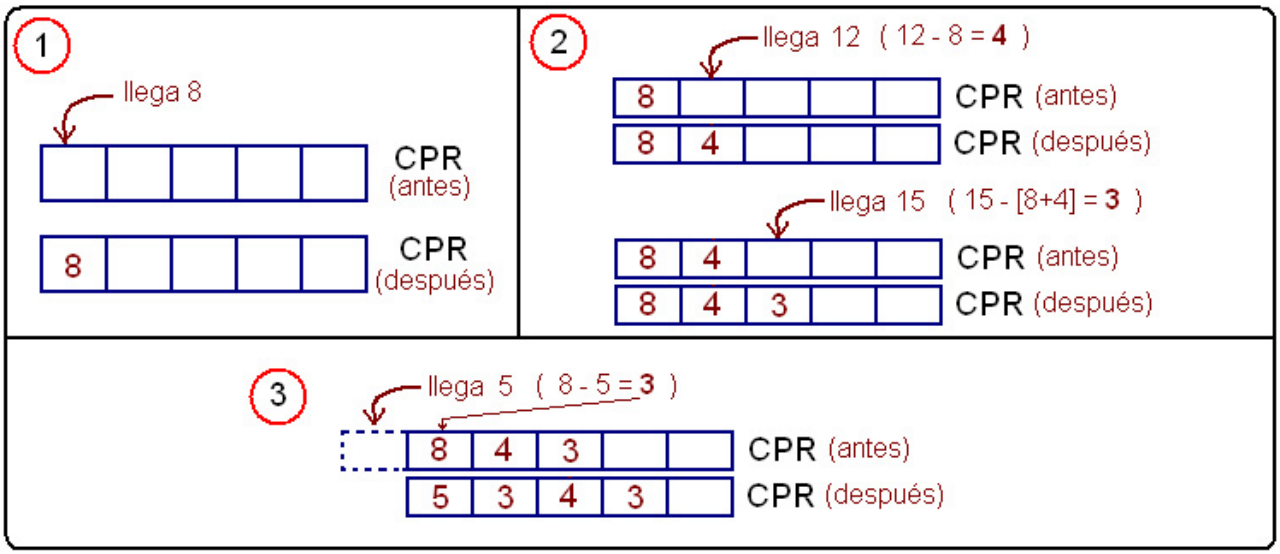

Figura 4.13: Ejemplos del Comportamiento de la Cola de Procesos Retrasados.

Procedimientos Utilizados en el Manejo de Colas

Siguiente :

NOMBRE: `unsigned Siguiente (unsigned X).`

FUNCIÓN: Sirve para encontrar el siguiente elemento en la cola.

ENTRADAS: El apuntador inicial o final de los elementos en la cola.

SALIDAS: Regresa el índice del siguiente elemento en la cola.

```
1.  unsigned Siguiente (unsigned X)
2.    { X++;
3.      X=X%Max;
4.      return X;
5.    }
```

Tabla 4.23: Primitiva *Siguiente*

BuscaMayor :

NOMBRE: `unsigned int BuscaMayor(struct COLA cola[]).`

FUNCIÓN: Encuentra el proceso con mayor prioridad en la cola indicada.

ENTRADAS: La cola en la cual se desea encontrar al proceso con mayor prioridad.

SALIDAS: Devuelve el identificador del proceso con mayor prioridad encontrado en la cola. En caso de estar vacía, devuelve 0.

```
1. unsigned int BuscaMayor(struct COLA cola[])
2.   { unsigned int X=0,I;
3.     while (X<MaxPrio && cola[X].Tam == 0) X++; // Salta de prioridad si en la actual la cola esta vacía
4.     if (X == MaxPrio) // Si no encontró ningún proceso...
5.       return 0;
6.     else //Si lo encontró, regresa el primero de la cola
7.       { I=cola[X].Inicial;
8.         I=Siguiente(I); // Encuentra el índice del primer proceso en la cola circular
9.         return cola[X].Proceso[I];
10.      }
11.  }
```

Tabla 4.24: Primitiva *Busca Mayor*.

InicializaCola :

NOMBRE: `void InicializaCola (struct COLA cola[])`.

FUNCIÓN: Inicializa cualquier cola tipo FIFO circular.

ENTRADAS: La cola que se necesita inicializar.

SALIDAS: Ninguna.

```
1. void InicializaCola (struct COLA cola[])
2.   { unsigned int i,MP;
3.     MP=MaxPro-1;
4.     for (i=0;i<MaxPrio;i++)
5.       //Se le asigna MP (MaxPro-1) para que la primera vez que se usen,
6.       { cola[i].Inicial=MP;
7.         // el primer elemento quede almacenado en la primer posición de la cola (en 0)
8.         cola[i].Final=MP;
9.         cola[i].Tam=0;
10.      }
11.  }
```

Tabla 4.25: Primitiva *Inicializa Cola*.

Inserta :

NOMBRE: `void Inserta(struct COLA, unsigned int, unsigned int)`.

FUNCIÓN: Inserta un proceso al final de la cola en base a su prioridad.

ENTRADAS: El identificador de un proceso, su prioridad y la cola donde va a ser insertado.

SALIDAS: En caso de error envía el código que le corresponde.

Saca :

NOMBRE: `unsigned int Saca(struct COLA cola[], unsigned int prioridad)`.

FUNCIÓN: Saca el primer proceso de la cola en base a su prioridad.

ENTRADAS: La cola y la prioridad donde se desea sacar el proceso.

```
1. void Inserta(struct COLA cola[], unsigned int prioridad, unsigned int proceso)
2.   { unsigned int I,F;
3.     I = cola[prioridad].Inicial;
4.     F = cola[prioridad].Final;
5.     if (I==F && cola[prioridad].Tam == MaxPro) // Revisa si la cola esta llena
6.       ERROR(2);
7.     else  // si no esta llena...
8.       { F=Siguiente(F); // Avanza a la posición siguiente en la cola
9.         cola[prioridad].Proceso[F] = proceso; // Inserta el proceso en la cola
10.        cola[prioridad].Final = F; //Actualiza el apuntador "Final"
11.        cola[prioridad].Tam++; //Actualiza el conteo de los elementos en la cola
12.        // Actualiza la prioridad en caso de que haya cambiado al replanificarse con EDF
13.        PROCESO[proceso].prioridad=prioridad;
14.      }
15.  }
```

Tabla 4.26: Primitiva *Inserta*.

SALIDAS: El identificador del proceso encontrado en la cola, y si algo falló, regresa el código del error que lo causó.

```
1. unsigned int Saca(struct COLA cola[], unsigned int prioridad)
2.   { unsigned int I,F,valor=0;
3.     I = cola[prioridad].Inicial;
4.     F = cola[prioridad].Final;
5.     if (I==F && cola[prioridad].Tam==0) // Revisa si la cola esta vacía
6.       ERROR(3);
7.     else
8.       { I = Siguiente(I); // Avanza "Inicial" a la posición siguiente en la cola
9.         valor = cola[prioridad].Proceso[I]; // Saca el primer elemento de la cola
10.        cola[prioridad].Inicial = I; // Actualiza el apuntador "Inicial"
11.        cola[prioridad].Tam--; // Actualiza el conteo de los elementos en la cola
12.      }
13.    return valor;
14.  }
```

Tabla 4.27: Primitiva *Saca*

Rota :

NOMBRE: `void Rota(struct COLA cola[], unsigned int prioridad)`.

FUNCIÓN: Rota los elementos de la cola de la misma prioridad, quedando el primer elemento al final de la cola, el segundo elemento como el primero, y así sucesivamente, dejando todos los elementos de la cola recorridos a la izquierda.

ENTRADAS: La cola y la prioridad donde se desean rotar los elementos.

SALIDAS: La cola con los elementos rotados.

```
1. void Rota(struct COLA cola[], unsigned int prioridad)
2.   { Inserta(cola,prioridad,Saca(cola,prioridad)); }
```

Tabla 4.28: Primitiva *Rota*

Elimina :

NOMBRE: `void Elimina(struct COLA cola[],unsigned int nproc)`.

FUNCIÓN: Elimina un proceso de la cola sin importar en qué posición se encuentre dentro de ésta.

ENTRADAS: La cola y el identificador del proceso a eliminar.

SALIDAS: La cola sin el proceso que se eliminó.

```
1. void Elimina(struct COLA cola[],unsigned int nproc)
2.   { unsigned int F,X,prio,sig,ant; //antes, necesaria para saber que proceso estaba antes de otro
3.     prio = PROCESO[nproc].prioridad;
4.     if (cola[prio].Tam==0)
5.       ERROR(4);
6.     else
7.       { X=cola[prio].Inicial;
8.         X=Siguiente(X); // "X" apunta al primer proceso de la cola.
9.         F=cola[prio].Final;
10.        if (cola[prio].Proceso[X]==nproc) // Si el proceso a eliminar de la cola es el primero...
11.          Saca(cola,prio);  //Saca el proceso de manera natural
12.        else
13.          { while (cola[prio].Proceso[X]!=nproc && X!=F) //Busca donde se encuentra el proceso a eliminar
14.              {ant=X; X=Siguiente(X);}
15.           if (X==F) //Si llega hasta el final de la cola, "X=F" ocurren 2 casos:
16.            { if (cola[prio].Proceso[X]==nproc) // 1) que el ultimo de la cola sea el proceso que queremos eliminar
17.                { cola[prio].Final=ant; //sacarlo indicando que el ultimo de la cola es ahora el proceso anterior
18.                  cola[prio].Tam--; // Indicar que hay uno menos en la cola
19.                }
20.              else  // 2) o que el proceso a eliminar no exista en la cola
21.                ERROR(5);
22.            }
23.           else  // Si encontró al proceso antes de llegar al final de la cola...
24.            { sig=X;
25.              while (X!=F)  // Recorre los procesos en la cola después de la posición que ocupaba el proceso eliminado
26.               { sig=Siguiente(sig); // Avanza al siguiente proceso en la cola
27.                 cola[prio].Proceso[X]=cola[prio].Proceso[sig]; //recorre el proceso hacia la izquierda
28.                 if (sig==F) //Si es el siguiente es el ultimo en la cola...
29.                  { cola[prio].Final=X; //Actualiza el apuntador final
30.                    cola[prio].Tam--; //Actualiza el contador de elementos en la cola
31.                  }
32.                 X=sig;
33.               }
34.            }
35.         }
36.     }
37.  }
```

Tabla 4.29: Primitiva *Elimina*

Arregla_cola :

NOMBRE: `void arregla_cola(unsigned int i,unsigned long temp,unsigned long valor,unsigned long temp2)`.

FUNCIÓN: Arregla la cola de retrasados recorriendo los procesos que se encuentran de-

spués del nuevo proceso que se agregó a la cola y le calcula el nuevo tiempo de retraso al proceso que se encontraba en la posición del nuevo proceso que se insertó.

ENTRADAS: La nueva posición (nvapos), el tiempo de retraso (tiemp) y el identificador (id) del primer proceso a recorrer en la cola después de insertar al nuevo. Además el tiempo de retraso del nuevo proceso agregado a la cola (valor).

SALIDAS: La cola de retrasados con los procesos recorridos y el primer proceso inmediato al nuevo con su tiempo de retraso recalculado.

```
1. void arregla_cola(unsigned int nvapos,unsigned long tiemp,unsigned long valor,unsigned long id)
2.   { unsigned int x;
3.     x= cpr[0][0]+1;
4.     while (x > nvapos)
5.       { cpr[0][x] = cpr[0][x-1];
6.         cpr[1][x] = cpr[1][x-1];
7.         x--;
8.       }
9.     // Una vez recorridos los procesos, le calcula el tiempo que le corresponde
10.    // al proceso que se encontraba en la posición del nuevo proceso.
11.    cpr[0][x] = tiemp - valor;
12.    cpr[1][x] = id;
14.  }
```

Tabla 4.30: Primitiva *Arregla Cola*

Inserta_r :

NOMBRE: `void inserta_r(unsigned int num_p,unsigned long tiempo)`.

FUNCIÓN: Se encarga de insertar un proceso en la cola de retrasados en la posición que le corresponda en base a su tiempo de retraso.

ENTRADAS: El tiempo a retrasar y el identificador del proceso a insertar.

SALIDAS: La cola con el proceso insertado en la posición correcta.

Saca_r :

NOMBRE: `unsigned int saca_r()`.

FUNCIÓN: Saca al primer proceso de la cola de retrasados y recorre los elementos restantes hacia la izquierda.

ENTRADAS: Ninguna.

SALIDAS: El identificador del proceso que sacó de la cola de retrasados.

