Centro de Investigación y de Estudios Avanzados del IPN

Departamento de Ingeniería Eléctrica
Sección Computación

Diseño de un Manejador de Bases de Datos para un Sistema Operativo de Tiempo Real

Presenta:
Luis Arturo Jiménez Mendoza
jimenezl@prodigy.net.mx

Para obtener el grado de:
Maestro en Ciencias en la especialidad de Ingeniería Eléctrica Opción Computación.

Director de Tesis:
Dr. Pedro Mejía Álvarez
pmejia@cs.cinvestav.mx

México, D.F. octubre del 2006.

Resumen

En este trabajo de tesis, presentamos el diseño, la estructura y el desarrollo de un Manejador de Base de Datos de Tiempo Real (MBDTR). El MBDTR tiene una arquitectura que forma parte de un kernel de tiempo real que se ejecuta en MS-DOS, y que trabaja con tareas periódicas, desalojables y en ambientes de un sólo procesador. Las ventajas de este MBDTR son principalmente su tamaño, su modularidad, la facilidad de elegir un tipo de planificador, y la portabilidad de sus algoritmos a otros procesadores.

Por su tamaño el MBDTR y el kernel pueden ser llevados a plataformas empotradas, y que a pesar de estar desarrollados sobre MS-DOS, tienen poca dependencia ya que no usan llamadas al sistema ni hacen uso del BIOS ni de los dispositivos del sistema.
Las características que principalmente soporta este Manejador de Base de Datos en Tiempo Real son: a) Garantiza los requerimientos de tiempo real de las tareas, b) Garantiza los requerimientos de consistencia de los datos, c) Las tareas pueden realizar accesos de lecturas simultaneas y escrituras exclusivas a la Base de Datos, d) Mantenimiento de la integridad de los datos en la Base de Datos, e) Utiliza control de Concurrencia, f) Utiliza técnicas de bloqueo estándar, y g) Soporta la configuración de diferentes algoritmos de planificación.

Las pruebas realizadas sobre el MBDTR fueron totalmente experimentales, considerando la política de planificación FIFO ROUND-ROBIN, y diferentes aplicaciones que incluyen en sus tareas las primitivas del monitor lectores-escritores para el acceso a la base de datos.

ÍNDICE GENERAL

Capítulo 1

Introducción

En los últimos años, el desarrollo de los Sistemas de Tiempo Real se ha incrementado drásticamente, así como también su complejidad, y el monto de datos que manejan. Sin embargo en los sistemas de tiempo real tradicionales el manejo de los datos se hace usando técnicas comunes y estructuras de datos internas. El incremento de los datos causa que surjan nuevos problemas cuando se requiere realizar mantenimiento y desarrollo de aplicaciones. *Una posible solución es integrar a los Sistemas de Base de Datos en Tiempo Real con los Sistemas de Tiempo Real*. Los Sistemas de Base de Datos en Tiempo Real pueden proporcionar al sistema de tiempo real una vista uniforme de los datos, así como el acceso a los mismos. Además de esto, los Sistemas de Base de Datos en tiempo real pueden asegurar la consistencia lógica y temporal de los datos. Otra ventaja importante de estos Sistemas de Base de Datos en tiempo real es que estos nos permiten realizar consultas en tiempo de ejecución, esto es muy útil en sistemas de administración y monitoreo de sistemas de tiempo real.

Integrar estos dos tipos de sistemas tiene sus problemas. Aumenta la sobrecarga por la recuperación de elementos de datos. Esto es principalmente por el sistema de indexamiento usado por los Sistemas de Base de Datos en Tiempo Real (sistemas de indexamiento con árboles y tablas de hash). Además, los datos compartidos en un sistema con accesos concurrentes necesitan protegerse con algún mecanismo de bloqueo. El bloqueo es un factor importante para los datos que se generan en tiempo de ejecución ya que estos son usados frecuentemente por múltiples tareas y cuando se necesita que estén bloqueados

deben de estarlo por un tiempo tan corto como sea posible. Ejemplos de sistemas que generan datos en tiempo de ejecución y que utilizan un sistema de base de datos en tiempo real son:

- Automóviles, trenes urbanos, aviones, ferrocarriles y barcos.
- Control de tráfico de autopistas, control del espacio aéreo, control de rutas de ferrocarriles y líneas de transporte.
- Control de procesos en plantas de energía y plantas químicas.
- Sistemas médicos para terapia radioactiva y monitoreo de pacientes
- Sistemas de misiles, rastreo, comando y control militar.
- Sistemas de manufactura con robots.
- Sistemas de comunicación de radio, teléfono y satélite.

Considerando que las bases de datos en tiempo real tienen una amplia aplicación en los diferentes sistemas como los mencionados anteriormente en este capítulo, en una de sus secciones se realiza un estudio comparativo considerando las diferentes características de las base de datos en tiempo real comerciales, así como también se da un descripción resumida de las base de datos en tiempo real consideradas como puras.

1.1. Motivación

En esta tesis, la motivación de desarrollar un *Manejador de Base de Datos en Tiempo Real* como parte de un Kernel de tiempo real, desde su diseño hasta su construcción, se debe a que se desea obtener el completo conocimiento del Kernel, de las operaciones básicas del álgebra relacional y de los mecanismos para el control de concurrencia denominados monitores, así como también la utilización de estos mecanismos en la solución al problema del monitor de lectores-escritores. Siendo este último el mecanismo que controla el acceso compartido a una base de datos que se localiza en la memoria principal de un sistema computo de un sólo procesador. *El objetivo final es obtener un Manejador de Base de Datos en Tiempo Real que trabaje en forma conjunta con un Kernel de tiempo real* y con capacidad de poder ser instalado en sistemas empotrados que

contienen recursos de memoria limitada, en plataformas que no tienen más de 64 KB de RAM.

La mayoría de los Manejadores de Base de Datos en Tiempo Real han sido diseñados para trabajar en diferentes sistemas operativos de tiempo real con determinadas plataformas donde sus primitivas se encuentran ligadas al sistema operativo y a la arquitectura del hardware. Si se quisiera desarrollar aplicaciones que utilizan un MBDTR, entre los posibles problemas que se presentarían se encuentran:

1) La necesidad de conseguir los sistemas operativos de tiempo real en los que trabajan estos manejadores de base de datos son demasiados caros y pocos comunes.
2) La necesidad de adaptar los sistemas operativos tradicionales para trabajar con tareas de tiempo real y que utilicen un Manejador de Base de Datos es demasiado complejo, costoso, y por consiguiente el tiempo para su adaptación sería mucho mas de lo planeado.

3) La mayoría de los sistemas operativos de tiempo real que cuentan con un Manejador de Base de Datos, y en especial los comerciales, no distribuyen el código fuente por lo que es casi imposible adaptarlos o portarlos a otras plataformas para las que fueron diseñados.

4) La necesidad de conseguir la plataforma de hardware para la cual el Kernel y el Manejador de Base de Datos en la que se encuentran diseñados lo cual podría ser muy caro y en el peor de los casos poco común.

Con el Manejador de Base de Datos en Tiempo Real como parte de un Kernel de tiempo real que se diseñó, estos problemas se solucionan ya que se tiene toda la documentación, el conocimiento sobre el Kernel y el Manejador de Base de Datos, y por lo tanto su modificación o migración a otro tipo de plataformas no sería tan complejo ni requeriría mucho tiempo para ello. Sería solo necesario conocer a detalle la plataforma a la que se quiere migrar sin tener que preocuparse de la arquitectura del Kernel ni de la arquitectura del Manejador de Base de Datos.

1.2. Trabajo Propuesto

Se propone el diseño de un Manejador de Base de Datos en Tiempo Real como parte de un Kernel de Tiempo Real capaz de manipular y controlar procesos concurrentes que cuenten con soporte de transacciones (accesos controlados de lectura y escritura) sobre una base de datos como recurso compartido y que utilizan los distintos mecanismos de planificación (existentes o experimentales).
Se requiere que el Manejador de Base de Datos en Tiempo Real no sea muy grande y que sea parte de un Kernel de Tiempo Real, por lo que se propone un conjunto de primitivas para el control de concurrencia, así como las primitivas para accesar la base de datos, con el objetivo de poder portarlo a distintas plataformas sin tener que realizarle muchos cambios. Actualmente, la mayoría de los Manejadores de Base de Datos en Tiempo Real que existen (tanto comerciales como experimentales), han sido diseñados basándose principalmente en modelo relacional y para plataformas de hardware especificas. El MBDTR que se propone diseñar estará construido como parte de un Kernel de Tiempo Real el cual soportará concurrentemente operaciones del algebra relacional sobre una base de datos y que se ejecuta en una máquina con procesador INTEL 80x86 o compatible, sin embargo debido a su tamaño no cuenta con los manejadores propios para controlar el disco duro, CD-ROM, impresora, etc., este sistema se podrá fácilmente portar a cualquier otra plataforma con tan solo modificar los archivos de configuración del Kernel y del MBDTR en base al código o instrucciones que maneje la nueva plataforma o sistema empotrado.

Aún considerando el Kernel que no cuenta con los manejadores de disco duro, y como parte de las pruebas experimentales a realizar con el MBDTR se tendrán los procedimientos básicos basados en MS-DOS (no necesarios para nuestro trabajo de tesis) para tener acceso al disco duro. Los procedimientos a probar serán los de lectura y escritura de datos en la base de datos de tiempo real. Estos podrán utilizarse tanto al inicializar el MBDTR, así como también podrán utilizarse durante la ejecución de la aplicación del usuario.

1.3. Trabajos Relacionados

En esta sección se realiza una comparación de una serie de sistemas de bases de datos en tiempo real comerciales, así como también se dará una breve descripción resumida de los sistemas de base de datos en tiempo real experimentales consideradas como puras.

1.3.1. Sistemas de Bases de Datos en Tiempo Real Comerciales

La investigación realizada sobre los diferentes sistemas de bases de datos en tiempo real comerciales nos sirvió como base para hacer un resumen comparativo considerando las diferentes características que tienen entre ellos. Considerando lo anterior se seleccionaron los siguientes sistemas que a continuación se describen:

Bases de Datos Investigadas

- ***Pervarsive.SQL*** .- Esta Base de Datos tiene tres versiones diferentes para los sistemas empotrados:
  - Pervarsive.SQL para Smart-Cards
  - Pervarsive.SQL para Sistemas Móviles
  - Pervarsive.SQL para Sistemas Empotrados

  La razón por la cual fue considerada en el estudio esta base de datos fueron los requerimientos bajos de memoria [1].

- ***Polyhedra*** .- Esta Base de Datos fue seleccionada por tres razones [2]:

  - Por ser una Base de Datos en Tiempo Real
  - Por su funcionamiento en Memoria Principal
  - Por su comportamiento activo

- ***Velocis*** .- Este Sistema en sus principios fue creada para *e-comerce* y aplicaciones Web, pero también tiene soporte para correr en Sistemas Operativos Empotrados [3].

- ***RDM*** .-Esta también es una Base de Datos en Tiempo Real. pero a diferencia de Polyhedra, esta es una librería empotrada y no soporta el modelo cliente/servidor [3].

- ***Berkeley DB*** .- Esta Base de Datos también es una librería empotrada y ofrece interesantes puntos de vista [4].

- ***TimesTen*** .- Es una Base de Datos Relacional, es igual a Polyhedra, es una Base de Datos que radica en Memoria Principal [5].