Inicializa_r :

NOMBRE: `void inicializa_r()`.

```
1. void inserta_r(unsigned int num_p,unsigned long tiempo)
2.   { unsigned int X=1; // contador
4.     unsigned long temporal=0,temporal2=0, sumatoria=0;
5.     char modificado=FALSO;
7.     if (cpr[0][0] == 0) // si NO hay procesos en la cola se pasa sin cambios
8.       { cpr[0][1] = tiempo; cpr[1][1] = num_p;  }
12.    else
13.      { while (X<=cpr[0][0] && modificado == 0)
14.         { if (tiempo > cpr[0][X]+sumatoria)
17.            { sumatoria += cpr[0][X]; X++; }
20.         else
21.          { modificado = VERDADERO;
22.            temporal = cpr[0][X];
23.            temporal2 = cpr[1][X];
25.            cpr[0][X] = tiempo - sumatoria;
26.            cpr[1][X] = num_p;
29.            arregla_cola(X+1,temporal,cpr[0][X],temporal2);
30.          }
31.         } // Si es 0 indica que al proceso le toca al final de la cola
32.         if (modificado == FALSO)
33.           { cpr[0][X]=tiempo-sumatoria; cpr[1][X]=num_p; }
36.      }
37.    cpr[0][0]++;
38.  }
```

Tabla 4.31: Primitiva *Inserta a la Cola de Retrasados*

```
1. unsigned int saca_r()
2.   { unsigned int x,n_elem,valor;
3.     valor = cpr[1][1]; //saca el primer valor de la cola
4.     for (x=1;x<=cpr[0][0];x++)
5.       { cpr[0][x] = cpr[0][x+1];
6.         cpr[1][x] = cpr[1][x+1];
7.       }
8.     cpr[0][0]--;
9.     return valor;
10.  }
```

Tabla 4.32: Primitiva *Saca de la Cola de Retrasa*

FUNCIÓN: Inicializa la cola de procesos retrasados a 0, el cual sirve para indicar que no hay elementos en ella.

ENTRADAS: Ninguna.

SALIDAS: La cola de retrasados vacía y lista para su uso.

```
1. void inicializa_r()
2.   {
3.     unsigned int i;
4.     for(i=0;i<MaxPro;i++)
5.       { cpr[0][i]=0;
6.         cpr[1][i]=0;
7.       }
8.   }
```

Tabla 4.33: Primitiva *Inicializa Cola de Retrasa*

4.3.4. Manejo de Planificadores

Los tres mecanismos de planificación que tiene el Kernel son:

1. FIRO ROUND ROBIN que es cíclico.
2. RATE MONOTONIC (RM) que es con prioridades estáticas.
3. EARLIEST DEADLINE FIRST (EDF) que es con prioridades dinámicas.

En el manejador de planificadores se encuentran los mecanismos de planificación que el Kernel soporta actualmente además de un procedimiento que verifica si el proceso es planificable o no (ver Figura 4.14).

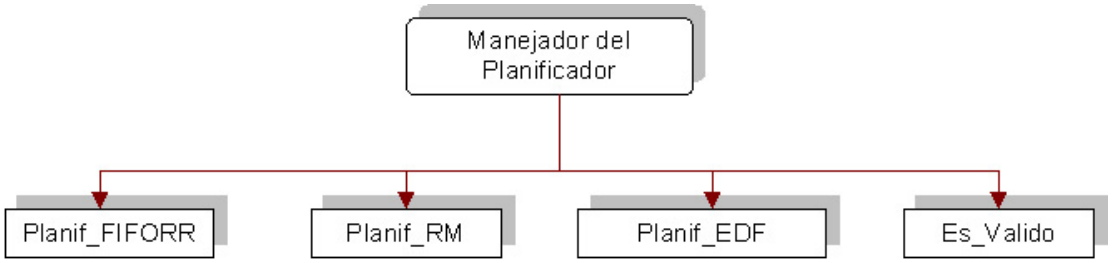

Figura 4.14: Procedimientos del Manejador del Planificador.

La estructura de los procedimientos que se encuentran en el manejador del planificador son los siguientes:

Planif_FIFORR :

NOMBRE: `unsigned int Planif_FIFORR()`.

FUNCIÓN: Es el planificador FIFO Round Robin, y consiste en elegir al proceso con mayor prioridad que se encuentre en la cola de procesos listos.

ENTRADAS: Ninguna, directamente busca el proceso en la cola de procesos listos.

SALIDAS: El identificador del siguiente proceso a ejecutar.

Planif_RM :

NOMBRE: `unsigned int Planif_RM()`.

```
1. unsigned int Planif_FIFORR()
2.   {unsigned int Mayor;
3.    TiempoC=0;
4.    Mayor=BuscaMayor(cpl); // Buscamos la tarea de mayor prioridad
5.    Rota(cpl,PROCESO[Mayor].prioridad); //Se rota la cola de listos
6.    return Mayor;
7.  }
```

Tabla 4.34: Planificador *FIFO ROUND ROBIN*

FUNCIÓN: Es el planificador Rate Monotonic y se encarga de elegir un proceso para su ejecución con base en una serie de reglas descritas en la sección 3.5.1.
ENTRADAS: Ninguna, directamente busca el proceso en la cola de procesos listos.
SALIDAS: El identificador del siguiente proceso a ejecutar.

Planif_EDF :

NOMBRE: `unsigned int Planif_EDF()`.
FUNCIÓN: Es el planificador Earliest DeadLine First y se encarga de elegir un proceso para su ejecución con base en un conjunto de reglas descritas en la sección 3.6.1.
ENTRADAS: Ninguna, directamente busca el proceso en la cola de procesos listos.
SALIDAS: El identificador del siguiente proceso a ejecutar.

Es_Valido :

NOMBRE: `char Es_Valido()`.
FUNCIÓN: Es una función que comprueba si la tarea en ejecución es válida para ser planificada. Esto se hace debido a que tanto la primera tarea como la última no tienen un Tiempo de Cómputo o un período definido y que son necesarios para que los planificadores tomen la decisión correcta.
ENTRADAS: El identificador de proceso que está en ejecución.
SALIDAS: un valor de FALSO o VERDADERO dependiendo de si es o no válido.

4.3.5. Manejo del Reloj

Un sistema de tiempo real tiene restricciones de tiempo bien definidas, por lo que es obligatorio para todo sistema operativo de tiempo real que todos los tiempos de respuesta que ofrezcan sean lo suficientemente rápidos, ya que las acciones de control (para este tipo

```
1. unsigned int Planif_RM()
2.   { unsigned int Mayor;
3.     //Lleva el control del tiempo que les faltan para salir a los procesos de la cola de retrasados.
4.     unsigned long TRetrasa;
5.     char finaliza=FALSO; //indica si ocurrió un evento
6.     char VER=VISUALIZAR; // Sirve elegir si se muestra o no el ic y fc de las tareas
7.     // METE RETRASA
8.     if (Es_Valido(idposeedor) && PROCESO[idposeedor].tcc == PROCESO[idposeedor].TComputo)
9.       { PROCESO[idposeedor].tcc = 0;
10.        finaliza=VERDADERO;
11.        TRetrasa = PROCESO[idposeedor].TA-Ticks;
12.        // Si TRetrasa es 0 Significa que termino pero que es su tiempo de Activación
13.        if (TRetrasa != 0)
14.         { Elimina(cpl,idposeedor);
15.           inserta_r(idposeedor,TRetrasa);
16.           PROCESO[idposeedor].estado=RETRASADO;
17.         }
18.      }
19.     Mayor=BuscaMayor(cpl);
20.     //Rota todos los procesos que tienen la misma prioridad que el nuevo proceso a ejecutarse
21.     Rota(cpl,PROCESO[Mayor].prioridad);
22.     // COMPRUEBA SI PERDIO SU PLAZO
23.     if (Es_Valido(Mayor)) // Si la tarea a ejecutar no es ni la primera ni la ultima...
24.       {if (PROCESO[Mayor].TA == Ticks && PROCESO[Mayor].tcc>0)
25.          { //Si es la hora de activarse y su tcc es mayor a 0, indica perdida de plazo
26.            PROCESO[Mayor].tcc=0;
27.          }
28.         // Se calcula el siguiente Tiempo de Activación
29.         if (PROCESO[Mayor].tcc == 0)
30.          { PROCESO[Mayor].TA += PROCESO[Mayor].Periodo;
31.            IDposAnt = Mayor; // guarda el idposeedor anterior.
32.            finaliza=FALSO;
33.          }
34.         PROCESO[Mayor].tcc++; // aumentar el tiempo de computo consumido
35.        }
36.     if (finaliza==VERDADERO)
37.       IDposAnt = Mayor;
38.     return Mayor;
39.  }
```

Tabla 4.35: Planificador *RATE MONOTONIC*

de sistemas) no pueden hacerse esperar. En el manejador del reloj del Kernel se encuentran los procedimientos necesarios para poder modificar el tiempo (ver Figura 4.15) y obtener una resolución de 1000 interrupciones por segundo, en lugar de las 18.2 interrupciones por segundo que proporciona la máquina por defecto[4].

El control del tiempo e interrupciones periódicas dentro del Kernel se llevan a cabo mediante el temporizador 8254. Este temporizador, en cualquier PC con procesador compatible con Intel, inicialmente se encuentra alimentado con una señal de 1,193,180Hz y configurado en modo 2 con lo cual genera 18.2 ticks de reloj por segundo (una interrupción 08 es generada

[4]Para el caso de esta tesis, nos referimos a aquellas máquinas con procesador Intel 80x86.

```
1. void Planif_EDF()
2.   { unsigned int val,i,Cambios;
3.     unsigned long ARREGLO[2][NTAREAS]; //ARREGLO utilizado para ordenar las tareas
4.     char finaliza=FALSO; //indica si ocurrió un evento
5.     if (cola_retrasa[0][1] > 0)// SACA RETRASA
6.       { cola_retrasa[0][1]--;
7.         while (cola_retrasa[0][1]==0 && cola_retrasa[0][0])
8.           { val=saca_r();  // Es while porque puede que varias tareas terminen
9.             Inserta(cpl,PROCESO[val].prioridad,val);  //su retraso al mismo tiempo.
10.            PROCESO[val].estado=LISTO;
11.      }}
12.     if (Es_Valido() && PROCESO[idposeedor].tcc == PROCESO[idposeedor].TComputo)// METE RETRASA
13.       { PROCESO[idposeedor].tcc = 0;
14.         finaliza=VERDADERO;
15.         PROCESO[idposeedor].P += PROCESO[idposeedor].Periodo;
16.         // Copia de las tareas y sus plazos a un arreglo de ayuda para el ordenamiento
17.         for (i=1;i<=NTAREAS;i++) // No incluye la primera ni la ultima tarea
18.           { ARREGLO[0][i-1] = i;  //Copia el Numero de la tarea
19.             ARREGLO[1][i-1] = PROCESO[i].P; //Copia el Plazo
20.           }
21.         Cambios=OrdBurMej(ARREGLO); // Se busca la tarea con plazo mas cercano
22.         TRetrasa = PROCESO[idposeedor].TA-Ticks;
23.         if (TRetrasa != 0) // Si es 0 Significa que termino pero que es su tiempo de Activación
24.           { Elimina(cpl,idposeedor);
25.             inserta_r(idposeedor,TRetrasa);
26.             PROCESO[idposeedor].estado=RETRASADO;
27.           }
28.         if (Cambios==VERDADERO) //si hubo cambios de prioridad al ordenar
29.           { for (i=0;i<NTAREAS;i++) // Cambiar Prioridades
30.               { switch (PROCESO[ARREGLO[0][i]].estado)
31.                   {  case LISTO: Elimina(cpl,ARREGLO[0][i]);
32.                                  Inserta(cpl,i+1,ARREGLO[0][i]);
33.                                  break;
34.                      case BLOQUEADO: //si estaba bloqueado o retrasado no es necesario rotarlos
35.                      case RETRASADO: PROCESO[ARREGLO[0][i]].prioridad = i+1;
36.                                  break;
37.            }}}}
38.     idposeedor=BuscaMayor(cpl);
39.     Rota(cpl,PROCESO[idposeedor].prioridad); //Rota los procesos con la misma prioriodad que idposeedor
40.     // COMPRUEBA SI PERDIO SU PLAZO
41.     if (Es_Valido() && PROCESO[idposeedor].TA == Ticks && PROCESO[idposeedor].tcc>0)
42.       PROCESO[idposeedor].tcc=0; //Si es hora de activarse y tcc es mayor a 0, indica perdida de plazo
43.     if (Es_Valido() && PROCESO[idposeedor].tcc == 0) // Se calcula el siguiente Tiempo de Activación
44.       { PROCESO[idposeedor].TA += PROCESO[idposeedor].Periodo;
45.         IDposAnt = idposeedor; // guarda el idposeedor anterior.
46.         finaliza=FALSO;
47.       }
48.     if (finaliza==VERDADERO) IDposAnt = idposeedor;
49.     if (Es_Valido)  PROCESO[idposeedor].tcc++; // aumentar el tiempo de computo consumido
50.  }
```

Tabla 4.36: Planificador *EARLIEST DEADLINE FIRST*

por cada tick). Esta frecuencia es muy baja para los sistemas de tiempo real, por lo que en el Kernel se re-programó el timer cambiando su valor inicial por otro que pudiera generar interrupciones periódicas a 1000Hz (1,000 por segundo); esta frecuencia es similar a la que se

```
1. char Es_Valido(unsigned int tarea)
2.   { //Las tareas a planificar no debe ser ni la primera ni la ultima
3.     if (tarea!=PrimerTarea && tarea!=UltimaTarea)
4.       return VERDADERO;
5.     else
6.       return FALSO;
7.  }
```

Tabla 4.37: Primitiva que revisa si es válido el proceso a planificar

Figura 4.15: Procedimientos del Manejador del Reloj.

obtiene con el reloj de tiempo real (RTC) pero con la ventaja de que el timer tiene la mayor prioridad en el vector de interrupciones y por lo tanto ninguna otra interrupción generada en la máquina lo puede detener.