1.3.2. Características y Comparativo de los Sistemas de Base de Datos de Tiempo Real Comerciales

Los sistemas de base de datos en tiempo real investigadas deben contar con las siguientes características fundamentales, las cuales están relacionadas con: *el modelo DBMS que soportan, el modelo de datos que determina como se almacenan los datos físicamente, el indexamiento de datos que permite realizar búsquedas de datos a través de ciertas estrategias, el tamaño de memoria que ocupan, medios de almacenamiento usados, los sistemas operativos que soportan, y las técnicas usadas para el control de concurrencia.*

Modelo DBMS

Hay básicamente dos diferentes modelos de DBMS soportados (ver Tabla 1.1). El primer modelo es el cliente/servidor, donde el servidor de la base de datos puede ser considerado como una aplicación que corre por separado de la aplicación de tiempo real, muchas veces estas corren en el mismo servidor. El DBMS es invocado usando un protocolo de solicitud/respuesta. El servidor es accesado a través de la comunicación interproceso en el mismo procesador o entre procesos en una red. El segundo Modelo es cuando el DBMS esta compilado junto con la aplicación del sistema de tiempo real dándonos un ejecutable. Cuando una tarea quiere accesar la base de datos esta sólo realiza la función invocada para atender la solicitud.

Sistema DBMS	Cliente / Servidor	Librería
Pervasive.SQL	•	
Polyhedra		
Velocis	•	
RDM		•
Berkeley DB		•
TimesTen	•	

Tabla 1.1: Modelos DBMS soportados por los Sistemas BDTR

Modelo de Datos

El modelo de Datos concierne a como los datos están lógicamente estructurados. En casi todos los modelos, el que se usa es el Modelo Relacional, donde los datos están organizados en tablas con columnas y filas. Una ventaja de las bases de datos relacionales es que las columnas en una tabla pueden relacionarse con otras tablas arbitrariamente creando estructuras lógicas complejas. De esta estructura lógica, consultas pueden ser realizadas para extraer una selección específica de datos. Sin embargo una desventaja con este modelo es que al agregar datos produce carga al sistema. Existen otros modelos como es el orientado a objetos donde los objetos son compartidos entre las diferentes aplicaciones que corren en el sistema. Otro es el modelo Objeto-Relacional, este incorpora objetos en las relaciones. Como podemos ver en la tabla 1.2, todos los sistemas investigados son relacionales, excepto Berkeley DB.

Sistema DBMS	Relacional	OO	Obj.-Rel.	Otro
Pervasive.SQL	•			
Polyhedra	•		•	
Velocis	•			
RDM	•			
Berkeley DB				•
TimesTen	•			

Tabla 1.2 Modelos de Datos soportados por los Sistemas BDTR

Indexamiento de Datos

Para poder hacer eficientes las búsquedas de datos específicos, un sistema de indexamiento debe existir. Para búsquedas lineales a través de una única llave de una lista no ordenada no podría ser una buena solución ya que los tiempos de respuesta de una transacción crecería tanto como el número de datos crezca en la base de datos. Para resolver este problema dos soluciones son usadas: estructuras de árbol y listas hash. La estructura de árbol de indexamiento más usado son los B+-tree que son manejados en disco y el indexamiento T-tree el cual es usado en base de datos que se encuentran en memoria principal.
Las base de datos en memoria principal, como son Polyhedra y TimesTen usan hashing. Adicionalmente TimesTen soporta T-trees. Pervarsive, RDM y Berkeley DB usan indexamiento basado en B+-tree (ver tabla 1.3).

Sistema DBMS	**B+-tree**	**T-tree**	**Hashing**	**Otro**
Pervasive.SQL	•			
Polyhedra			•	
Velocis	n/a	n/a	n/a	n/a
RDM	•			
Berkeley DB	•		•	•
TimesTen	•	•	•	

Tabla 1.3 Estrategias de indexamiento de datos usados por los Sistemas BDTR

Requerimientos de Memoria

Los requerimientos de Memoria de la Base de Datos es un factor importante para las Bases de Datos Empotradas residentes en memoria en ambientes con pocos recursos de memoria.. Hay dos propiedades a considerar en las bases de datos empotradas: El primero es tamaño de la memoria con los datos dados por el fabricante. Segundo la sobrecarga de datos, esto es, el número de bytes requeridos para almacenar un elemento de dato aparte del tamaño de los datos que ocupa para una entrada en la lista indexada con un apuntador a la dirección física de almacenamiento es esto tipo como datos de sobrecarga. Comúnmente

soluciones cliente/servidor requieren mucha mas memoria que las librerías empotradas (ver tabla 1.4).

Sistema DBMS	**Requerimientos de Memoria**
Pervasive.SQL para Smart Cards	8kb
Pervasive.SQL para sistemas empotrados	50kb
Pervasive.SQL para sistemas móviles	50 – 400kb
Polyhedra	1.5 – 2Mb
Veloces	4Mb
RDM	400 – 500kb
Berkeley DB	175kb
TimesTen	5Mb

Tabla 1.4 Diferentes necesidades de memoria para los Sistemas BDTR

Medios de Almacenamiento

Los sistemas de cómputo empotrados soportan diferentes medios de almacenamiento. Normalmente, los datos usados en la computadora son almacenados en disco. Las computadoras de mano (Palms) usan memoria flash. Los datos en una base de datos necesitan ser persistentes, incluso con ausencia de energía. Casi todos los sistemas soportan flash y adicionalmente disco duro (ver tabla 1.5).

Sistema DBMS	**Disco duro**	**Flash**	**Smart - Card**	**FTP**
Pervasive.SQL	•	•	•	
Polyhedra	•	•		•
Velocis	•			
RDM	•			
Berkeley DB	•	•		
TimesTen	•			

Tabla 1.5 Medios de almacenamiento usados por los Sistemas BDTR

Plataformas de Sistemas Operativos

Diferentes sistemas empotrados pueden correr en varios sistemas operativos, aún teniendo diferencias en el ambiente de hardware. También la naturaleza de la aplicación hace la diferencia de cual sistema operativo podría se mas útil. Por lo que todas las base de datos empotradas deben poder correr sobre diferentes plataformas de sistemas operativos.
Los diferentes sistemas operativos soportados por las bases de datos en tiempo real que se han investigado básicamente se ubican en cuatro categorías:

- *Sistemas Operativos tradicionales que son usados por las computadoras de escritorio, como es el sistema operativo Windows de Microsoft , así como diferentes versiones de Unix y Linux.*

- *Sistemas Operativos para computadoras de mano (Palms). Aquí podemos encontrar a PalmOS y Windows CE.*

- *Sistemas Operativos de Tiempo Real. En esta categoría encontramos a VxWorks y QNX.*

- *Sistemas Operativos para Tarjetas Inteligentes (Smart-Cards). Estos sistemas son como Java Card y MultOS son hechos para ambientes muy pequeños, comúnmente no pasan de 32 KB.*

Casi todos los sistemas de bases de datos empotrados estudiados soportan un sistema operativo de tiempo real.

Sistema DBMS	**Sistemas Operativos de Tiempo Real**			
	VxWorks	*pSOS*	*LynxOS*	*QNX*
Pervasive.SQL para sistemas empotrados	•			•
Polyhedra	•	•		
Velocis				
RDM	•			
Berkeley DB	•			
TimesTen	•		•	

Sistema DBMS	**Hand-Held OS**		**Smart-Card OS**	
	PalmOS	*WinCE*	*Java Card*	*Mult OS*
*Pervasive.SQL para sistemas móviles y smart-card**	•	•	• *	• *
Polyhedra				
Velocis				
RDM				
Berkeley DB				
TimesTen				

Tabla 1.6 Sistemas Operativos soportados por los Sistemas BDTR (continuación)

Control de Concurrencia

El Control de Concurrencia usado en las transacciones serializables debe guardar las propiedades ACID. Todas las transacciones en el Sistema de Base de Datos deben cumplir todas las estas propiedades (ser: Atómicas, Consistentes, Aisladas y Durables).

Hay dos técnicas fundamentalmente diferentes que logran esta serialización, una es el optimista y el otro pesimista.

Para sistemas de tiempo real un número de variantes de estos algoritmos han sido propuestos que demandan estas base de datos [6,7]. Estos algoritmos pretenden encontrar un buen balance entre plazos perdidos y la inconsistencia temporal de las transacciones. Todas las base de datos excepto Polyhedra usa control de concurrencia pesimista (ver tabla 1.7).

Sistema DBMS	**Pesimista**	**Optimista**	**Ninguno**
Pervasive.SQL	•		
Polyhedra		•	
Velocis	•		
RDM	•		
Berkeley DB	•		•
TimesTen	•		

Tabla 1.7 Estrategias de Control de Concurrencia usados por los Sistemas BDTR

1.3.3. Características de los Sistemas de Base de Datos de Tiempo Real puros Experimentales

Hay un número considerable de sistemas de base de datos en tiempo real experimentales considerados como puros. Sin embargo aquí seleccionamos algunos de estos sistemas que son los desarrollos mas recientes y que han estado en constante estudio, pero que aún no se encuentran comercialmente disponibles. A continuación se da una descripción resumida de cada uno de ellos.

STRIP .- The **ST**anford **R**eal-Time **I**nformation **P**rocessor [8], es una base de datos en tiempo real desarrollada por la Universidad de Stanford, esta construida para un sistema operativo UNIX, es para ambiente distribuido y usa datos compartidos entre nodos. Esta es una base de datos activa que usa SQL3.

DeeDS .- Sistema de Base de Datos en Tiempo Real [9], soporta transacciones en tiempo real críticas y no críticas. Es desarrollada en la Universidad de Skövde, Suecia. Esta usa reglas ECA para lograr un comportamiento con propiedades de tiempo real. Su construcción toma ventaja de un ambiente multiprocesador.

BeeHive .- El sistema BeeHive [10], es una base de datos virtual desarrollada en la Universidad de Virginia, Charlottesville, US, en la cual los datos en la base de datos que puede estar en múltiples localidades y formas. La base de datos soporta cuatro diferentes interfaces: una interfase de tiempo real, una interfase para la calidad de servicio, una interfase para la tolerancia a fallas y interfase de seguridad.

REACH .- El sistema REACH [11], desarrollado en la Universidad de Dormstadt, Alemania, es una base de datos orientada a objetos activa desarrollada sobre Open OODB. Su meta es tener comportamiento activo sobre una base de datos Orientada a Objetos usando reglas ECA. Un modo de referencia es soportado para medir tiempos de ejecución de transacciones críticas. El sistema REACH esta proyectado para sistemas de tiempo real no críticos.

RODAIN .- El sistema RODAIN [12], desarrollado en la Universidad de Helsiki, Finlandia, es un sistema de base de datos de tiempo real firme que primariamente esta proyectado para telecomunicaciones. Esta diseñado para un alto grado de disponibilidad y tolerante a fallas. Está provisto para ajustarse a las características de las transacciones que se hacen en las telecomunicaciones, las cuales están identificadas como consultas y actualizaciones cortas y grandes actualizaciones masivas.

ARTS – RTBD .- Este sistema [13], es desarrollado en la Universidad Virginia, Charlottesville, US, soporta transacciones críticas y no críticas en tiempo real . Esta usa computo impreciso para asegurar la temporalidad de las transacciones. Esta construida sobre el sistema operativo ARTS.

1.4. Objetivos de la Tesis

1.4.1. Objetivos Generales

En cuanto a su utilidad, el objetivo es diseñar un Manejador de Base de Datos en Tiempo Real como parte de un Kernel de tiempo real robusto y capaz de interactuar con el mundo exterior para que a través de procesos concurrentes puedan manipular una base de datos como recurso compartido. Además de tener la propiedad de permitir el monitoreo, control de todos los procesos, así como controlar todas las transacciones que se realicen a la base de datos.
En cuanto a lo académico, se requiere que sea un sistema pequeño y que se ejecute en sistemas con recursos limitados de hardware, que sea fácil su migración a plataformas empotradas y que permita experimentar con él en otros tipos de plataformas (sistemas operativos y otros hardware). Además debe permitir incluir nuevas características en su arquitectura, pudiendo ser: alguna técnica eficiente de búsqueda de datos y el soporte para comunicaciones con otros sistemas. Así como también su posible integración con algún sistema de control para el monitoreo y la adquisición de datos que pudieran ser almacenado en un base de datos.

1.4.1. Objetivos Particulares

- Obtener el mejor tiempo de respuesta posible del Manejador de Base de Datos en Tiempo Real, reprogramando el timer del sistema.

- Evitar que el Kernel y el Manejador de Base de Datos en Tiempo Real se extiendan (en tamaño) con la finalidad de que permita ser portado a plataformas empotradas, las cuales que por su naturaleza cuentan con pocos recursos de hardware (memoria y dispositivos de almacenamiento secundario).

- Lograr el conocimiento profundo del tema y poder documentarlo de tal forma que pueda servir en un futuro para su migración del Kernel y del Manejador de Base de Datos en Tiempo Real en algún sistema empotrado.

- Integrar modularmente al Kernel de tiempo real el módulo para el control de concurrencia y el módulo del Manejador de Base de Datos de Tiempo Real, sin tener que hacer modificaciones importantes al Kernel de tiempo real.

1.5. Organización de la Tesis

La escritura de la tesis se desarrollo 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 que 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 es 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 de un sistema operativo en tiempo real que debe contener.

El capítulo 3 está dedicado por completo a explicar los mecanismos de planificación en los sistemas de tiempo real. Aquí se clasifican y muestran las restricciones de las tareas de tiempo real, se define el problema de la planificación. En este capítulo además, se clasifican las políticas de planificación en base a las prioridades y se da dos ejemplos por clasificación. Se explica Rate Monotonic y Deadline Monotonic para planificadores basados en prioridades dinámicas.

En el capítulo 4 se encuentra lo principal del Kernel de tiempo real usado en esta tesis, se muestra la arquitectura que tiene el Kernel, se presenta el BCP utilizado para las tareas del Kernel y cual es el comportamiento que va a tener dentro del mismo, además se explican cada uno de los estados de un proceso, así como sus transiciones entre estados. Aquí también encontraremos una explicación de las primitivas de procesos, las primitivas de semáforos, y las primitivas de buzón. Esto es todo lo usado respecto al Kernel de tiempo real para realizar el desarrollo del trabajo propuesto en esta tesis.

En el capítulo 5 esta dedicado exclusivamente a los sistemas de base de datos en tiempo real. Aquí se da una breve introducción a los sistemas de base de datos en tiempo real, la relación que guardan estas bases de datos y los sistemas de tiempo real. En este capítulo además se explican las diferentes características de las bases de datos de tiempo real, así como las características de las transacciones en estos sistemas. También se explica la relación que existe entre las base de datos de tiempo real y los sistemas de bases de datos activas, y por último en este capítulo se explica el procesamiento de las transacciones, la necesidad de la predecibilidad en la ejecución de las transacciones, así como el tratamiento de transacciones con plazo crítico y suave.

En el capítulo 6 se desarrolla la parte principal de este proyecto de tesis. Empezando con una breve explicación de la arquitectura general del sistema de tiempo real y el esquema general del Manejador de Base de Datos en tiempo real. También se explica la estructura general de un monitor, así como, el uso de las variables condición para la sincronización de tareas en un monitor. Se da una explicación de la estructura de una variable condición y de la estructura de un monitor. En este capítulo se explican y se desarrollan las primitivas utilizadas en un monitor, así como las primitivas del monitor de lectores-escritores que son los principales procedimientos para la sincronización de tareas lectoras y escritoras. Aquí también se explican cada uno de las primitivas que constituyen al manejador de bases de datos que se usan para la recuperación y actualización de registros de datos de la base de datos. Por último se describe la estructura utilizada por las tablas, la organización de la memoria, el manejo de errores en el Manejador de Base de Datos, y se explica como debe configurarse e inicializarse el Kernel y el Manejador de Base de Datos.

Por último, el capítulo 7 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 Manejador de Base de Datos de tiempo real como trabajo futuro con la finalidad de extender su funcionalidad y así su campo de aplicación.

Capítulo 2

Sistemas de Tiempo Real

2.1. Definición de Sistema de Tiempo Real

Un sistema de tiempo real (STR) es un sistema informático que interacciona repetidamente con su entorno y responde a los estímulos que recibe del mismo dentro de un plazo de tiempo determinado[14]. Para que el funcionamiento del sistema sea correcto no basta con que las acciones del sistema sean correctas, si no que además, tienen que ejecutarse dentro del intervalo de tiempo especificado. Los resultados deben producirse en los momentos en que aún tengan validez dentro del ambiente a controlar.

Los sistemas de tiempo real controlan un ambiente que tiene restricciones de tiempo bien definidas (ver figura 2.1), es por ello, que se vuelven mas 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 mas simples microcontroladores (tales como un microcontrolador para el control de un automóvil) hasta sistemas grandes y distribuidos (como los sistemas de control de tráfico aéreo). Se espera que en un futuro los sistemas de tiempo real sean aun más complejos y se utilicen en el control de estaciones espaciales, en sistemas integrados de visión/robótica/Inteligencia Artificial y en entornos peligrosos como plantas químicas, nucleares, etc.

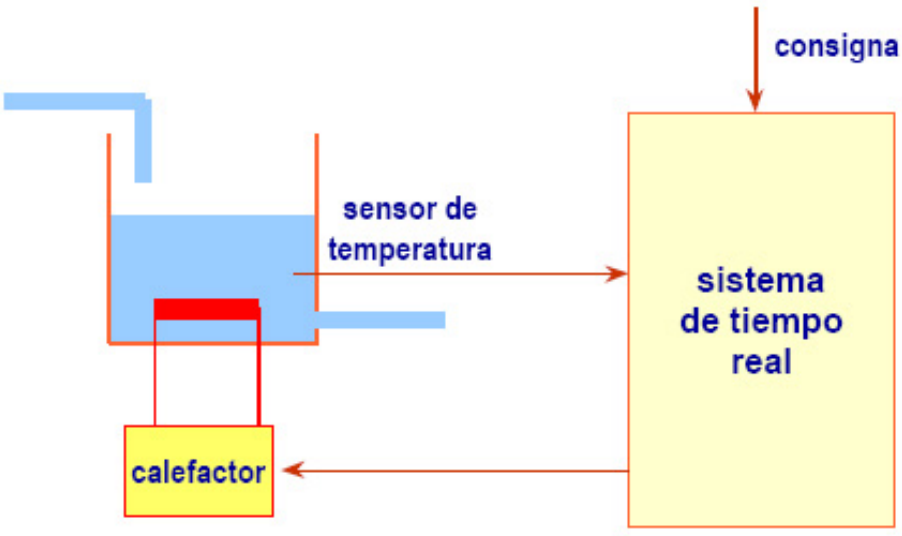

Figura 2.1: Sistema de Tiempo Real.

2.2. Aplicaciones de los Sistemas de Tiempo Real

En el campo de los sistemas de tiempo real se distinguen tres tipos 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 existen 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 analógico-digital convierten las señales generadas por el ambiente (analógicas) a una serie de datos que la computadora interpreta (digitales).

- **Reloj de Tiempo Real**: El reloj de tiempo real permite al sistema contar el tiempo en que se ejecutan acciones. De la misma forma, el reloj de tiempo real permite al sistema configurar los tiempos para la planificación de las tareas. Mediante el reloj de tiempo real es posible configurar interrupciones para controlar dispositivos externos, recibir y sincronizar señales de comunicación, y monitorear el

cumplimiento de los requerimientos temporales de las tareas del sistema. Sin el reloj de tiempo real no sería posible configurar los tiempos de ejecución de las tareas de tiempo real, y por tanto la planificación del sistema, ni tampoco sería posible saber si

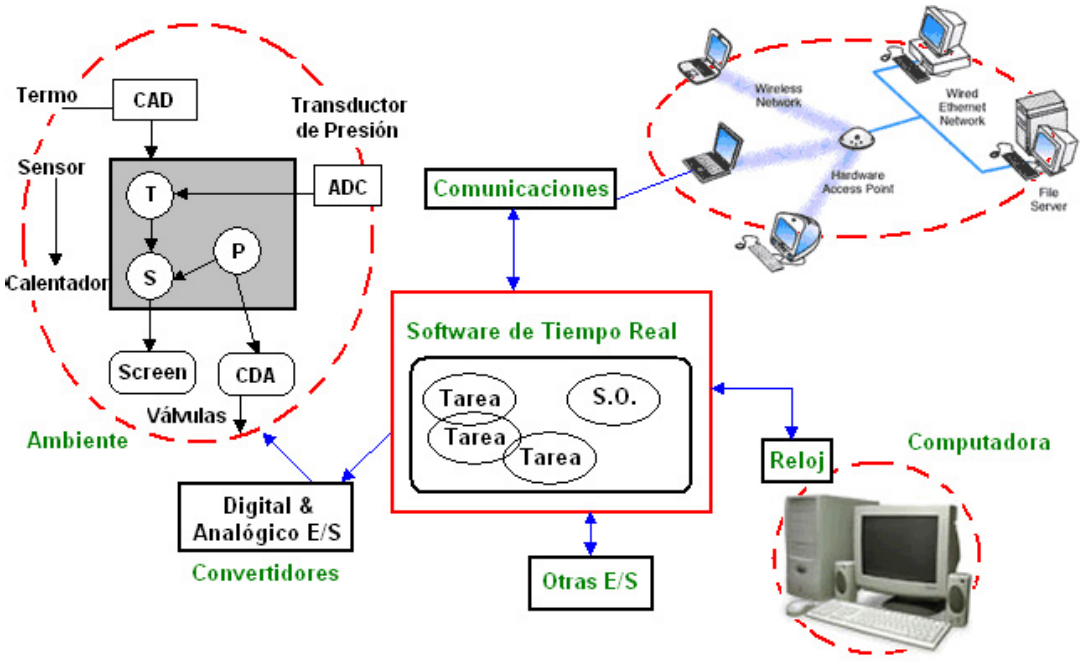

Figura 2.2: Elementos de un Sistema de Tiempo Real.

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 esta compuesto de un sistema operativo (o kernel) y de tareas las cuales son planificadas por el kernel. La estructura del kernel esta 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 ó Determinista**: No es fácil diseñar e implementar un sistema que garantice que la salida apropiada se generará en el tiempo adecuado bajo cualquier circunstancia, sin embargo los sistemas de tiempo real lo deben asegurar.

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

2.5. 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 mas conocida es aquella en la que distinguen a los sistemas de tiempo real en base a 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

[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

- 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.
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 mas que podría suceder es que una trama de sonido o de video se retrasara ó 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 esta. 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 este 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 este ocupe [15].
Un programa por si sólo 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 este 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 (τi) 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 retrazo en la terminación de una tarea o proceso con respecto a su plazo de respuesta. Si $L_i \leq 0$ la tarea (τ_i) no pierde su plazo.

- **Prioridad (P):** Es un número que representa la importancia del proceso o la tarea dentro del sistema. 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.