Procedimientos del Manejador del Reloj

Cambia_Timer :

NOMBRE: `void Cambia_Timer (int frecuencia)`.

FUNCIÓN: Es una función que permite reprogramar el timer, de tal manera que se puede acelerar o disminuir el número de interrupciones por segundo generadas por el timer.

ENTRADAS: La frecuencia dada en interrupciones por segundo.

SALIDAS: El timer con una frecuencia distinta a la inicial.

Normaliza_Timer :

NOMBRE: `void Normaliza_Timer(void)`.

FUNCIÓN: Esta función se encarga de regresar el timer a la normalidad, esto es, que el timer funcione como inicialmente estaba antes de ejecutarse el kernel (a 18.2 interrupciones por segundo).

```
1. void Cambia_Timer (int frecuencia)
2.   { long contador;
3.     asm pushf;
4.     contador = 1193181/frecuencia; //Calcula el valor del conteo para tim0
5.     outportb (0x43,0x34); //Pide acceso al Tim0
6.     // Programa el tim0 para contar hasta el numero deseado
7.     outportb(0x40,contador%255); // Parte Baja
8.     outportb(0x40,contador/255); // Parte Alta
9.     asm popf;
10.  }
```

Tabla 4.38: Primitiva *Cambia Timer*

ENTRADAS: Ninguna.

SALIDAS: El timer con la frecuencia original.

```
1. void Normaliza_Timer(void)
2.   { // Pide acceso al Tim0
3.     outportb (0x43,0x34);
4.     // Programa el Tim0 para contar hasta contador (16 bits)
5.     outportb(0x40,0);
6.     outportb(0x40,0);
7.   }
```

Tabla 4.39: Primitiva *Normaliza Timer*

4.3.6. Manejo de Interrupciones

Actualmente el manejador de interrupciones sólo controla las solicitudes de interrupción del teclado. Sin embargo es posible agregarle la capacidad de reconocer interrupciones generadas por el puerto paralelo, en el cual podría estar conectado cualquier tipo de interfaz dotada de sensores y actuadores. Dentro del manejador de interrupciones se deben colocar todos los procedimientos que controlen los dispositivos de las plataformas (empotradas o no) donde se implemente el Kernel. Como se ve en la Figura 4.16, los procedimientos que se tienen implementados son dos: 'IntTeclado" y "RevisaTecla".

IntTeclado :

NOMBRE: `void interrupt IntTeclado()`.

FUNCIÓN: Es una rutina de tipo interrupción y es la que reemplazó a la int 0x09 dedicada a controlar el teclado. Esta función es llamada cada vez que se pulsa una tecla y especialmente se encarga de revisar si el usuario desea salir del sistema.

ENTRADAS: Ninguna, al iniciar contiene en el buffer de teclado la tecla que se pulsó.

SALIDAS: Ninguna.

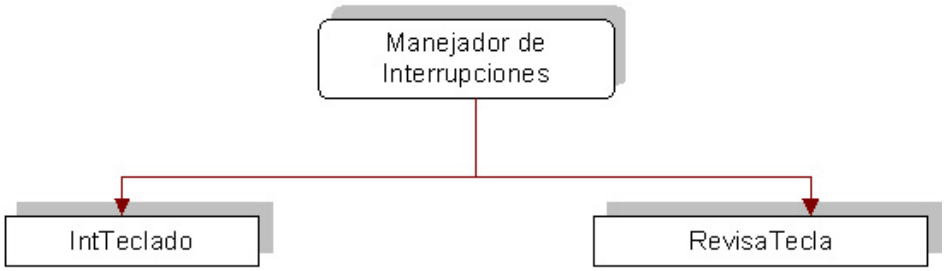

Figura 4.16: Procedimientos del Manejador de Interrupciones.

```
1. void interrupt IntTeclado()
2.   { old_IntTeclado();
3.     RevisaTecla(); // Revisa que tecla se pulsó.
4.     if (SALIR==VERDADERO) //Si se pulso la tecla FIN o ESC
5.       { _SS=oldss;
6.         _SP=oldsp;
7.       }
8.   }
```

Tabla 4.40: Primitiva *Interrupción del Teclado*

RevisaTecla :

NOMBRE: `void RevisaTecla()`.

FUNCIÓN: Revisa en el buffer del teclado si la tecla que se pulsó es FIN o ESCAPE. Si no son, continúa con la ejecución normalmente. En caso de que sí las hayan pulsado, regresa las interrupciones del teclado y el reloj a su estado original y termina la ejecución del kernel.

ENTRADAS: Ninguna, al iniciar contiene en el buffer de teclado la tecla que se pulsó.

SALIDAS: Si fue la tecla ESC o FIN, termina con la ejecución del kernel.

```
1. void RevisaTecla()
2.   { int val;
3.     val=inportb(0x60);
4.     if(val==0x4F OR val==1) //Tecla FIN  o ESC
5.       { setvect(0x08,old_IntReloj); //regresa la interrupcion del reloj original
6.         setvect(0x09,old_IntTeclado); //regresa la interrupcion del teclado original
7.         SALIR=VERDADERO; //SALIR
8.   }
```

Tabla 4.41: Primitiva *Revisa Tecla*

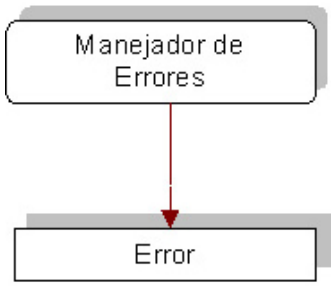

Figura 4.17: Procedimiento del Manejador de Errores.

4.3.7. Manejo de Errores

Cuando un proceso llama a una primitiva, antes de ejecutarla, el Kernel lleva a cabo una serie de validaciones para detectar posibles parámetros erróneos o eventos que no se esperaban. El Kernel tiene implementado un manejador de errores de forma que si se llega a detectar uno, el sistema manda un mensaje de error indicando qué fue lo que ocurrió y además, termina la ejecución del mismo debido a que es un sistema de tiempo real y los procesos que se manejan son muy delicados por lo que no debe continuar si ocurre un error. Este manejador tiene un solo procedimiento encargado de manipular los errores (ver Figura 4.17) y tiene la siguiente estructura:

Error :

NOMBRE: `void ERROR(unsigned int E);`.

FUNCIÓN: Se encarga de mostrar al usuario en caso de que ocurriera un error, la posible causa que lo generó. Debido a que un Kernel es un sistema delicado, cualquier error encontrado lleva a la finalización del sistema y por lo tanto, esta función antes de finalizar intercambia las rutinas del teclado y el reloj por las que tenía el sistema antes de iniciar el Kernel.

ENTRADAS: Un identificador del error (E), el cual contiene el código del error que se generó.

SALIDAS: Un mensaje con el error detectado y la terminación completa del Kernel.

Códigos de Error

ERROR 1: Ocurre cuando el procedimiento "BuscaMayor" intenta conocer el proceso de mayor prioridad pero resulta que la cola de procesos listos está vacía.

ERROR 2: Este error se genera cuando se intenta insertar un proceso en cualquiera de las colas circulares tipo FIFO (la de listos, buzones o semáforos) y no hay espacio disponible en la cola.

ERROR 3: Este error sale cuando se intenta sacar un proceso de cualquier cola tipo FIFO circular y no hay elementos en ella (se encuentra vacía).

ERROR 4: Error que ocurre cuando se intenta eliminar un proceso de una cola tipo FIFO circular y no hay elementos en ella.

ERROR 5: Este error se genera cuando el proceso que se desea eliminar no está en la cola indicada.

ERROR 6: Se genera cuando la primitiva "Retrasa" tiene un tiempo de retraso inválido.

ERROR 7: Ocurre cuando la primitiva "Activa" contiene un identificador de proceso mayor a los permitidos en el Kernel.

ERROR 8: Este error se genera al intentar activar un proceso con mayor prioridad que la del proceso que lo creó.

ERROR 9: Ocurre cuando se intenta utilizar un identificador de semáforo mayor al número de semáforos permitidos.

ERROR 10: Este error sale cuando se intenta dar un valor inicial diferente de 0 o 1 al semáforo binario.

ERROR 11: Error que se genera cuando se intenta crear un semáforo ya creado.

ERROR 12: Error que sale cuando se intenta utilizar un semáforo que todavía no ha sido creado.

ERROR 13: Este error ocurre cuando la cuenta del semáforo es inválida.

ERROR 14: Ocurre cuando se intenta utilizar un número de buzón mayor a los permitidos.

ERROR 15: Sale este error cuando se quiere crear un buzón que ya ha sido creado.

ERROR 16: Error que se genera cuando se intenta utilizar un buzón que todavía no ha sido creado.

ERROR 17: Este error sale cuando en el archivo de configuración se ha escrito un planificador no válido.

4.4. Configuración e Inicialización del Kernel

4.4.1. Configuración del Kernel

El Kernel cuenta con un archivo de configuración llamado *CONFIG.H*, en el cual el usuario puede modificar el contenido de las variables para que el Kernel se adapte a sus necesidades. Entre las cosas que se pueden modificar se encuentran:

1. El mecanismo de planificación que el Kernel utilizará (PLANIFICADOR). Puede ser Rate Monotonic, Earliest Deadline First y FIFO Round Robin.
2. El número de tareas hechas por el usuario (NTAREAS). La cantidad máxima de tareas o procesos depende de esta variable ya que automáticamente se suman la primera y última tarea al valor total de procesos en el Kernel (MAXPRO=NTAREAS+2). Cada vez que se cree una nueva tarea o proceso se debe aumentar aquí su valor.
3. El número máximo de prioridades manejadas por el Kernel (MAXPRIO).
4. El número total de semáforos (MAXSEM).
5. El total de buzones a utilizar en el Kernel (MAXBUZ).
6. El número total de mensajes a almacenar en el buzón (MAXMJE).
7. La longitud máxima para el mensaje (MAXLONGMJE).
8. El tiempo de cómputo permitido en el mecanismo de planificación FIFO Round Robin (QUANTUM).

Por defecto el Kernel se encuentra configurado de la siguiente manera:

- Utiliza el planificador Rate Monotonic.
- Permite crear 10 semáforos.
- Existen 5 tareas hechas por el usuario.
- Maneja 15 niveles de prioridad.
- Soporta 15 Buzones.
- Almacena 10 mensajes por buzón con una longitud de 15bytes cada uno.
- El Quantum para la planificación FIFO Round Robin es de 5.

4.4.2. Inicialización del Kernel

Para poder ejecutar el Kernel es necesario compilar el archivo KERNEL.C, este archivo es el principal y es el que inicializa todo el sistema. Aquí se encuentran declaradas todas las librerías que contienen los manejadores y las primitivas del kernel y además, es aquí donde se deben indicar el nombre de los archivos que contienen las tareas o proceso a ejecutar. El proceso de inicialización del sistema es el siguiente:

1. Se cambia la resolución del "timer" para forzar que el temporizador de la máquina genere interrupciones periódicas cada milisegundo.
2. Se ejecuta la primitiva *inicializa()* la cual se encarga de lo siguiente:
   - Inicializar cada una de las tareas poniendo su estado a "NUEVO".
   - Inicializar las colas de procesos utilizadas en el kernel (cola de procesos listos, retrasados, de semáforos y de procesos bloqueados por buzón), además de crear el semáforo 0, el cual está reservado para la región crítica.
3. Crea los procesos primero y último.
4. Se crean los procesos hechos por el usuario.
5. Activa los procesos primero y último.