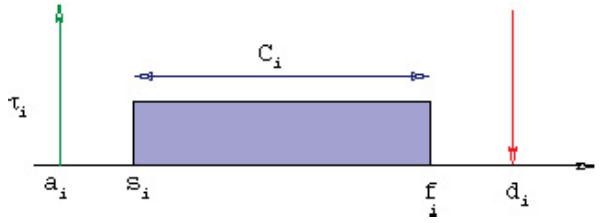

Figura 2.3: Parámetros de un Proceso de Tiempo Real

2.7.4. Estado del Proceso

Durante la ejecución de un proceso, su estado cambia y por lo tanto, el estado de un proceso esta 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 esta 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 mas 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.

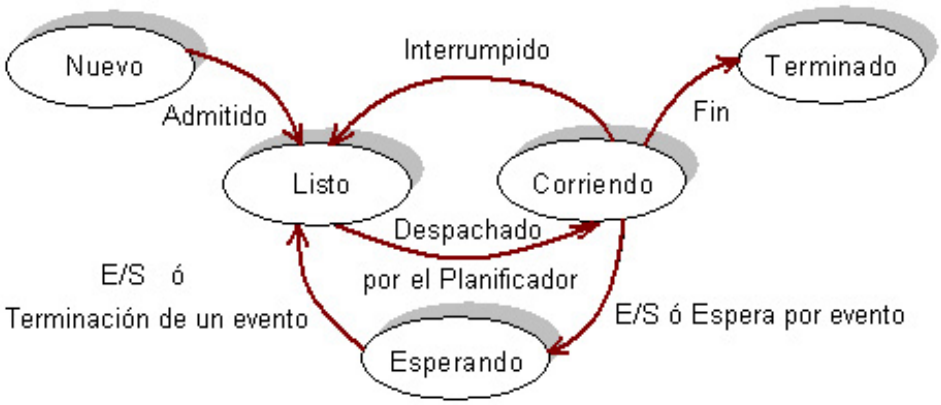

Figura 2.4: Diagrama de estado de los procesos

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 computo (Quantum) ó cuando algún proceso de mayor prioridad le quita el procesador.

- ***E/S ó Espera por evento***: Ocurre cuando el proceso necesita esperar por algún dispositivo de entrada/salida o por la llegada de un evento.

- ***E/S ó Terminación de un evento***: Es cuando el dispositivo o el evento que esperaba el proceso ya ocurrió y por lo tanto el proceso esta 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.

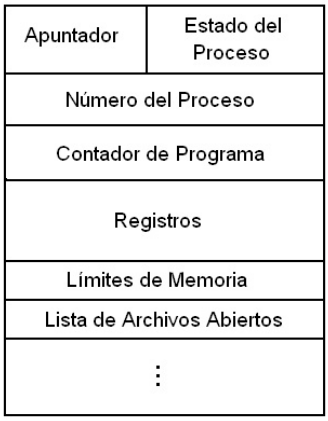

Figura 2.5: Bloque de Control de Procesos (PCB)

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 [16].

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 ó 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 incluidos 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 resguarda el estado de la tarea que está en ejecución.

2. Se utiliza al planificador para conocer que tarea es la siguiente a ejecutar.

3. Se recupera el estado de la siguiente tarea para que al finalizar el cambio de contexto sea ésta quien ahora haga uso del procesador.

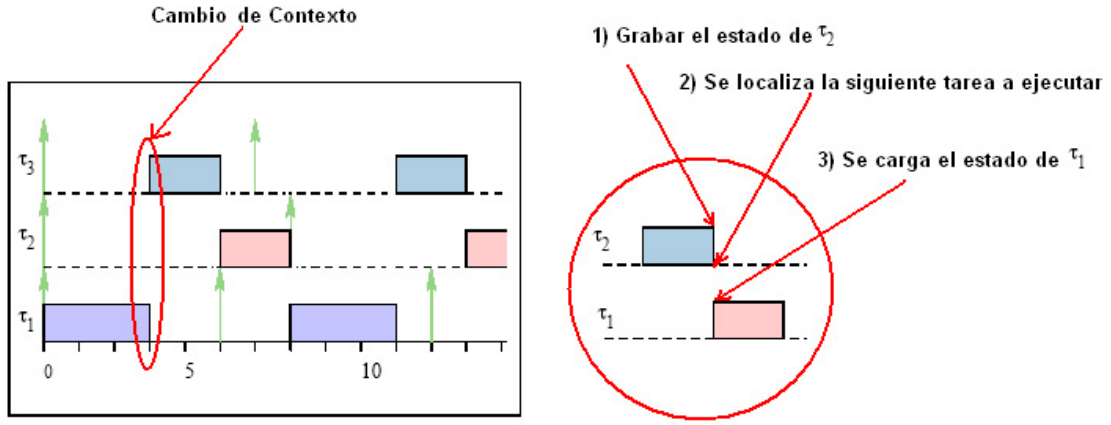

Figura 2.6: Cambio de Contexto.

El cambio de contexto es considerado como *sobrecarga* ya que durante el tiempo que este 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], depende 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)[15].

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 aquí y además le sirve para escribir información temporal. Los dispositivos de E/S la utilizan para leer y escribir mediante el DMA (Acceso Directo a Memoria). Las funciones que el manejador de memoria debe realizar son:

- Decidir que procesos deben cargarse en memoria cuando esta este disponible.
- Conocer que partes de la memoria están siendo utilizadas y por que procesos.
- Poder liberar y conceder espacio de memoria cuando se le requiera.

[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.

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

2.8.3. Manejador de Reloj

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

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

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

2.8.4. 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.

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 esta dentro la libera, dicho de otra manera, un proceso puede excluir temporalmente a todos los demás de utilizar un recurso compartido con el fin de asegurar la integridad del sistema.

Semáforos

El concepto de semáforo fue inventado por Dijkstra para solucionar el problema de la exclusión mutua. El mecanismo semáforo consta básicamente de dos operaciones primitivas, *señal (signal)* y *espera (wait)*[3] que operan sobre un tipo especial de variable semáforo, *S*. La variable semáforo puede tomar valores enteros y solo 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).

[3]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.

- ***Signal(S):*** Es una operación que incrementa el valor de su argumento semáforo (*S*)

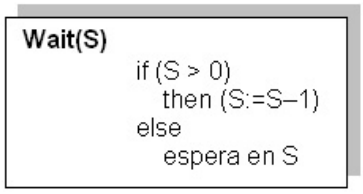

Figura 2.7: operación WAIT.

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).

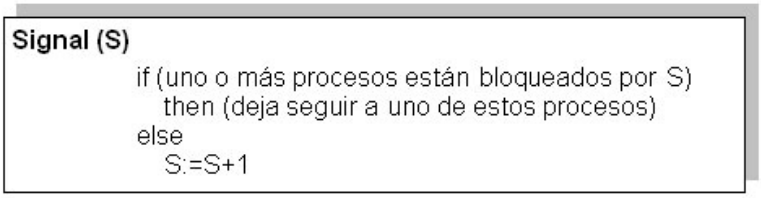

Figura 2.8: operación SIGNAL.

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**:* En los semáforos binarios los valores que pueden tomar son cero y uno debido a que el recurso que controla, 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 esta 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 0, 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 ó sin bloqueo.

- ***Semáforos con bloqueo:*** Son los que normalmente se utilizan, este tipo de semáforos dependiendo de la condición en WAIT pueden llegar a bloquear un proceso hasta que el recurso por el que espera 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 cual es) mientras que cuando se utiliza 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 esta es un ciclo el cual se encuentra constantemente revisando el valor de S en lugar de bloquear el proceso (ver figura 2.9).

```
Wait(S)
   loop
      if (S > 0)
         then (S:=S–1)
         exit
   end loop
```

Figura 2.9: operación WAIT en semáforos sin bloqueo.

Comunicación entre Procesos

Para que los procesos puedan comunicarse es necesario tener una forma de referenciarse unos con otros. Para lograr esto existen mecanismos de comunicación que operan a través de comunicación directa ó 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). Recibe 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 [17], 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 estos requieran. La comunicación entre los dispositivos y el sistema operativo es llevada a cabo principalmente a través de interrupciones.

El sistema de entradas y salidas consiste de:

- Un sistema de memoria caché 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 mas que un conjunto de reglas que determinan qué tarea se debe ejecutar en cada instante. Estos algoritmos deben tener en cuenta las necesidades de recursos y tiempo de las tareas de tal manera que éstas cumplan ciertos requisitos de prestaciones.
Los objetivos a alcanzar por toda política de planificación de tiempo real son:

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

3.1. Tareas de Tiempo Real

El sistema operativo de tiempo real en relación con la planificación, considera a las tareas como proceso 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 [18].

3.1.1. Clasificación de las Tareas de Tiempo Real

En base a 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 en base a la necesidad del cumplimiento de su plazo por parte de las tareas. La división de las tareas queda de la siguiente manera:

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

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

Otra característica relacionada con los parámetros temporales, concierne a *la regularidad de la activación* de las tareas. Tomando en cuenta la regularidad en la ejecución de las tareas de tiempo real, estas 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 solo durante una instancia de ejecución al termino de la cual desaparecen del sistema. Las tareas aperiódicas, no tienen restricciones críticas.

- **Esporádicas:** Las tareas esporádicas son tareas aperiódicas con restricciones temporales críticas (o duras). Si se monitorea el arribo de las tareas esporádicas es posible determinar una separación mínima entre activaciones consecutivas, lo cual podría permitir caracterizarlas como tareas periódicas (ver figura 3.1).

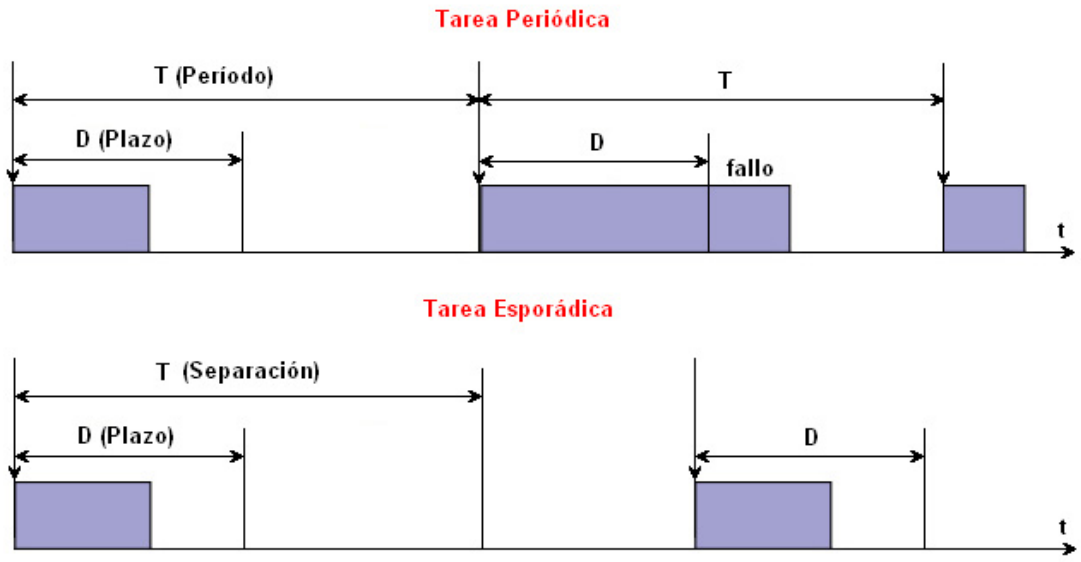

Figura 3.1: Tareas Periódicas y Esporádicas.

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 tarea de mayor prioridad. Por otro lado, las tareas no expulsables no podrán ser interrumpidas sino hasta que estas terminen su ejecución.

- **Con restricciones de precedencia:** Las tareas con restricciones de precedencia presentan un orden de ejecución con respecto a otras tareas del sistema. Si la tarea τ_i precede a la tarea τ_j , significa que la tarea τ_i solo 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 solo cuando τ_2 a terminado, mientras que τ_5 tendrá que esperar hasta que τ_2 y τ_3 terminen su ejecución.

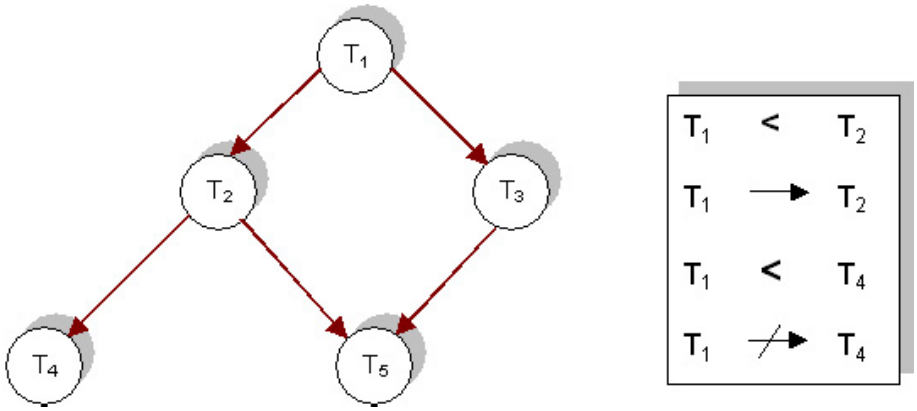

Figura 3.2: relación de precedencia para cinco tareas.

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

Para mantener la consistencia de datos en *recursos compartidos*, no debe permitirse el acceso simultáneo por dos o más tareas a estos datos. En este caso deben implementarse mecanismos de *exclusión mutua* entre las tareas que compiten por este recurso. Supongamos que R es un recurso compartido exclusivo para las tareas τ_a *y* τ_b. Si A es una operación realizada por τ_a sobre R, y B es la operación realizada por τ_b

sobre *R*, entonces *A* y *B* nunca pueden ejecutarse al mismo tiempo. Un fragmento de código ejecutado bajo restricciones de exclusión mutua se le conoce como *sección crítica*.

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

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, estas 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 ejecutivos cíclicos es la poca flexibilidad en el momento de modificar parámetros (añadir o borrar tareas), ya que esto conlleva a la re-elaboración de todo el plan.

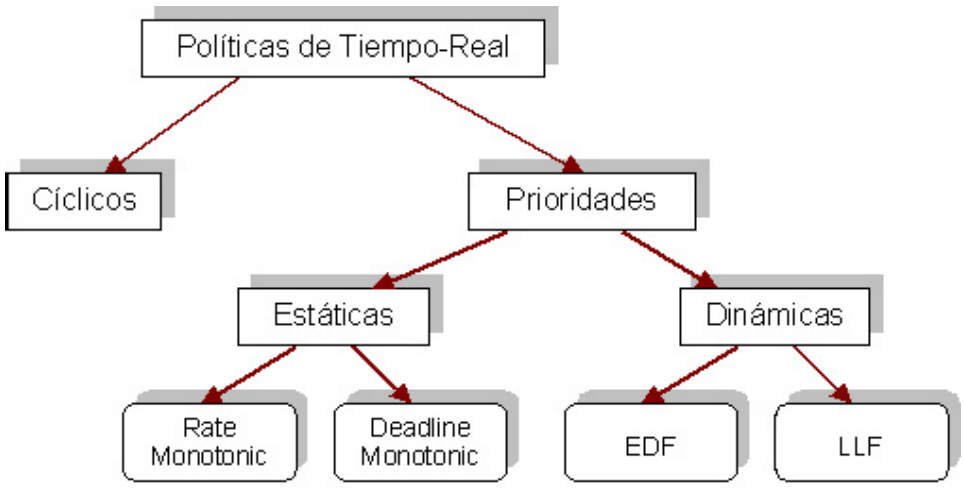

Figura 3.3: Clasificación de los algoritmos de planificación para sistemas de tiempo real.

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*.) ó 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 principales 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 periodos 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:

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

- El ciclo secundario debe ser menor o igual que el plazo de ejecución de cada tarea, $m \leq \{D_i\}$
- El ciclo secundario debe ser mayor o igual que el máximo de los tiempos de cómputo de las tareas, $m \geq max\{C_i\}$.
- El ciclo secundario debe ser divisor del período de cada tarea. Es decir, m divide a T_i
- Se debe 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:

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

Con los valores obtenidos para m se puede calcular el número de ciclos secundarios del ciclo principal mediante $N_{T_S} = T_M/m$. Para m = 10 se tiene $N_{T_S} = 10$, para $m = 20$ se tiene $N_{T_S} = 5$ mientras que para $m = 25$ se tiene $N_{T_S} = 4$. Como la complejidad aumenta con el número de ciclos secundarios se elige m = 25 con lo que se tiene 4 ciclos secundarios por ciclo principal. Otro dato que se puede calcular es el número de ejecuciones N_{e_i} de cada tarea τ_i en el ciclo principal, mediante la expresión $N_{e_i} = T_M / T_i$. En el ejemplo de la tabla 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.

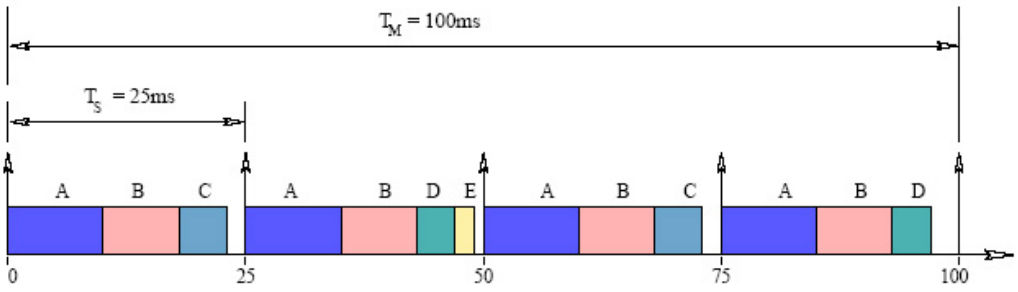

Figura 3.4: Planificación Cíclica.

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 [19] 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 [19], 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 mínima* $n(2^{1/n} - 1)$ *cumpla la siguiente condición:*

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

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

[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 solo una vez, al principio de la ejecución del sistema.

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:

$$U = \frac{C_1}{T_1} + \frac{C1}{T_2} + \frac{C1}{T_3} = \frac{10}{30} + \frac{10}{40} + \frac{12}{50} = \frac{494}{600} \approx 0.82 \tag{3.4}$$

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 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.

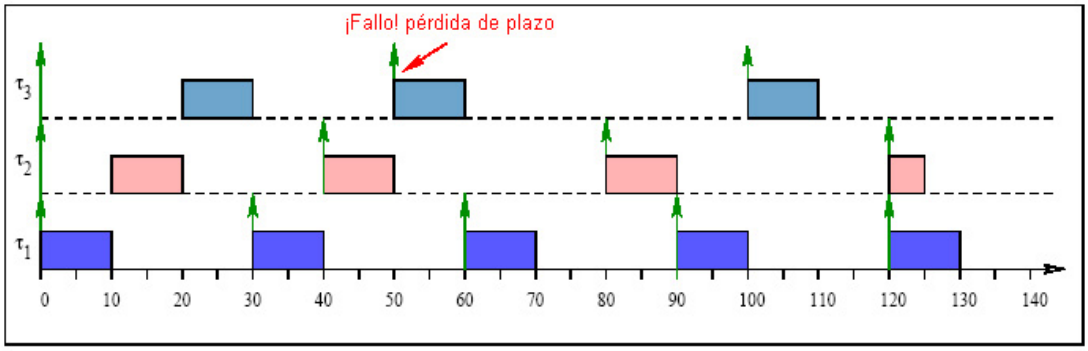

Figura 3.5: Conjunto de tareas que no satisface la Ecuación 3.2 y por lo tanto no es planificable.

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

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 utilizado 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.

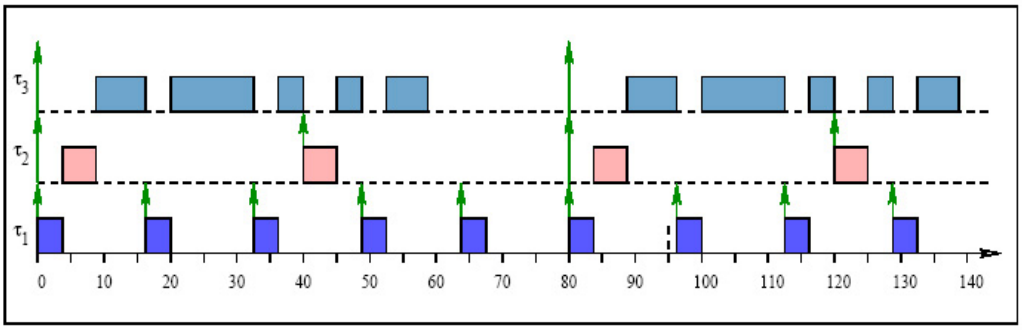

Figura 3.6: Conjunto de tareas que satisface 3.2 y por tanto es planificable.

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 estas hayan pérdido su plazo.

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

Tabla 3.4: Conjunto de tareas armónico, es planificable

Condición de Planificabilidad Suficiente y Necesaria

Lehoczky et al. [20] 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* $T_1 \leq T_2 \leq T_3 \leq ... \leq T_n$.

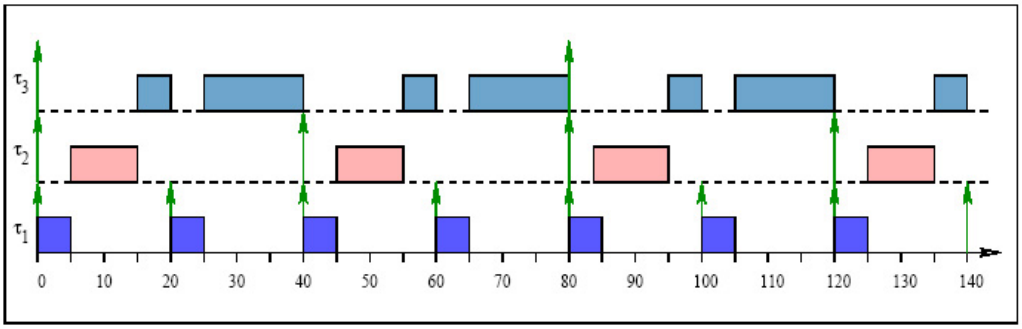

Figura 3.7: Conjunto de tareas que no satisface 3.2. Sin embargo, es planificable.

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

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

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

$$L = max_{\{1 \leq i \leq n\}} L_i \leq 1. \quad (3.6)$$

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*.

$$S_i = \{k \, . \, t_j \mid j=1, \ldots, i; k = 1, \ldots, [T_i / T_j]\} \quad (3.7)$$

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

$$W_i(t) = \sum_{j=1}^{i} C_i \, . \, [t/T_j]. \quad (3.8)$$

$$L_i(t) = W_i(t)/t. \quad (3.9)$$

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 < 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 es planificable.