6. Por último manda a ejecutar la primera tarea.

Una vez ejecutada la primera tarea, quien se encargó de activar todos los procesos que se van a correr en el kernel, el control y la decisión de lo que se va a ejecutar en el procesador la toma el planificador y a partir de ahí, es este quien decide el comportamiento del kernel y la única manera de terminar con la ejecución del kernel es pulsando la tecla ESC o FIN.

4.5. Datos Técnicos

4.5.1. Tamaño

Debido a que el Kernel fue diseñado para su implementación en un sistema empotrado, el Kernel contiene sólo las funciones que son necesarias para el control de los sistemas de tiempo real en este tipo de dispositivos. Por lo tanto, el tamaño del Kernel es muy pequeño (20.5Kb). Esta característica hace posible que se pueda implementar en sistemas empotrados con altas restricciones de memoria. En la Tabla 4.42 se muestra a detalle el tamaño de cada uno de los archivos que conforman el Kernel.

Nombre del archivo	*Tamaño*
kernel	1.05Kb
config	231 bytes
constant	1.55Kb
libreria	1.97Kb
manreg	1.49Kb
mancpu	491 bytes
manplan	1.99Kb
mancolas	4.24Kb
manreloj	376 bytes
manint	304 bytes
manerror	1.85Kb
pribuz	2.51Kb
pripro	621 bytes
prisem	1.67Kb
pritim	255 bytes
TOTAL	20.5Kb

Tabla 4.42: Tamaño del Kernel.

4.5.2. Tiempos de ejecución del Kernel

En cualquier aplicación de tiempo real es importante conocer el tiempo de ejecución de las primitivas diseñadas en el Kernel, debido a la predecibilidad con que debe contar el sistema.

En la Tabla 4.43 se indica que el cambio de contexto tarda en promedio 32.44 microsegundos. Debido a que la granularidad del Kernel es de 1 milisegundo (1,000 interrupciones por segundo), el tiempo que le resta al proceso para ejecutarse (después de la ejecución de la rutina de cambio de contexto) sería de aproximadamente 967.55 microsegundos.

Los tiempos presentados en la Tabla 4.43 fueron obtenidos en una computadora Laptop Pentium 4 a 1.4GHz, con 240MB de RAM y 16MB de RAM en video. Con la finalidad de obtener las mediciones más realistas posibles, el Kernel se ejecutó sólo bajo el ambiente de MS-DOS (fuera del ambiente de Windows).

Primitiva	*Tiempo mínimo(μs)*	*Tiempo máximo(μs)*	*Tiempo promedio(μs)*
Inicialización del sistema	11.43	13.23	12.34
Cambio de contexto	30.37	34.12	32.44
Activa	11.73	12.57	11.93
Elimina	11.73	14.24	12.08
Retrasa	11.23	13.23	12.23
Crea semáforo	11.73	12.57	11.81
Señal	11.73	13.40	12.17
Espera	11.73	12.57	12.17
Crea buzón	11.73	12.57	11.83
Envía Mensaje	12.57	14.24	12.80
Recibe Mensaje	13.40	15.92	14.23
Inserta a la cola	11.73	12.57	12.32
Saca de la cola	11.73	12.57	12.30
Busca el mayor	23.46	24.30	23.59
Earliest Deadline First	11.73	15.08	12.28
Rate Monotonic	11.73	13.40	12.25

Tabla 4.43: Tiempos de ejecución de las primitivas del Kernel.

Capítulo 5

Ejemplo de Aplicación: Control Proporcional Derivativo de Motores de Corriente Directa

5.1. Introducción

Este capítulo está dedicado a la descripción de una aplicación desarrollada en el *Centro de Servicios Experimentales* del departamento de Control Automático. El objetivo es experimentar con el kernel elaborado en esta tesis, y conocer el comportamiento de éste ante aplicaciones industriales de tiempo real. Debido a que la aplicación maneja términos específicos de control automático, este capítulo contiene una breve descripción sobre los sistemas de control así como algunos conceptos básicos, los tipos de retroalimentación más utilizados en este tipo de sistemas y el control proporcional derivativo. Además, en este capítulo se explican a detalle las características de los amplificadores y motores de corriente directa con escobillas, la arquitectura de la aplicación que se va a hacer y los resultados obtenidos.

5.2. Sistemas de Control

5.2.1. Introducción al Control

Los sistemas de control automático, son sistemas dinámicos y el conocer la teoría de control, proporcionará una base para entender su comportamiento. Estos sistemas emplean frecuentemente componentes de diferentes tipos, como son: componentes mecánicos, eléctricos, hidráulicos y neumáticos.

En años recientes, los sistemas de control han venido adquiriendo un papel muy importante en el desarrollo y avance de la civilización y tecnologías modernas. Casi todos los aspectos de nuestras actividades cotidianas son afectados por algún tipo de sistemas de control. Por ejemplo, en el campo doméstico, los controles automáticos para calefacción y aire acondicionado que regulan la temperatura y la humedad de los hogares, para lograr una vida cómoda. Para alcanzar una eficiencia máxima en el consumo de energía, muchos sistemas modernos de calefacción y de aire acondicionado están computarizados, en especial en los grandes edificios y las fábricas.

5.2.2. Sistemas de Control Automático

Los sistemas de control son muy comunes en todos los sectores industriales, desde el control de calidad de productos industriales, líneas de ensamble automático, control de máquinas, herramientas, tecnología espacial, armamento, control por computadora, sistemas de transportación, robótica y muchos otros. Incluso problemas relacionados con el control de inventarios, los sistemas de control sociales y económicos, pueden resolverse con enfoques de la teoría del control.

Para comprender los sistemas de control automáticos, es necesario definir los siguientes términos para este tipo de sistemas [28].

Planta. Una planta es un equipo o quizás simplemente un juego de piezas de una máquina funcionando juntas, cuyo objetivo es realizar una operación determinada.

Sistema. Un sistema es una combinación de componentes que actúan conjuntamente y cumplen determinado objetivo. Un sistema no está limitado a los objetos físicos. El

concepto de sistema puede ser aplicado a fenómenos abstractos y dinámicos, como son los sistemas de tiempo real.

Perturbaciones. Una perturbación es una señal que tiende a afectar adversamente el valor de la salida de un sistema. Un ejemplo de perturbaciones en los sistemas de tiempo real es la variación en los tiempos de ejecución de las tareas con respecto a los tiempos estimados en el peor caso.

Sistema de control retroalimentado. Es aquél que tiende a mantener una relación preestablecida entre la salida y la entrada de referencia, comparando ambas y utilizando la diferencia como parámetro de control.

Sistema de control automático. Es un sistema de control retroalimentado en el que la entrada de referencia o la salida deseada son constantes o varían lentamente en el tiempo y donde la tarea fundamental consiste en mantener la salida en el valor deseado a pesar de las perturbaciones presentes.

Cualquiera que sea el tipo de sistema de control considerado, los ingredientes básicos del sistema pueden describirse en términos de:

1. Objetivos del control.
2. Componentes del sistema de control.
3. Resultados.

En la Figura 5.1(a) se ilustra la relación entre estos tres componentes básicos en forma de diagrama de bloques. Estos tres componentes básicos pueden identificarse como entradas (referencias), componentes del sistema (controlador y planta) y resultados (salidas), respectivamente, como se muestra en la Figura 5.1(b).

En general, el objetivo del sistema de control consiste en controlar las salidas y de una manera predeterminada, la cual está dada por la referencia r y el controlador. A las entradas del sistema se le llama también consigna de operación y a las salidas variables controladas.

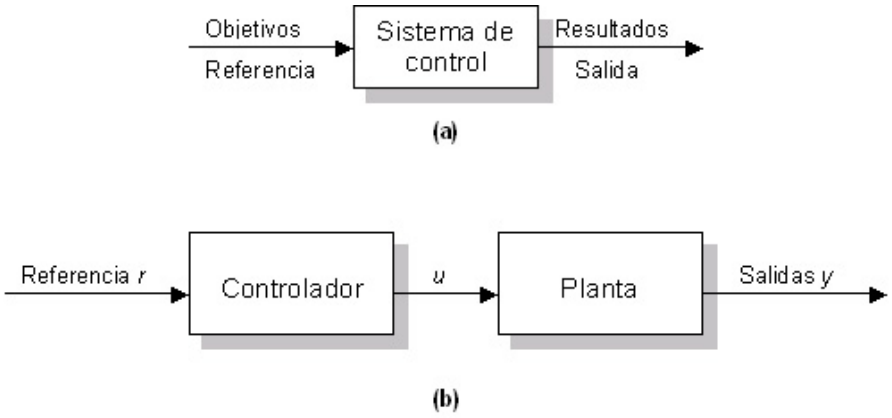

Figura 5.1: Componentes Básicos de un Sistema de Control.

Sistemas de Control en Lazo Abierto. Es un sistema de control simple y económico que no cuenta con una retroalimentación de la salida en el sistema. Un ejemplo de sistemas de lazo abierto son las lavadoras eléctricas. En su diseño típico, el ciclo de lavado queda determinado en su totalidad por la estimación y el criterio del operador humano. Los elementos del sistema de control en lazo abierto casi siempre pueden dividirse en dos partes: el controlador y el proceso controlado. Tal como se puede apreciar en el diagrama de bloques de la Figura 5.2 se aplica una señal de entrada o comando r al controlador, cuya salida actúa como señal de control u; la señal actuante controla el proceso controlado, de tal manera que la variable controlada y se comporte de acuerdo con estándares predeterminados.

Figura 5.2: Elementos de un Sistema de Control en Lazo Abierto.

Sistemas de Control en Lazo Cerrado. Se le llama *sistema en lazo cerrado* a los sistemas con uno o más lazos de retroalimentación. En estos sistemas, para obtener un control más preciso, la señal controlada $y(t)$ debe retroalimentarse y compararse con la entrada de referencia, tras lo cual se envía a través del sistema una señal de control

proporcional a la diferencia entre la entrada y la salida, con el objetivo de corregir el error. En la Figura 5.3 se muestra el diagrama de bloques de un sistema de control de marcha en reposo en lazo cerrado.

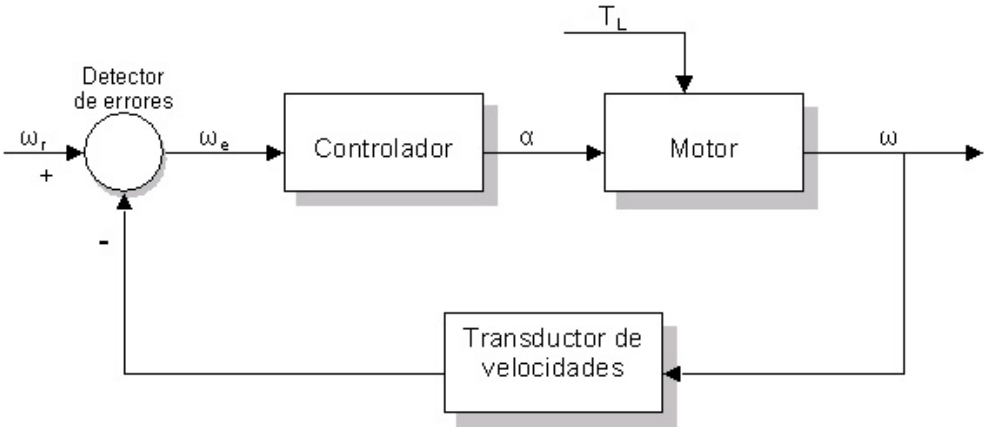

Figura 5.3: Sistema de Control de Marcha en Reposo en Lazo Cerrado.

5.2.3. Tipos de Retroalimentación

La retroalimentación en los sistemas es muy importante, ya que además de permitir la reducción de errores también tiene efectos en las características de desempeño del sistema tales como: estabilidad, ganancia total, sensibilidad y reducción de ruido. En los controles automáticos industriales, son muy comunes los siguientes tipos de retroalimentación: P, I, PI, PD y PID los cuales son descritos a continuación. Es importante comprender las características básicas de cada retroalimentación, para poder elegir la que más se adecúe a determinada aplicación.