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

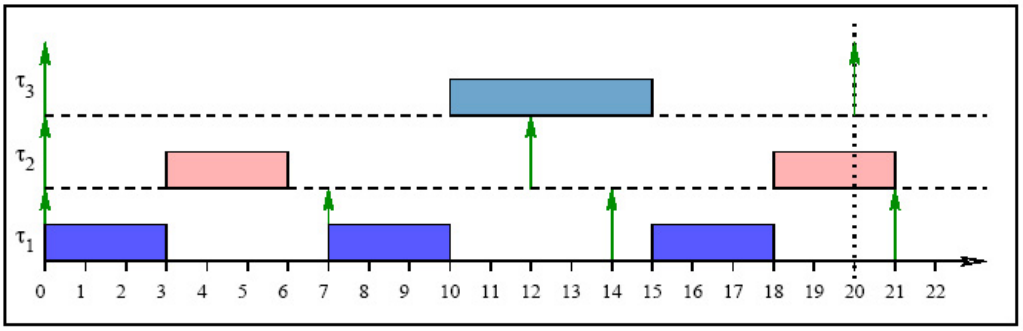

Figura 3.8: Puntos de Planificación.

3.5.2. Deadline Monotonic (DM)

En 1982 Leung y Whitehead [21] proponen un esquema de asignación de prioridades fijas denominado Deadline Monotonic (DM), el cual consiste en asignar la prioridad más alta las tareas que tengan plazos mas cortos. Es decir, la tarea con el plazo más corto se le asigna la prioridad más alta.

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

Teorema 3.5.4 *[21] 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 mas 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 que tareas pierden sus plazos de respuesta. Lo que es mas grave, es que en una sobrecarga, la perdida de plazos de algunas tareas puede producir un efecto en cascada de perdida de plazos de otras tareas.

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

3.6.1. Earliest Deadline First (EDF)

Es uno de los algoritmos más conocidos para el esquema de asignación dinámica es el algoritmo de planificación guiado por plazos (DDSA: Deadline Driven Scheduling Algorithm), al cual posteriormente se le denominó como el plazo más próximo primero (EDF: Earliest Deadline First).
En el algoritmo EDF las prioridades se asignan en forma dinámica. La política de asignación de prioridades consiste en asignar la prioridad mas alta a la tarea con plazo mas 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 %

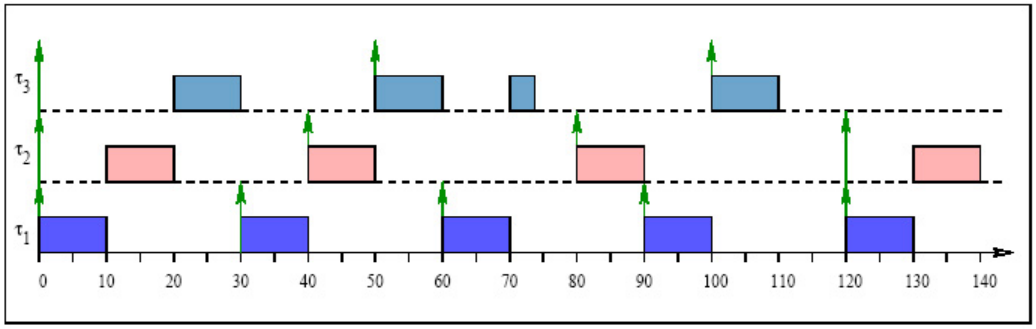

Figura 3.9: Planificación de las tareas de la tabla 3.7 bajo EDF.

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 *Di* en cualquier instante de tiempo *t* es:

$$holgura = D_i - t - C_i(t) \qquad (3.12)$$

donde $C_i(t)$, es el tiempo de cómputo pendiente por ejecutarse para que la tarea termine. Consideremos el siguiente conjunto de tareas $\Gamma = \{\tau_1=(6,3), \tau_2=(8,2), \tau_3=(70,2)\}$. 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.

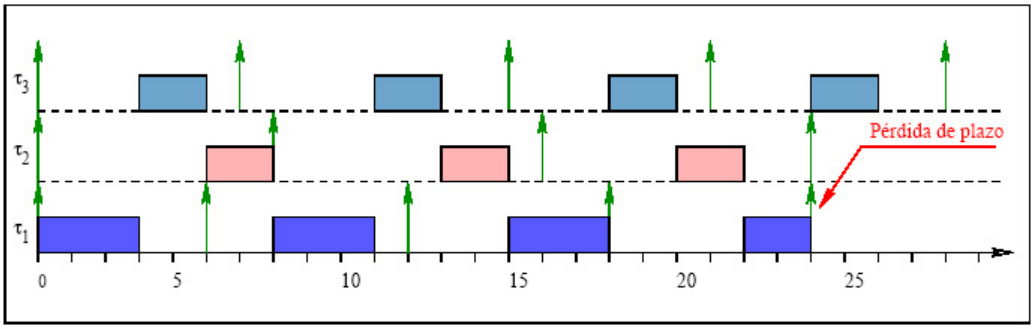

Figura 3.10: Planificación de tareas bajo el algoritmo LLF.

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 el algoritmo LLF si se necesita de dicho parámetro temporal. Esto es una desventaja debido a que la holgura es calculada en base al 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

Kernel de Tiempo Real

En la actualidad el avance de la tecnología ha permitido que cada vez mas, 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 operan correctamente, si no que además, se requiere que estos sean seguros y que respondan a eventos en un tiempo de respuesta acotado. En este proyecto se utiliza un kernel de tiempo real. 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 de la administración de la memoria, los procesos, las tareas y los discos.

En este capítulo se describen las características generales mas importantes del kernel de tiempo real para el control de procesos, así como también se describen cada una de las primitivas que comprende al sistema: primitivas de procesos y primitivas de tiempo. También se explican algunas consideraciones generales que ayudaran a comprender mejor como hacer uso de éste.

4.1. Arquitectura General

En la Figura 4.1 se presenta la arquitectura del Kernel en tiempo real utilizado. 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 sólo procesador, en el cual los procesos se ejecutan concurrentemente. El Kernel utilizado esta desarrollado en lenguaje de programación C, y se ejecuta en procesadores de la familia Intel 80x86 o compatible.

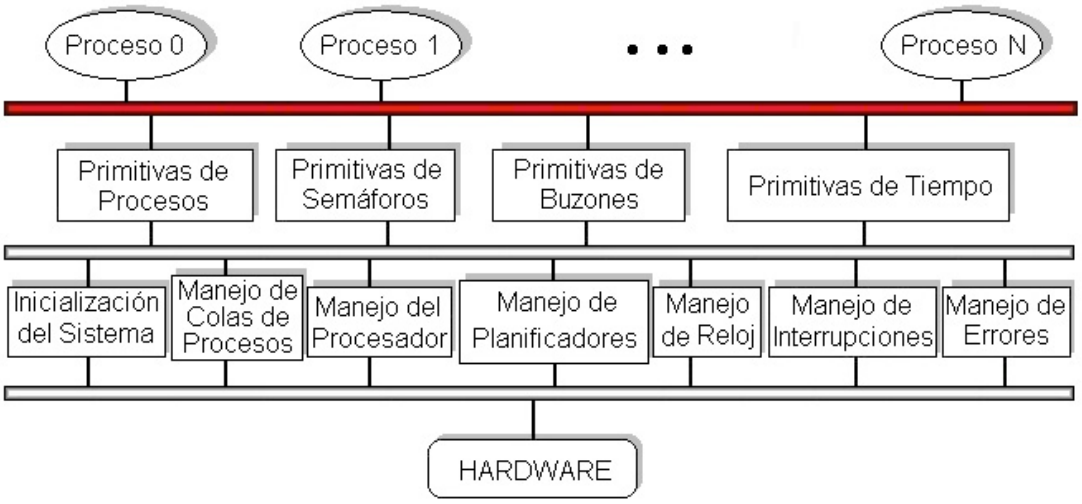

Figura 4.1: Arquitectura del Kernel.

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])

{ Deshabilitar interrupciones
Validar los parámetros

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

```
        Se llama al manejador de errores y excepciones;

        Si hay error en los parámetros entonces

        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 típica tarea puede incluir una sección de declaración 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, como él que se muestra a continuación

```
while(1) {
      código que se ejecuta periódicamente
}
```

el cuerpo de este ciclo es el código de la tarea que deseamos que se ejecute periódicamente. Dentro de este ciclo infinito, de manera opcional se puede utilizar la primitiva *Fin_de_ciclo* del kernel 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.

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 más alta , y la *última tarea* que tiene la prioridad más baja en el Kernel.

4.1.4. Procesos o Tareas

Inicialmente el Kernel esta configurado para trabajar con un máximo de 32 procesos, sin embargo el usuario en base a 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.

Primer Proceso

El proceso primero es la primer tarea que se ejecuta en el kernel, tiene la mayor

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

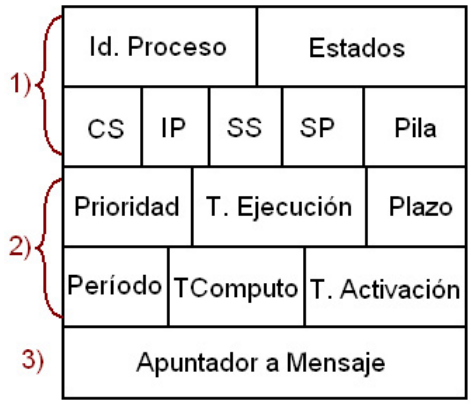

Tabla 4.2: Bloque de Control de Procesos en el Kernel

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 se elimina del sistema y no se vuelve a ejecutar jamás en el Kernel.

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) , este proceso no tiene mas 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.

4.1.5. BCP de los Procesos en el Kernel

Como se vió 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:

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 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

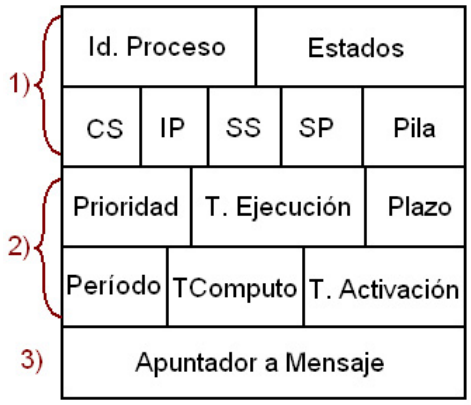

Figura 4.2: Bloque de Control de Procesos en el Kernel.

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).

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:

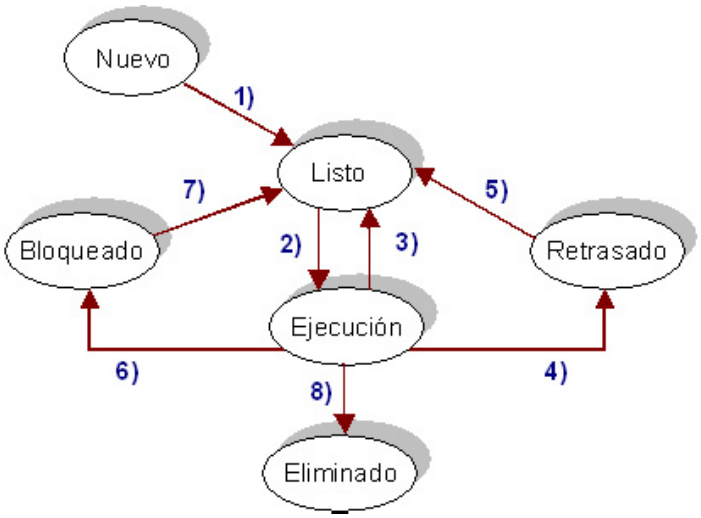

Figura 4.3: Estados de los Procesos en el Kernel.

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, este 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, estas no se pueden comunicar con el hardware, pero si 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 las primitivas de procesos son las siguientes:

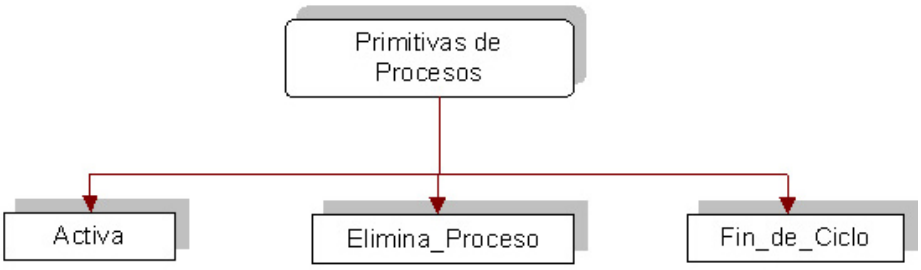

Figura 4.4: Primitivas de Procesos.

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.

Elimina Proceso :

NOMBRE: `void Elimina_Proceso().`

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

ENTRADAS: El proceso en ejecución (idposeedor).

SALIDAS: El proceso eliminado del Kernel.

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.

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.

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 las primitivas de tiempo “Retrasa” y “Revisa ProcRetrasado” (ver Figura 4.5). Las estructuras de estas primitivas de tiempo son las siguientes:

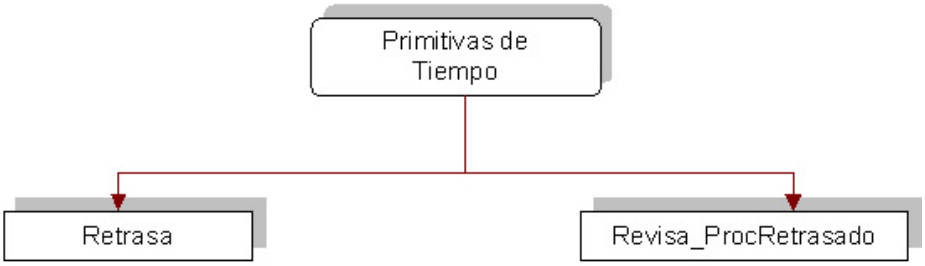

Figura 4.5: Primitivas de Tiempo.

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.

Revisa_ProcRetrasado :

NOMBRE: `void  Revisa_ProcRetrasado(void).`

FUNCIÓN: Actualiza el tiempo de retrasa y alista a todos los procesos que hayan cumplido con su tiempo de retraso.

ENTRADAS: Ninguna.

SALIDAS: Ninguna.

La primitiva **Revisa_ProcRetrasado** por su parte, se encarga de actualizar el tiempo de retraso que falta para activar los procesos y en caso de que alguno haya cumplido con su tiempo, este es alistado de nuevo colocándolo en la cola de procesos listos y cambiando su estado a “listo” .

4.2.3. Primitivas de Semáforos

Las cuatro primitivas utilizadas en el Kernel para manejar los semáforos son: “*CreaSe*m”, “*IsColSemVacia*”, “*Espera*” y “*Senial*”, las cuales se pueden ver Figura 4.6.

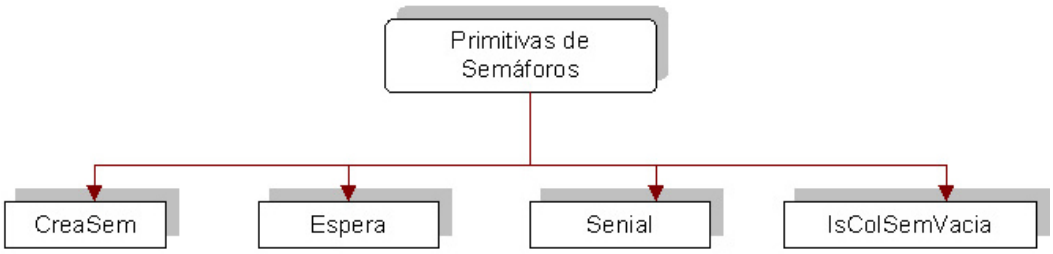

Figura 4.6: Primitivas de Semáforo.

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.

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.

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.

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 esta 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.

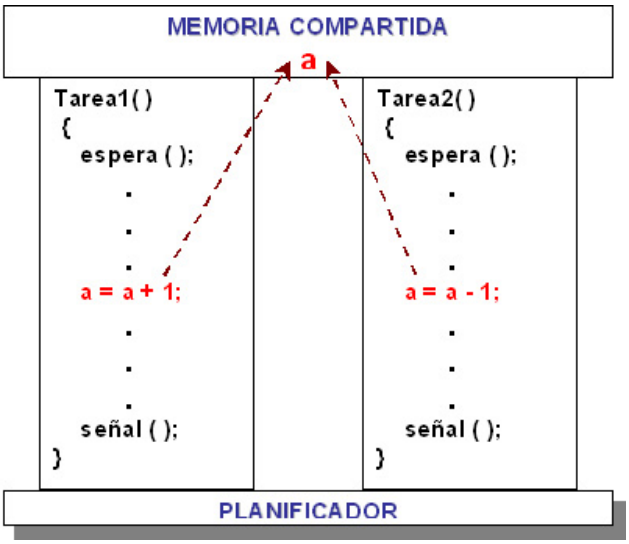

Figura 4.7: Utilización de los semáforos en el Kernel.

4.2.4. Primitivas de Buzón

Las primitivas de buzón, como se muestran en la Figura 4.8 son: "*CreaBuzon*", "*RecibirMje*" y "*EnviarMje*".

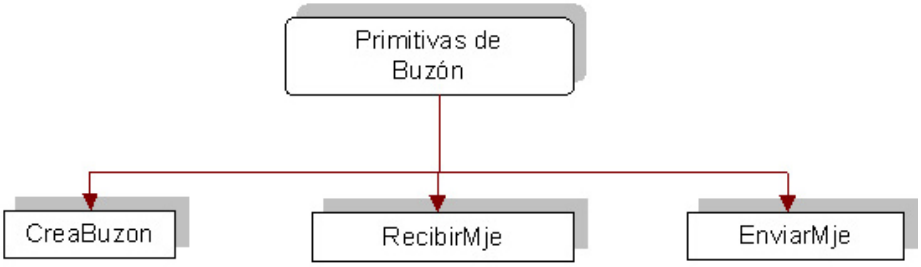

Figura 4.8: Primitivas de Buzón.

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).

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

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.

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 solo a procesos bloqueados esperando recibir mensajes, o solo 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.

Capítulo 5

Base de Datos en Tiempo Real

5.1. Introducción

Muchas aplicaciones del mundo real involucran restricciones de tiempo en el acceso a datos, así como también tienen validez temporal. Por ejemplo considere sistemas telefónicos, administración de redes, programas de manejo de inventario, fábricas para la elaboración automatizada de productos, y sistemas de comando y control. Todos estos involucran la obtención de datos del entorno, procesamiento y la generación de nueva información a partir de datos obtenidos en procesos anteriores, y a partir de ellos proporcionar una respuesta oportuna. Otro aspecto importante a considerar en estos ejemplos es que involucran el almacenamiento y procesamiento de datos temporales, los cuales pierden su validez después de un intervalo de tiempo.

Por ejemplo, considere el reconocimiento y direccionamiento de objetos en movimiento que van fijos sobre una banda transportadora en una fabrica. Algunas de las características del objeto son obtenidas por una cámara y permiten determinar el tipo de objeto y reconocer si tiene alguna anormalidad. Dependiendo de las características observadas, el objeto es dirigido al área de trabajo apropiada. Adicionalmente, el sistema actualiza la base de datos con la información referente al objeto. Los siguientes aspectos de

este ejemplo son notables. En primer lugar, deben coleccionarse las características de un objeto mientras el objeto pasa por enfrente de la cámara, las características obtenidas simplemente aplican al objeto delante la cámara, y estos pierden su validez una vez que un nuevo objeto entra al sistema. Entonces el objeto debe ser reconocido comparando sus características contra modelos de los diferentes objetos almacenados en la base de datos. Esta comparación tiene que ser terminada a tiempo, para que la orden de dirigir el objeto al destino apropiado pueda darse antes que objeto alcance el punto donde este vaya a ser dirigido a una banda transportadora y a una área de trabajo diferente. La actualización de la base de datos debe ser también completada a tiempo para que el sistema pueda mover al siguiente objeto para ser reconocido. Si por alguna razón o por restricciones de tiempo las acciones no son completadas dentro de los plazos establecidos, alguna acción alternativa podría ser tomada. En este ejemplo si la extracción de las características no son completadas en tiempo establecido, el objeto podría ser descartado y volver a la banda transportadora para pasar por el proceso nuevamente. *Las aplicaciones con estas características introducen la necesidad de utilizar los sistemas de bases de datos en tiempo real.*

Durante los últimos años, el área de las bases de datos de tiempo real ha llamado la atención de investigadores en ambas direcciones: sistemas de tiempo real y los sistemas de base de datos. La motivación de los investigadores de las bases de datos ha sido llevar muchos de los beneficios de la tecnología de las bases de datos para resolver problemas del manejo de los datos en sistemas de tiempo real.

El objetivo de este capitulo es señalar las características especiales, en particular los requisitos de consistencia temporal de los datos las en bases de datos de tiempo real, y mostrar cómo estas llevan a la imposición de restricciones de tiempo en la ejecución de una transacción. Considerando las características anteriores se propone adaptar y extender las características antes mencionadas en nuestro kernel de tiempo real descrito en el capítulo anterior.

5.1.1. Base de Datos y Sistemas de Tiempo Real

Las bases de datos tradicionales tratan con datos persistentes. Las transacciones accesan estos datos mientras es mantenida su consistencia. La seriabilidad es el criterio de

correctitud usualmente asociado con las transacciones y el cual nos permite garantizar la consistencia de la base de datos.

Los sistemas de tiempo real tratan con datos temporales. Debido a la naturaleza temporal de los datos y los requerimientos de tiempo de respuesta impuestos por el ambiente, las tareas de tiempo real poseen restricciones de tiempo (periodos o plazos) que deben cumplirse.

5.1.2. ¿Porque las Bases de Datos de Tiempo Real?

Las bases de datos combinan diferentes características que facilitan 1) la descripción de los datos, 2) el mantenimiento de la correctitud e integridad de los datos, 3) el acceso eficiente a los datos y 4) las ejecuciones correctas de las consultas, 5) las ejecuciones de las transacciones concurrentes y 6) el tratamiento de fallas. Específicamente

- los esquemas de las bases de datos ayudan a: la descripción de los datos y a reducir la redundancia.

- el soporte de manejo de datos, como es indexamiento que ayuda al acceso eficiente a los datos, y

- el soporte de transacción. Donde las transacciones tienen las propiedades ACID (Atomicidad, Consistencia, Aislamiento, y Durabilidad), estas aseguran la compleción de las ejecuciones de las transacciones concurrentes y asegura el mantenimiento de la integridad de datos incluso en la presencia de fallas.

Sin embargo, *el soporte para sistemas de base de datos de tiempo real* deben tomar en cuenta lo siguiente:

- No todos los datos en la base de datos son permanentes; algunos son temporales.
- Los planes serializables temporalmente correctos son un subconjunto del conjunto de planes serializables.

- La temporalidad es más importante que la correctitud, en muchas situaciones, la correctitud puede ser tratada temporalmente.
- La atomicidad puede ser tolerada.
- Muchas de las extensiones para la seriabilidad que han sido propuestas en bases de datos son también aplicables en bases de datos de tiempo real [22]. Algunas de estas asumen que el aislamiento de las transacciones no siempre es necesitado.

Muchas de las ventajas de la tecnología de las bases de datos, serían beneficiosas si podemos hacer uso de ellas para el manejo de datos obtenidos en sistemas de tiempo real. En un sentido similar, los adelantos realizados en sistemas de tiempo real para procesar tareas con restricciones de tiempo podrían ser aprovechadas para tratar transacciones con restricciones de tiempo en sistemas de bases de datos de tiempo real.

5.2. Características de los Datos en las Bases de Datos en Tiempo Real

Típicamente un sistema de tiempo real consiste de un sistema de control y un sistema controlado (ambiente). El sistema de control interactúa con el ambiente el cual a través de sensores obtiene datos. Es indispensable que el estado del ambiente percibido por el sistema de control sea consistente, de otra forma, los efectos de las tareas de los sistemas de control pueden ser desastrosas. De aquí, que el monitoreo oportuno del ambiente y el procesamiento oportuno de la información sensada sea necesario. Los datos sensados son procesados para derivar nuevos datos. Por ejemplo, el sensando de temperatura y presión información que pertenecen a una reacción pueden ser usados para derivar la proporción a la que la reacción esta progresando. Esta derivación dependerá de la temperatura y presión históricamente almacenda y por lo tanto alguna de la información necesitada pueda obtenerse de un archivo almacenado (una base de datos temporal). Basados en los datos derivados, se dan las órdenes al actuador.