Retroalimentación Proporcional (P)

Para un control de acción proporcional, la relación entre la salida del controlador $u(t)$ y la señal de error $e(t)$ es:

$$u(t) = K_p e(t) \tag{5.1}$$

o en magnitudes de transformada de Laplace,

$$\frac{U(s)}{E(s)} = K_p \tag{5.2}$$

donde K_p se conoce como sensibilidad proporcional o ganancia. El control proporcional esencialmente es un amplificador con ganancia ajustable. Valores muy grandes de K_p a menudo pueden llevar a la inestabilidad. Para la mayoría de los sistemas hay un límite superior en la ganancia proporcional para lograr una respuesta bien amortiguada y estable.

Retroalimentación Integral (I)

Es un control con acción integral, el valor de la salida del controlador $u(t)$ varía proporcionalmente a la señal de error actuante $e(t)$. Esto es:

$$\frac{du(t)}{dt} = K_i e(t) \tag{5.3}$$

o

$$u(t) = K_i \int_0^t e(t)dt \tag{5.4}$$

donde K_i es una constante regulable. La función de transferencia del control integral es:

$$\frac{U(s)}{E(s)} = \frac{K_i}{s} \tag{5.5}$$

Si se duplica el valor de $e(t)$, el valor de $u(t)$ varía dos veces más rápido. Para un error actuante igual a cero, el valor $u(t)$ se mantiene estacionario. La acción de control integral recibe a veces el nombre de control de reposición.

Retroalimentación Proporcional e Integral (PI)

La acción de control proporcional e integral queda definida por la siguiente ecuación:

$$u(t) = K_p e(t) + \frac{K_p}{T_i} \int_0^t e(t)dt \tag{5.6}$$

y la función de transferencia del control es:

$$\frac{M(s)}{E(s)} = K_p \left(1 + \frac{1}{T_i s}\right) \tag{5.7}$$

donde K_p representa la sensibilidad proporcional o ganancia y T_i el tiempo integral o tiempo de restablecimiento. Tanto K_p como T_i son regulables. El tiempo integral regula la acción de

control integral, una modificación en K_p afecta tanto la parte integral como la proporcional de la acción de control. A la inversa del tiempo integral T_i se le llama velocidad de restablecimiento. Esta retroalimentación tiene la virtud de poder proporcionar un valor finito de $u(t)$ sin señal de error de entrada $e(t)$. La motivación principal para añadir la acción integral es el reducir o eliminar los errores en estado estable, pero esta ventaja se consigue a costa de una estabilidad reducida.

Retroalimentación Proporcional Derivativo (PD)

La acción de control proporcional y derivativo queda definida por la siguiente ecuación:

$$u(t) = K_p e(t) + K_p T_d \frac{de(t)}{dt} \tag{5.8}$$

y la función de transferencia es:

$$\frac{U(s)}{E(s)} = K_p(1 + T_d s) \tag{5.9}$$

donde K_p es la sensibilidad proporcional y $T(d)$ es el tiempo derivativo, los dos parámetros son regulables. La acción de control derivativa es conocida también como control de velocidad debido a que el valor de control es proporcional a la velocidad de variación de la señal de error actuante. El tiempo derivativo T_d es el intervalo de tiempo en el que la acción de velocidad se adelanta al efecto de acción proporcional. La acción de control derivativa tiene la ventaja de ser anticipada, pero tiene la desventaja de que amplifica las señales de ruido y puede producir efectos de saturación en el actuador. La acción derivativa se utiliza junto a la proporcional para aumentar la amortiguación y para incrementar la estabilidad del sistema. La acción derivativa por sí misma no lleva el error a cero.

Retroalimentación Proporcional Integral Derivativo (PID)

Representa la combinación de las acciones proporcional, integral y derivativa, y tiene las ventajas de cada una de las tres acciones de control individuales. Normalmente se le conoce por sus siglas *PID*. La ecuación de un control con esta acción de control combinada está dada por:

$$u(t) = K_p e(t) + \frac{K_p}{T_i} \int_0^t e(t)dt + K_p T_d \frac{de(t)}{dt} \tag{5.10}$$

y la función de transferencia es:

$$\frac{U(s)}{E(s)} = K_p \left(1 + \frac{1}{T_i s} + T_d s\right) \tag{5.11}$$

donde K_p representa la sensibilidad proporcional, T_d el tiempo derivativo y T_i el tiempo integral. Esta combinación es a menudo utilizada para proveer un grado aceptable de reducción del error simultáneamente a una estabilidad y amortiguación aceptables. Normalmente, los controladores disponibles comercialmente tienen esta forma, y el ingeniero de control únicamente tiene que ajustar o sintonizar las tres constantes de la ecuación para obtener un comportamiento aceptable.

5.2.4. Implementación Digital del Proporcional Derivativo

En la actualidad es una práctica común implementar controladores como el *PD* u otros mediante microprocesadores, para lo cual deben tomarse en cuenta algunas consideraciones al implementarlo en forma digital. Las consideraciones más importantes tienen que ver con el muestreo, la discretización y la selección del periodo de muestreo.

Muestreo

Cuando una computadora digital es utilizada para implementar una ley de control, todo el procesamiento de señales es realizada en instancias discretas en tiempo. La secuencia de operaciones es:

1. Esperar la interrupción del reloj.
2. Leer la entrada analógica.
3. Calcular la entrada de control.
4. Escribir la salida analógica.
5. Actualizar las variables del controlador.
6. Ir al paso 1.

Las acciones de control están basadas sobre los valores de la salida en tiempos discretos. Este proceso es llamado *muestreo*. El caso normal es muestrear la señal periódicamente con periodo T.

Discretización

Para implementar una ley de control continua en el tiempo como el controlador PD sobre una computadora digital, es necesario aproximar la parte derivativa que aparece en la Ecuación 5.8. El término derivativo es simplemente reemplazado por una expresión diferencial de primer orden [29], por lo que la ecuación discretizada para el control proporcional derivativo queda de la siguiente forma:

$$u(t) = K_p e(t) + K_p T_d \left(e(k) - e(k-1)\right) \tag{5.12}$$

Selección del Periodo de Muestreo

La selección del periodo de muestreo T es un problema fundamental en los sistemas muestreados [30]. El periodo de muestreo seleccionado depende de las propiedades de la señal, el método de reconstrucción y el propósito del sistema. En un problema de procesamiento de señales, el propósito es simplemente obtener una señal digital y después recuperarla desde sus muestras, por lo tanto, un criterio razonable para la selección del periodo de muestreo puede ser el tamaño del error entre la señal original y la señal reconstruida.

La fundamentación principal para la selección de la frecuencia de muestreo w_s o el periodo de muestreo T es el teorema de muestreo de Shannon. Este teorema especifica que una frecuencia de muestreo debe ser al menos el doble de la frecuencia más alta de la señal. El ancho de banda de las computadoras afecta la precisión de los valores obtenidos debido a los convertidores A/D y D/A, los coeficientes del controlador y las operaciones aritméticas de la computadora. Otras consideraciones que afectan la precisión incluyen los compensadores, las señales de perturbaciones, las mediciones del ruido, el tiempo de retardo inherente en el sistema, etc.

5.3. Motores y Amplificadores

5.3.1. Motores de Corriente Continua

Todos los motores eléctricos tienen básicamente los mismos componentes. Todos tienen un magneto estacionario denomidado el *estator* y un electroimán denominado la *armadura*.

El estator genera el campo magnético. Cuando una corriente eléctrica se hace pasar por el embobinado de la armadura que se ha colocado en el campo magnético generado por el estator, ésta comienza a rotar debido al *par magnético*. De esta manera, la energía eléctrica se convierte en energía mecánica.

En general, los motores de corriente continua son similares en construcción a los generadores. De hecho, podrían describirse como generadores que funcionan al revés. Todas los motores DC son reversibles. Si se les suministra un voltaje a los terminales del motor eléctrico, su armadura gira y realiza un trabajo mecánico. Por otro lado, si se hace girar el eje del motor y su armadura con la ayuda de algún dispositivo entonces el motor trabajará como un generador eléctrico (dínamo) y producirá corriente eléctrica.

Los cuatro motores utilizados en nuestro ejemplo de aplicación son como el que se muestra en la Figura 5.4. Se trata de motores de corriente directa con escobillas, tienen una masa inercial balanceada (pesa de latón) de 1.3 kg y un codificador óptico incremental (enconder) con una capacidad de resolución de 2,500 pulsos por revolución en hardware, y por software de 5,000 con la posibilidad de incrementarse hasta 10,000.

Las características mecánicas son las siguientes:

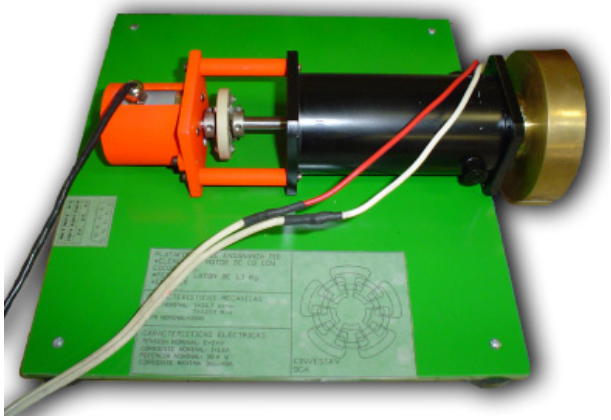

Figura 5.4: Motor de Corriente Continua.

PAR nominal = 36.7 oz-in ó 0.259 N-m

RPM nominal = 2000

Sus características eléctricas son:

Tensión Nominal $E = 29V$

Corriente Nominal $I = 1{,}6A$

Potencia Nominal $38{,}4W$

Corriente Máxima $Imax = 8A$

5.3.2. Amplificadores

Los amplificadores son necesarios para mover y controlar un motor. Estos además de regular su propio desempeño, amplifican la potencia (corriente y voltaje) a 52 volts. Los amplificadores utilizados en esta tesis, son de la compañía Copley Controls Corp. Este modelo contiene una fuente de poder no regulada que opera desde $+24a + 180VDC$ y una corriente de salida desde 10 a $20A$. Estos amplificadores ofrecen muchas ventajas para el control de motores de corriente directa con escobillas. Todos los modelos toman el estándar industrial de $\pm 10V$ para señales de control como entrada, y operan en tres modos diferentes: 1) fuerza de torsión o par, 2) velocidad, y 3) voltaje retroalimentado.

El modo par es el más usado en aplicaciones para motores con tarjetas de control digital que tienen un encoder que permite calcular la velocidad del motor y el control de posición. El modo de velocidad, es usado para ciclos abiertos en el control de velocidad así como en ciclos de control de posición. Por último, el control de velocidad también puede ser hecho mediante el modo de voltaje retroalimentado, sobre todo cuando se requiere un menor costo.

5.4. Arquitectura de la Aplicación

La aplicación consiste en controlar *en posición*, cuatro motores de corriente directa con escobillas a través del Kernel mediante el control proporcional derivativo. La arquitectura de la aplicación se puede ver en la Figura 5.5. El objetivo es lograr el mejor comportamiento de los motores al tratar de imitar una señal de referencia dada. Los dispositivos que se utilizaron para desarrollar la aplicación son los siguientes:

1. *Una computadora*, la cual contiene el kernel que va a controlar la aplicación.

2. *Una tarjeta de adquisición de datos (TAD)*, que cuenta con varias salidas y entradas analógicas y digitales que nos permitirán conocer datos importantes sobre los motores

a controlar [3].

3. *Una caja de interfaz de conexiones*, la cual es un acondicionamiento físico de los cables entre la TAD y los demás dispositivos.

4. *Un dispositivo de aislamiento galvánico*, que permite aislar la computadora de control de la etapa de potencia. Su función es evitar algún daño a la tarjeta de adquisición de datos que por lo general son costosas.

5. *Cuatro amplificadores de corriente y voltaje*, que nos permitirán mover y controlar los motores.

6. *Cuatro motores de corriente directa con escobillas*, y que son los que se quieren controlar.

5.5. Pruebas y Resultados

Para conocer a detalle el comportamiento del kernel ante aplicaciones industriales de tiempo real, se hicieron varias pruebas, todas éstas sin cambiar la arquitectura mostrada en la Figura 5.5. Los resultados de estas pruebas son los siguientes:

5.5.1. Prueba 1: Control PD con Fifo Round Robin

En esta prueba se controlan los cuatro motores de corriente directa utilizando el planificador *Fifo Round Robin*. Por las características de este planificador, las tareas se ejecutan una tras de otra en el orden que han sido preasignadas y no cambia nada durante toda su ejecución. La utilización del procesador es del 100 % y no hay pérdidas de plazo en las tareas. Dentro del Kernel se ejecutan cuatro tareas. Cada tarea (de manera independiente) controla un motor, dibuja una gráfica en pantalla del comportamiento del motor y genera una onda cuadrada como señal de referencia. Para esta prueba, cada tarea genera una señal de referencia diferente, logrando con esto que cada motor gire exactamente una vuelta en un instante de tiempo preciso. Por ejemplo: en esta prueba el motor 1 gira una vez cada segundo, el motor 2 gira cada dos segundos, el motor 3 cada tres segundos y el motor 4 gira cada cuatro segundos.

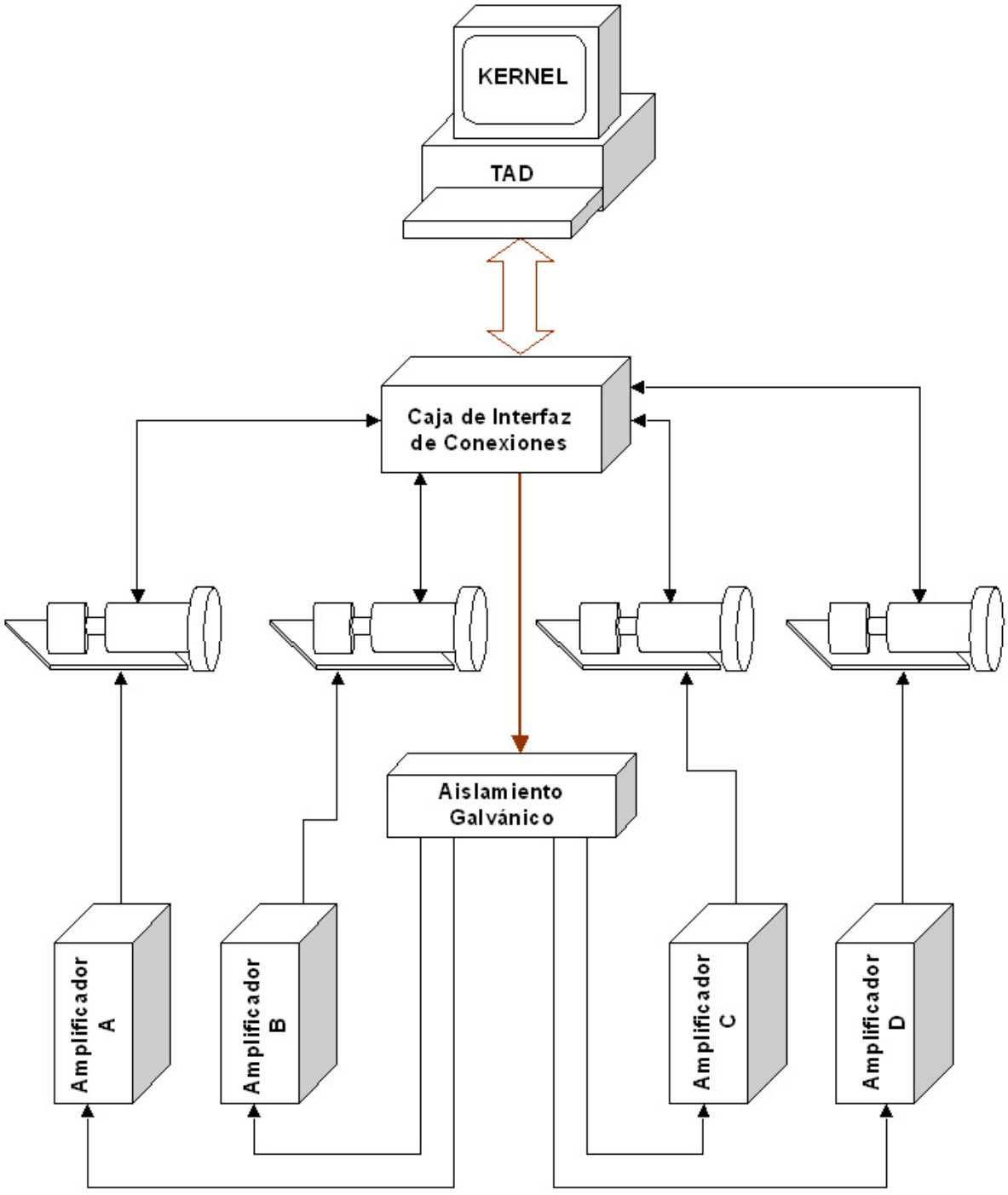

Figura 5.5: Arquitectura de la Aplicación.

Gráficamente, en el Kernel las tareas planificadas con *Fifo Round Robin* cuyo quantum es de 5 milisegundos se comportan como lo muestra la Figura 5.6; y las gráficas que dibujan las tareas en el monitor para esta prueba son como se muestra en la Figura 5.7.

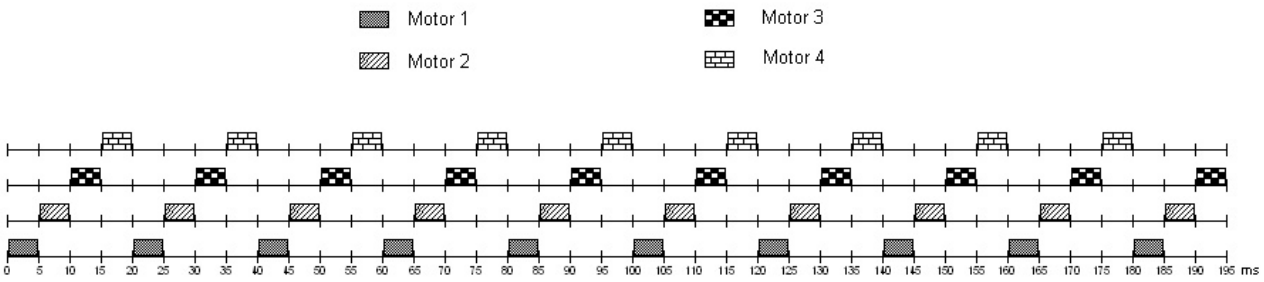

Figura 5.6: Control PD para cuatro motores con *Fifo Round Robin*.

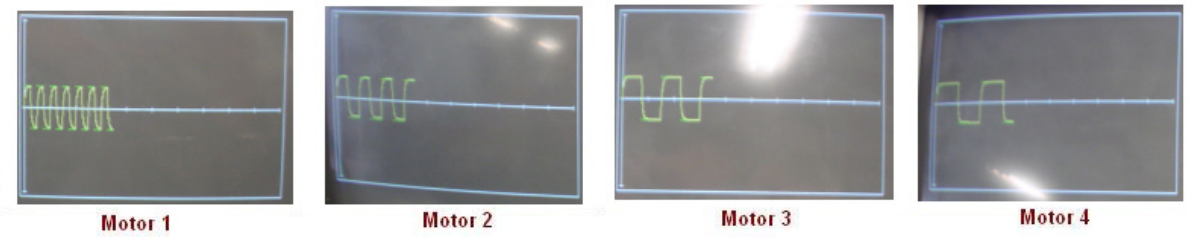

Figura 5.7: Gráficas en Tiempo Real Generadas en la Prueba 1.

5.5.2. Prueba 2: Control PD con Rate Monotonic

Esta prueba consiste de cinco tareas periódicas expulsivas. La tarea 1 (T1) se encarga de generar una señal de referencia intercambiando cada 2 segundos el voltaje de $0{,}25V$ a $-0{,}25V$ y viceversa. Las otras cuatro tareas (T2, T3, T4 y T5) se encargan de controlar los motores y al mismo tiempo graficar en pantalla el estado del motor sobre la señal de referencia[1]. Cada tarea controla su motor mediante el sistema de control proporcional derivativo (PD). Con este control es posible encontrar el voltaje que debe ser enviado al motor para lograr que su comportamiento sea similar al de la señal de referencia generada por la tarea 1 (T1). Como resultado del control, se logra que cada motor gire una vuelta exacta cada 2 segundos

[1]Cada tarea controla y genera la gráfica que le corresponde a su motor asignado.

(intercambiando el sentido de la vuelta tal y como lo indica la señal de referencia).

El mecanismo de planificación utilizado para esta prueba fue Rate Monotonic, y por lo tanto las tareas tienen una prioridad, un tiempo de cómputo C_i y un periodo asignado T_i. La Tabla 5.1 muestra los valores (en milisegundos) asignados a cada tarea, así como la utilización que tienen sobre el procesador.

Tarea	*Descripción*	$T_i$ms	$C_i$ms	*Utilización*
T1	Señal de Referencia	2000	1	0.0005
T2	Motor1	3	1	0.333
T3	Motor2	5	1	0.2
T4	Motor3	7	1	0.142
T5	Motor4	9	1	0.111
				0.7865

Tabla 5.1: Conjunto de Tareas Utilizadas

Como se puede ver en la Tabla 5.1, la utilización total del procesador es del 78 % y por lo tanto no cumple con la condición 3.2 que garantiza que el conjunto de tareas en Rate Monotonic es planificable. La Ecuación 5.13 nos muestra la condición 3.2 aplicada para el caso de cinco tareas. Esta nos indica que la utilización total debe ser de 0.7433 (74 %) para garantizar los plazos de todas las tareas. Esta prueba tuvo el 78 % de utilización por lo que bajo esta condición no es planificable.

$$\left(n\left(2^{\frac{1}{n}}-1\right)\right) = 5\left(2^{\frac{1}{5}}-1\right) = 0{,}7433 \tag{5.13}$$

Sin embargo, la condición 3.2 es suficiente pero no necesaria. En este caso, el factor de utilización de las cuatro primeras tareas (T1,T2,T3 y T4) es igual a 0.6755 y como es menor a 0.7568 tenemos garantizado bajo la condición 3.2 que estas tareas terminan siempre dentro de sus plazos de respuesta. Para la quinta tarea se aplica la condición suficiente y necesaria de Lehoczky (ver teorema 3.5.3). Esta condición nos garantiza, en caso de salir positiva, que el conjunto de tareas no perderá sus plazos. A continuación se muestran los cálculos realizados para comprobar que la tarea 5 cumple con la condición de Lehoczky, y por lo tanto, todas las tareas de esta prueba son planificables sobre Rate Monotonic.

$W_5^0 = C_5 = 1$

$W_5^1 = C_5 + \sum_{\forall_j} \left\lceil \frac{W_5^0}{T_j} \right\rceil C_j = 1 + \left\lceil \frac{1}{2000} \right\rceil 1 + \left\lceil \frac{1}{3} \right\rceil 1 + \left\lceil \frac{1}{5} \right\rceil 1 + \left\lceil \frac{1}{7} \right\rceil 1 = 5$

$W_5^2 = C_5 + \sum_{\forall_j} \left\lceil \frac{W_5^1}{T_j} \right\rceil C_j = 1 + \left\lceil \frac{5}{2000} \right\rceil 1 + \left\lceil \frac{5}{3} \right\rceil 1 + \left\lceil \frac{5}{5} \right\rceil 1 + \left\lceil \frac{5}{7} \right\rceil 1 = 6$

$W_5^3 = C_5 + \sum_{\forall_j} \left\lceil \frac{W_5^2}{T_j} \right\rceil C_j = 1 + \left\lceil \frac{6}{2000} \right\rceil 1 + \left\lceil \frac{6}{3} \right\rceil 1 + \left\lceil \frac{6}{5} \right\rceil 1 + \left\lceil \frac{6}{7} \right\rceil 1 = 7$

$W_5^4 = C_5 + \sum_{\forall_j} \left\lceil \frac{W_5^3}{T_j} \right\rceil C_j = 1 + \left\lceil \frac{7}{2000} \right\rceil 1 + \left\lceil \frac{7}{3} \right\rceil 1 + \left\lceil \frac{7}{5} \right\rceil 1 + \left\lceil \frac{7}{7} \right\rceil 1 = 8$

$W_5^5 = C_5 + \sum_{\forall_j} \left\lceil \frac{W_5^4}{T_j} \right\rceil C_j = 1 + \left\lceil \frac{8}{2000} \right\rceil 1 + \left\lceil \frac{8}{3} \right\rceil 1 + \left\lceil \frac{8}{5} \right\rceil 1 + \left\lceil \frac{8}{7} \right\rceil 1 = 9$

$W_5^6 = C_5 + \sum_{\forall_j} \left\lceil \frac{W_5^5}{T_j} \right\rceil C_j = 1 + \left\lceil \frac{9}{2000} \right\rceil 1 + \left\lceil \frac{9}{3} \right\rceil 1 + \left\lceil \frac{9}{5} \right\rceil 1 + \left\lceil \frac{9}{7} \right\rceil 1 = 9$

Como $W_5^5 = W_5^6 = 9$ y $9 \leq T_5 = 9$, entonces, es planificable.

La Figura 5.8 nos muestra el comportamiento de las tareas en el kernel mediante una gráfica de Gantt y la Figura 5.9 contiene las gráficas en pantalla del comportamiento de los motores durante esta prueba.

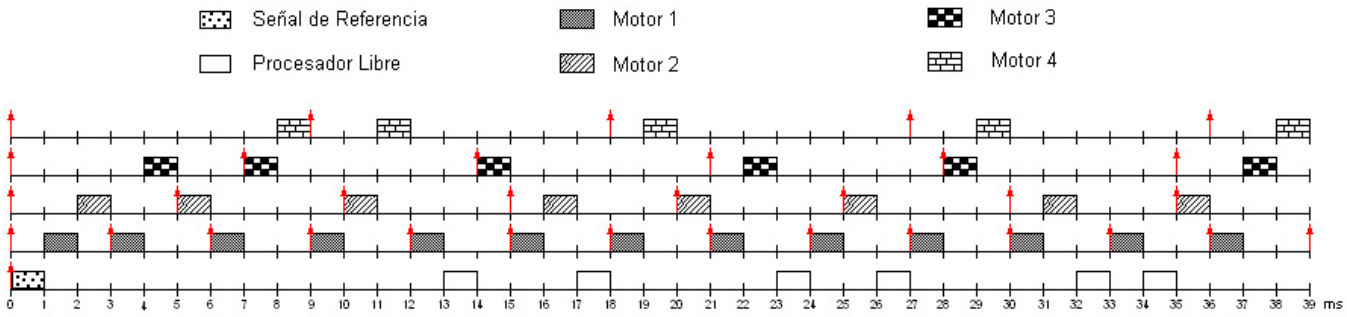

Figura 5.8: Control PD para cuatro motores con Rate Monotonic.

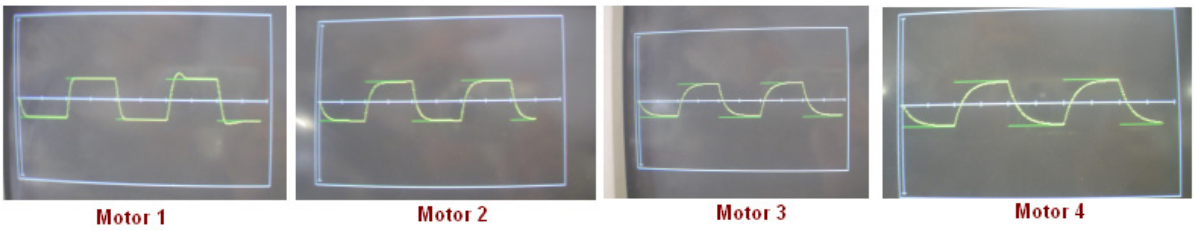

Figura 5.9: Gráficas en Tiempo Real Generadas en la Prueba 2.

5.5.3. Prueba 3: Control PD con Earliest DeadLine First

Esta prueba es similar a la prueba 2 y cuenta con cinco tareas que realizan las siguientes funciones: la tarea 1 (*T1*) genera un señal de referencia, esta señal es una onda cuadrada que cambia cada 2 segundos de $-0{,}25v$ a $0{,}25v$ y viceversa. Las otras cuatro tareas (*T2,T3,T4 y T5*) se encargan de controlar los motores mediante el control proporcional derivativo. Cada tarea controla a un motor en específico y además manda los resultados a un archivo para su graficación posterior. La tabla 5.1 muestra el conjunto de tareas y la utilización que éstas tienen en el procesador.

La diferencia en esta prueba con la anterior, se encuentra en el mecanismo de planificación utilizado; para esta prueba se utilizó EDF (Earliest Deadline First). Este algoritmo tiene la ventaja de garantizarnos que no tendremos pérdidas de plazos aún con un factor de utilización del 100 % (ver Condición 3.11). La utilización total para esta prueba es del 78 % y cumple satisfactoriamente esta condición. La gráfica de Gantt de la Figura 5.10 nos muestra de manera visual cuál es el comportamiento de las tareas en el Kernel.

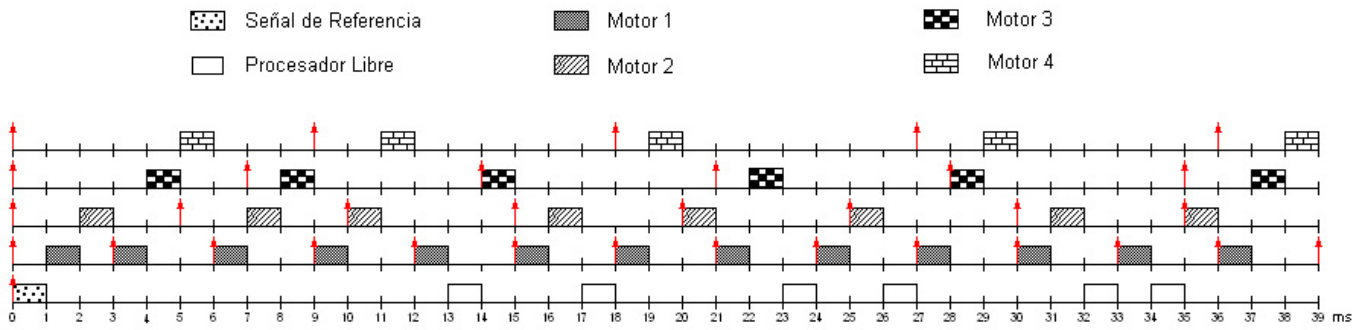

Figura 5.10: Control PD para cuatro motores con Earliest Deadline First.

En la Figura 5.11 se muestra el comportamiento de los motores durante la ejecución del Kernel; los datos graficados fueron obtenidos de un archivo llamado "resul.txt"[2]. Este archivo es creado al iniciar el Kernel, y durante su ejecución, cada tarea se encarga de almacenarle el tiempo, la señal de referencia y la posición del motor en que se encuentra en ese instante. También, en la Figura 5.11 se puede apreciar que el motor 1 es el que más se acerca a la

[2]El motivo de mandar los resultados a un archivo y no graficarlos en tiempo real, se debe a que las librerías gráficas del Kernel no están optimizadas y generan conflictos con el planificador EDF ya que éste pide más recursos que RM.

señal de referencia y que el motor 4 es el que más tiempo tarda en igualarla; esto se debe a la prioridad que se les asignó para esta prueba. La tarea que controla al motor 1 (T2) por tener la mayor prioridad y el período de activación más corto (cada 3 milisegundos) es atendida por el planificador (EDF) con mayor frecuencia y por lo tanto, es la tarea que más tiempo tiene para igualar la posición del motor a la señal de referencia. La tarea que controla al motor 4 (T5) tiene la menor prioridad y el tiempo de activación más largo de todas las tareas que controlan motores (cada 9 milisegundos), esto, en conjunto con el tiempo de respuesta que tiene el motor al aumentarle el voltaje, nos da como resultado que el comportamiento del motor 4 no sea tan preciso como el primer motor, sin embargo, es capaz de dar la vuelta requerida en el instante preciso (cada 2 segundos).

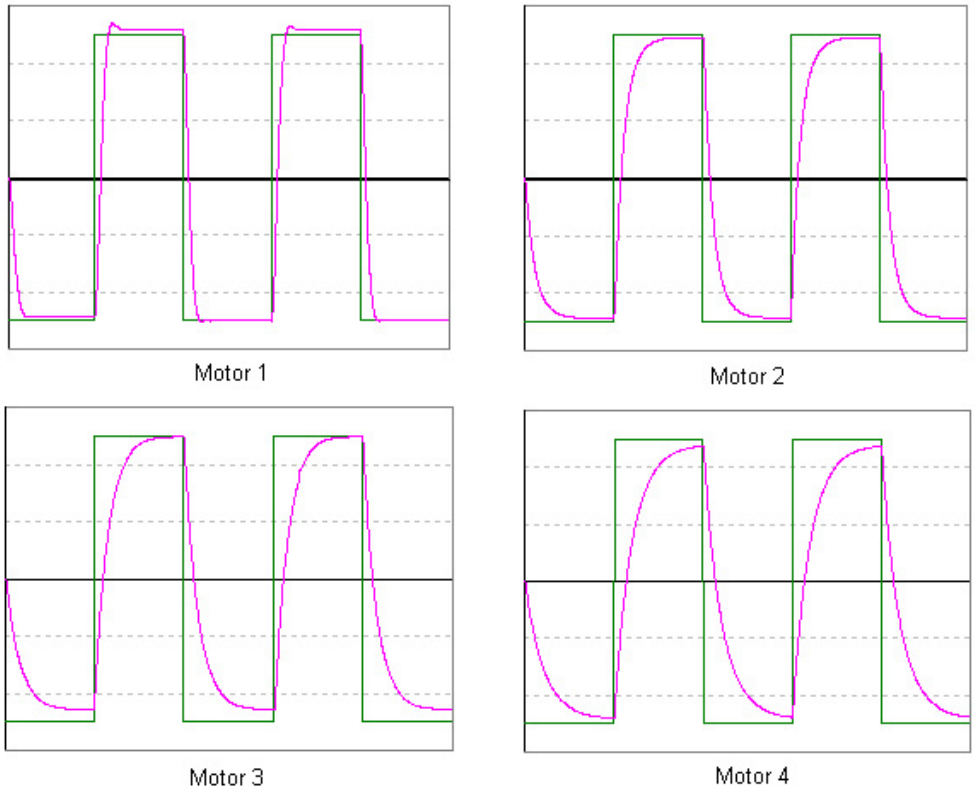

Figura 5.11: Gráfica del Comportamiento de los Motores en EDF.

5.5.4. Prueba 4: Control PD con Pérdidas de Plazo

Esta última prueba está compuesta de 5 tareas, y al igual que en las pruebas anteriores, la tarea 1 se encarga de generar una señal de referencia y las otras cuatro tareas de controlar los cuatro motores en tiempo real. El planificador utilizado es Rate Monotonic.

El objetivo de esta prueba es mostrar el comportamiento del kernel cuando existen pérdidas de plazo en la ejecución de las tareas, para esto fue necesario sobrecargar el uso del procesador de tal manera que su factor de utilización fuera mayor al 100 %. La Tabla 5.2 muestra los valores (en milisegundos) asignados a cada tarea, así como la utilización que tienen sobre el procesador.

Tarea	*Descripción*	$T_i$ms	$C_i$ms	*Utilización*
T1	Señal de Referencia	2000	1	0.0005
T2	Motor1	3	1	0.333
T3	Motor2	5	3	0.6
T4	Motor3	7	1	0.142
T5	Motor4	9	1	0.111
				1.1865

Tabla 5.2: Conjunto de Tareas Utilizadas en la Prueba 4

Con una utilización del 118 % del procesador se obtienen muchas pérdidas de plazo durante la ejecución de las tareas. En la Figura 5.12, la gráfica de Gantt nos muestra los instantes en los que las tareas perdieron sus plazos, y como era de esperarse, las tareas con menor prioridad fueron las que más lo perdieron.

En esta prueba las tareas las podemos clasificar en:

Tareas Críticas. Las tareas T2, T3, T4 y T5 que controlan los motores.

Tareas No-Críticas. La tarea T1 que cambia el valor de la señal de referencia.

Cuando ocurre una pérdida de plazo durante la ejecución de una tarea, el resultado al que se puede llegar a obtener depende completamente de la criticidad de la tarea que lo perdió. Por ejemplo:

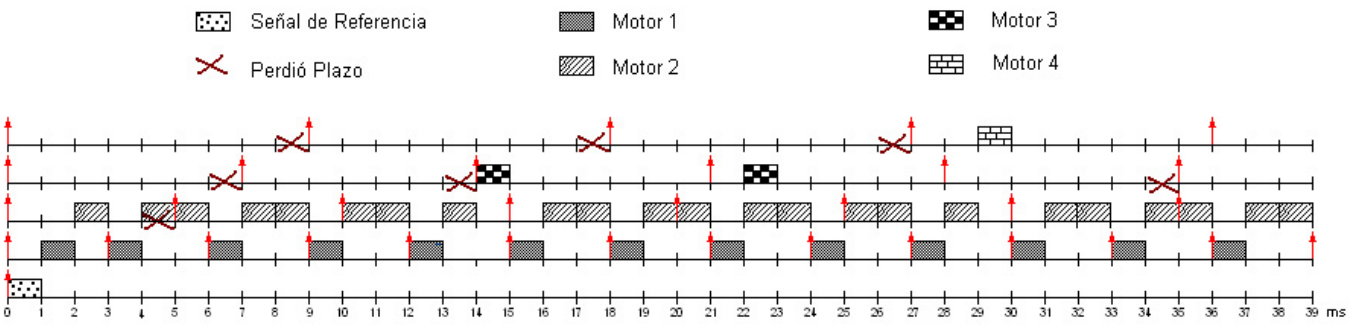

Figura 5.12: Pérdidas de Plazo Ocurridas en la Prueba 4.

Las tareas que controlan los motores, se encargan de enviar un voltaje al motor para aumentar o reducir la velocidad de movimiento, leen del decodificador óptico[3] la posición de giro exacta que tiene el motor en ese instante, y además, calcula el siguiente voltaje a enviar con base en la fórmula de control proporcional derivativo. Si estas tareas llegaran a perder su plazo (tal como sucede en esta prueba), el motor no obtendría el voltaje necesario y podría girar a una velocidad mayor a la que requería. Llegaría el momento en el que motor giraría demasiado rápido y en cuestión de milisegundos la estabilidad de todo el sistema estaría comprometida.

En el caso de que la tarea que perdiera su plazo fuera una tarea no-crítica como la tarea T1, el problema se vería reflejado en el tiempo de giro del motor ya que no cambiaría el sentido del giro cada 2 segundos como se tenía previsto, sin embargo el sistema seguiría funcionando.

Esta prueba es importante y nos muestra claramente lo que puede llegar a suceder en caso de pérdidas de plazo. Es necesario tener muy claro las prioridades que se asignan a las tareas y saber reconocer cuáles son críticas y cuáles no lo son. El analizar por completo el sistema y conocerlo a detalle, nos permitirá asignar de manera correcta, los períodos y tiempos de ejecución de las tareas, además de facilitarnos la decisión en la elección del planificador de tiempo real. La ventaja de estos planificadores, sin importar cual se elija, es que permiten

[3]El decodificador óptico está físicamente integrado al motor y se encarga de medir los pulsos por revolución. Cuando el motor tiene 2,500ppr significa que giró una vuelta exacta

conocer cuál va a ser el comportamiento de las tareas antes de ejecutarlas en el Kernel, permitiéndonos con esto, detectar las posibles fallas en la asignación de tiempos sin tener que comprometer el funcionamiento del sistema de tiempo real.

Capítulo 6

Conclusiones y Trabajo Futuro

6.1. Conclusiones

En la actualidad, los sistemas de tiempo real se han ido integrando a la vida cotidiana cada vez más. Las aplicaciones que los incluyen demandan una alta confiabilidad, y por lo tanto, los resultados que éstos ofrecen deben ser correctos, predecibles y precisos.

Como se mostró en esta tesis, una aplicación de tiempo real está compuesta por un conjunto de tareas concurrentes que cooperan entre sí. En el kernel que se desarrolló, las tareas son activadas a intervalos regulares y deben completar su ejecución antes del término de su plazo de respuesta.

A grandes rasgos, a lo largo de esta tesis se obtuvieron los siguientes resultados:

1. Se creó un kernel de Tiempo Real pequeño con capacidad de ser portado a cualquier otra plataforma portátil o empotrada, es capaz de responder a interrupciones externas, controla y manipula tareas concurrentes y además, comunica y sincroniza procesos mediante primitivas de acceso a buzones y semáforos.

2. Se estudiaron e implementaron los algoritmos de planificación de tareas *FIFO Round Robin*, *Rate Monotonic* y *Earliest Deadline First*, los dos últimos creados especialmente

para sistemas de tiempo real.

3. Se obtuvo un kernel con código fácil de entender y modificar, por lo que permite que se le puedan agregar futuros algoritmos de planificación de tiempo real diseñados en el Cinvestav y a su vez experimentar el funcionamiento de estos en tiempo real dejando atrás las simulaciones.

4. Se experimentó y probó el funcionamiento del kernel en el departamento de control. Para esto se crearon 4 tareas; cada tarea controló un servomotor en tiempo real y su objetivo era manipular el voltaje del motor de tal forma que su comportamiento fuera semejante al de la señal de referencia.

6.2. Trabajo Futuro

El área de los sistemas de tiempo real es muy amplia, y ahora que contamos con un kernel de tiempo real cuyo código conocemos y podemos cambiar con base en nuestras necesidades, son muchas las extensiones o variaciones que se le podrían hacer. A continuación se detallan algunos de estos posibles cambios:

- **Extensión a Tareas Aperiódicas.** El kernel que se desarrolló al igual que todas las investigaciones actuales realizadas dentro de éste, consideran únicamente el caso en donde las tareas son estáticas, los tiempos de cómputo de las tareas son conocidos y se ejecutan por un tiempo máximo pre-establecido. Entre los trabajos a futuro que podrían realizarse se encuentra, extender la capacidad de Kernel para que pueda manejar tareas aperiódicas cuyo tiempo de arribo al sistema sea en instantes aleatorios, y que además los parámetros temporales de las tareas varíen de forma impredecible durante su ejecución sin que esto llegue a afectar el cumplimiento de sus plazos.

- **Extensión a Restricciones de Precedencia.** En el modelo utilizado para diseñar el Kernel, se considera el hecho de que algunas de las tareas preceden en ejecución a otras. Una posible mejora a este modelo sería, la posibilidad de manejar tareas con restricciones de precedencia acíclicas, y con restricciones de precedencia de tipo AND/OR.

- **Extensión a Sistemas Distribuidos y a Sistemas con Múltiples Procesadores.** El Kernel ha sido diseñado para que funcione correctamente en una computadora con un solo procesador. Sin embargo, se le puede mejorar agregándole primitivas que permitan la comunicación entre tareas de kernels que se estén ejecutando en distintas máquinas con la finalidad de poder controlar sistemas de tiempo real distribuidos. Otra mejora importante, es la de cambiar el diseño del kernel de tal forma que sea capaz de ejecutar las tareas en máquinas con 2 o más procesadores.

- **Extensión a Linux.** La plataforma actual del Kernel es MS-DOS debido a que este sistema operativo permite ejecutar el Kernel sin que afecte o deshabilite las interrupciones del hardware de la máquina (tal como lo hace Windows), y además porque con muy pocas modificaciones es posible separarlo de MS-DOS dejando al kernel como sistema operativo independiente. El lenguaje que se utilizó es *ANSI C* por lo que, si en un futuro fuera necesario experimentar el comportamiento del kernel sobre plataformas Linux, es posible realizarlo con tan sólo cambiar aquellas primitivas que utilizan interrupciones como la del cambio de contexto.

- **Extensión a PDAS.** Las agendas electrónicas PALMS, teléfonos celulares y otros PDA's similares son de uso común actualmente y cada vez necesitan tener un mejor sistema operativo. El Kernel desde un principio se diseñó pensando en su adaptación a futuro para este tipo de aparatos (su código es pequeño y funciona con muy poca memoria).

- **Trabajar en Plataformas Empotradas.** Gracias a las características del kernel, es posible introducirlo en cualquier plataforma empotrada que cuente con un procesador con arquitectura de Intel 80x86. Actualmente se tiene pensado adquirir un plataforma de este tipo llamada LBC-9116 la cual tiene un procesador Pentium M de gran velocidad que permite controlar la frecuencia y el voltaje. Esta plataforma junto con el kernel, permitirán probar y experimentar con algoritmos de tiempo real que se tienen y cuya característica principal es maximizar el ahorro de energía y minimizar el número de plazos perdidos.

Bibliografía

[1] Wind River Systems, Inc. Sistemas Operativo de Tiempo Real VxWorks *http://www.windriver.com*

[2] FSM Labs. Real-Time Linux *http://www.fsmlabs.com*

[3] Servo To Go, Inc. Tarjeta SERVOTOGO *http://www.servotogo.com/*

[4] Universidad de Michigan EMERALDS (Extensible Microkernel for Embedded Real-Time, Distributed Systems) *http://kabru.eecs.umich.edu/rtos/emeralds.html*

[5] Escuela Superior Santa Ana de Pisa, Italia. SHARK (Soft and Hard Real-Timer Kernel) *http://shark.sssup.it*

[6] Universidad de Massachusetts en Amherts Sistema Operativo de Tiempo Real Spring. *http://www-ccs.cs.umass.edu/rts/spring.html*

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

[8] Wind River Systems, Inc. The Next Generation of Embedded Development Tools. *Document No. 825-000026-000 property of Wind River Systems, Inc.* Huntsville, Alabama. July 1998.

[9] Dr. Martin Timmerman and Bart Van Beneden. VxWorks/x86 5.3.1 evaluation. *Real-Time Magazine, issue 1999-4,* pages 6-9, 1999.

[10] Marcus Goncalves Is Real-Time Linux for Real?. *Real-Time Magazine, issue 1999-4,* 1999.

[11] Victor Yodaiken and Michael Barabanov. A Real-Time Linux. *In proceedings of Linux Applications Development and Development Conference (USELINUX).* January, 1997.

[12] José Ismael Ripoll Ripoll Tutorial de Real Time Linux *Universidad Politécnica de Valencia.* 2001.

[13] K. M. Zuberi and K. G. Shin. EMERALDS: A microKernel for embedded real-time systems. *In proceedings of IEEE Real-Time Technology and Applications Symposium PP.241-249*, June, 1996.

[14] Khawar M. Zuberi and K. G. Shin. EMERALDS: A Small-Memory Real-Time Microkernel. *In proceedings of IEEE Transactions on Software Engineering, Vol. 27, No. 10*, October, 2001.

[15] Paolo Gai, Luca Abeni, Massimiliano Giorgi and Giorgio Buttazzo. A New Kernel Approach for Modular Real-Time Systems Development. *In proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems,* Delft. NL, June 2001.

[16] Paolo Gai, Giorgio Buttazzo, Luigi Palopoli, Marco Caccamo and Giuseppe Lipari. S.H.A.R.K User Manual. *Scuola Superiore di Studi e Perfezionamento S. Anna, ReTiS Lab,* May, 2003.

[17] J.A. Stankovic and K. Ramamrithan. The Spring Kernel: A New Paradigm for Real-Time Systems *In proceedings of IEEE Software, Vol.8 No.3, pp 62-72,* May, 1991.

[18] John A. Stankovic. The Spring Architecture. *Dept. of Computer and Information Science, University of Massachusetts,* Amherst, Mass. June 1990.

[19] D. Niehaus, E. Nahum, J. Stankovic, K. Ramamritham. Architecture and OS Support for Predictable Real-Time Systems. *Dept. of Computer and Information Science, University of Massachusetts,* Amherst, Mass. March 1992.

[20] Marie Sie-Een Teo. A Preliminary Look at Spring and POSIX.4, *Spring internal document* July, 1995.

[21] Juan A. de la Puente Alfaro Apuntes del Doctorado en Sistemas de Tiempo Real, *Universidad Politécnica de Madrid* Curso 2002-2003.

[22] Silberschatz, Galvin and Gagne. Operating Systems Concepts, *Sixth Edition, Ed. John Wiley and Sons, inc.* 2002.

[23] Giorgio C. Buttazzo. Hard Real-Time Computing Systems, Predictable Scheduling Algorithms and Applications. *First Edition, Ed. Kluwer Academic Publishers.* January, 1997.

[24] C.L. Liu and W. Layland. Scheduling Algorithms for Multiprogramming in Hard Real-Time Environment. *Journal of the Association for Computing Machinery, 2:46-61.* 1973.

[25] Lui Sha John Lehoczky and Ye Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. *IEEE Real-Time Systems Symposium, pages 166-171.* December 1989.

[26] J. Y. T. Leung and J. Whiteheaad. On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Task. *Performance Evaluation, 2(4):237-250.* December 1982.

[27] José Ismael Ripoll Ripoll Planificación en Sistemas de Tiempo Real. *Universidad Politécnica de Valencia, Departamento de Informática de Sistemas y Computadores.* 2000.

[28] Katsuhiko Ogata *Ingeniería de Control Moderna.* Prentice Hall, 1980.

[29] Rolf Isermann *Digital Control Systems. Volume I Fundamentals, Deterministic Control.* Ed. Springer-Verlang, 1989.

[30] Karl J. *A°ström* and *Björn* Wittenmark. *Computer-Controlled Systems: Theory and Design (2nd ed.).* Prentice Hall, Inc. 1990.