También surge la necesidad de identificar requerimientos de precisión de tiempo en un sistema de tiempo real (base de datos) con el fin de poner los datos disponibles en el sistema de control para la toma de decisiones en las tareas. Por ejemplo, si la computadora que controla a un robot no ordena a tiempo que este pare o encienda, el robot podría chocar

con otro objeto que se encuentra sobre el piso de la fábrica. Este contratiempo puede ocasionar una mayor catástrofe.

La necesidad de mantener la consistencia entre el estado actual del ambiente y el estado reflejado por los contenidos de la base de datos conduce a la noción de la consistencia temporal. La consistencia temporal tiene dos componentes:

- Consistencia absoluta entre el estado del ambiente y su reflejo en la base de datos.

- Consistencia relativa entre los datos usados para derivar a otros.

Definamos formalmente estos dos componentes. Denotemos un elemento de datos en la base de datos de tiempo real por

d : (value, avi, timestamp)

donde el $\boldsymbol{d_{value}}$ denota el estado actual de ***d***, y el $\boldsymbol{d_{timestamp}}$ denota el tiempo en el cual la observación que relaciona a ***d*** fue hecha. $\boldsymbol{d_{avi}}$ denota el *intervalo de validez absoluto* o la longitud del intervalo de tiempo siguiente $\boldsymbol{d_{timestamp}}$ durante el cual se considera que ***d*** tiene validez absoluta.

Un conjunto de elementos de datos usados para derivar un nuevo elemento de dato forma un *conjunto de consistencia relativa*. Cada conjunto R esta asociado con un *intervalo de validez relativo* denotado por R_{rvi}. Se asume que $d \in R$**.**

d tiene un estado correcto, si y solo si

1. d_{value} es lógicamente consistente y por lo tanto satisface los restricciones de integridad.

2. *d* es temporalmente consistente, sí:

- Consistencia absoluta: $(current\text{-}time - d_{timestamp}) <= d_{avi}$
- Consistencia relativa: para toda $d' \in a\ R,\ |d_{timestamp} - d'_{timestamp}| <= R_{rvi}$

Considere el siguiente ejemplo: suponga $temperature_{avi} = 5$, $pressure_{avi} = 10$, $R = \{ temperature, pressure \}$, y $R_{rvi} = 2$. Si $current\text{-}time = 100$, entonces (a) $temperature = \{ 347, 5, 95 \}$, y $pressure = \{ 50, 10, 97 \}$ son temporalmente consistentes, pero (b) $temperature = \{ 347, 5, 95 \}$ y $pressure = \{ 50, 10, 92 \}$ no lo son. En (b), aunque los requisitos de consistencia absoluta se cumplen, la consistencia relativa de *R's* es violada. Sustituyamos valores para comprobar la consistencia temporal:

	Consistencia absoluta	*Consistencia relativa*
(a) temperature	*100-95 <= 5*	
(a) pressure	*100-97 <= 10*	
(b) temperature	*100-95 <=5*	*temperature: |95-95| <= 2*
(b) pressure	*100-92 <=8*	*pressure: |97-92| <= 2 no se cumple*

Los requisitos de consistencia temporal se traducen entonces en restricciones de tiempo sobre las transacciones.

Así, para satisfacer la consistencia lógica usamos el control tradicional de concurrencia y para satisfacer los requisitos de consistencia temporal se consideran los tiempos conocidos para la administración de transacciones con restricciones de tiempo.

5.3. Características de las Transacciones en Sistemas de Bases de Datos de Tiempo Real

Las transacciones pueden ser caracterizadas basándose en la naturaleza de las transacciones en los sistemas de base de datos de tiempo real, por la forma en la cual los datos son usados por las transacciones, y por las restricciones del tiempo, esto es, la importancia de la ejecución de la transacción en su plazo.

Los sistemas de base de datos de tiempo real realizan tres tipos de transacciones comunes en las bases de datos tradicionales:

- *Transacciones de solo escritura,* obtienen el estado del entorno y escriben en la base de datos.
- *Transacciones para actualización*, derivan nuevos datos y los almacena en la base datos.
- *Transacciones de solo lectura,* leen datos de la base de datos y se envían a los actuadores.

Esta clasificación puede ser usada para adaptarse a los esquemas de control de concurrencia en los sistemas de base de datos de tiempo real.

Algunas restricciones de tiempo vienen de los requerimientos de consistencia temporal y algunas vienen de los requerimientos impuestos en el tiempo de reacción del sistema. Los requerimientos de periodicidad comúnmente tienen la siguiente forma:

Cada 10 segundos muestra la velocidad del viento.
Cada 20 segundos actualiza la posición del robot.

Mas adelante se mostrará como los requerimientos de periodicidad pueden derivarse del *avi* de los datos.

Los requerimientos de reacción del sistema comúnmente toman la forma de la restricción del plazo impuesto en las transacciones aperiódicas: Por ejemplo,

If temperatura > 1000
en 10 segundos agregar refrigerante al reactor.

En este caso, la acción del sistema en respuesta a la alta temperatura debe ser completada en 10 segundos.

Las transacciones también pueden ser distinguidas basadas en los efectos de la perdida del plazo de una transacción. Considerando los diversos tipos de función que pueden asociarse a las transacciones, estas se categorizan de la forma siguiente:

- Transacciones de Plazo Crítico, son aquellas que pueden resultar en una catástrofe si el plazo falla.

- Transacciones de Plazo *Suave,* son aquellas que tienen algún valor aún después de un cierto tiempo de su plazo. Si el valor desciende hasta cero y alcanza su plazo, entonces obtenemos una transacción de plazo *firm*, pero si su plazo expira entonces no se toma en cuenta.

Una vez que se han identificado las restricciones de tiempos es importante entender como las transacciones son planificadas. Ahora veremos como los requisitos de validez absoluta en los datos inducen a requisitos de periodicidad.

Las transacciones periódicas son planificadas para que cada instancia de una transacción sea garantizada para empezar al inicio de un periodo. Entonces, el peor caso de separación entre el tiempo de inicio de una instancia y el tiempo de finalización de la siguiente instancia será (*P + (2*e)*). De aquí una transacción podría escribir el dato en cualquier tiempo durante esta ejecución, el intervalo (P + (2*e)) debe ser menor que el *avi* dado. Esto es, *P = (avi – (2*e)*).

La meta principal es asegurar que en un intervalo donde una transacción se ejecuta, desde el punto donde se mantiene la consistencia relativa hasta el fin del intervalo, haya suficiente tiempo para que la ejecución de la transacción se complete. Además, cuando tengamos una serie de datos derivados, la alternativa para una transacción es usar los *rvi* para imponer restricciones de precedencia sobre las acciones derivadas.

5.4. Relación de las Bases de Datos de Tiempo Real con Bases de Datos Activas

Esta sección trata de las distinciones específicas entre las bases de datos activas y los bases de datos de tiempo real.

La construcción básica de un bloque en una base de datos activa es la siguiente:

ON *evento*
IF *condición*
DO *acción.*

En la ocurrencia del *evento* especificado, si la *condición* se mantiene, entonces la *acción* especificada puede realizarse. Este constructor provee un buen mecanismo por el cual las restricciones de integridad de los datos puede mantenerse. El *evento* puede ser arbitrario, incluyendo eventos externos (como en el caso de eventos en tiempo real generados por el ambiente), eventos temporales, o eventos relacionados con transacciones (como es el inicio y confirmación de transacciones). La *condición* puede corresponder a condiciones del estado de los datos o del entorno. Se dice que la *acción* es *activada* [23,24] y esta puede ser una transacción arbitraria.

Las bases de datos activas proveen un buen modelo para el arribo de tareas periódicas/aperiódicas basadas en eventos y condiciones. Aunque la estructura anterior implica que una base de datos activa pueda reaccionar a las interrupciones, las restricciones de tiempo no son explícitamente consideradas por la transacción subyacente en el mecanismo de procesamiento.

La deficiencia principal en bases de datos activas es la ausencia de restricciones de tiempo en la compleción de las transacciones, por lo que esto hace necesario que las bases de datos activas consideren restricciones de tiempo.

Considere un sistema que controla el aterrizaje de un avión. Para asegurar que una vez que se a tomado la decisión de aterrizar; los pasos necesarios para bajar las ruedas, empezar la desaceleración, y para reducir altitud, se completa dentro de una duración dada, digamos 10 segundos. Aquí los pasos pueden depender de la ruta de aterrizaje, las restricciones específicas para el aeropuerto y el tipo de vuelo y puede involucrar acceso a una base de datos que contiene la información pertinente. En esas situaciones donde los pasos necesarios no se han completado a tiempo, nos gustaría que el aterrizaje fuera abortado dentro de un plazo dado, digamos 5 segundos; el aborto debe ser *completado* en este plazo, presumiblemente porque este es el “colchón” disponible para que el sistema tome las acciones alternativas. Este requerimiento puede ser expresado como sigue:

ON (*10 segundos después* “iniciados los preparativos de aterrizaje”)
  IF (los pasos no se completaron)
    DO (*en 5 segundos* “abortar el aterrizaje”)

En resumen, mientras las bases de datos activas poseen las características necesarias para tratar con muchos aspectos de los sistemas de bases de datos de tiempo real*, el ingrediente crucial faltante es la persecución del procesamiento oportuno de las acciones.*

5.5. Procesamiento de Transacciones en Sistemas de Base de Datos de Tiempo Real

Ahora se considerará que las transacciones y procesamiento de consultas tienen restricciones de tiempo y que hay diferentes consecuencias de no satisfacer esas restricciones. Una característica principal al problema del procesamiento de una transacción es la predecibilidad.

5.5.1 La necesidad de la Predecibilidad

Si una *transacción de plazo crítico* en tiempo real falla en su plazo, ésta tiene consecuencias catastróficas. Considerando esto, es necesario predecir de antemano que las transacciones se completarán antes de que finalicen su plazo. Esta predicción será posible sólo si conocemos el peor tiempo de ejecución de una transacción, los datos y las necesidades de recursos de la transacción. Además, es deseable tener pequeñas variaciones entre los casos peores de predicción y las necesidades reales. La *predecibilidad* también es importante para las *transacciones de plazo suave*, aunque en menor grado. Conociendo de antemano al inicio de una transacción que esta no puede completarse en su plazo permitirá al sistema descartar la transacción, y así, evitará gastar tiempo en la transacción y ninguna sobrecarga en la recuperación se incurra.

En un Sistema de Base de Datos, existen un número de fuentes de impredecibilidad:

- Dependencia de la secuencia de ejecución de las transacciones sobre los valores de los datos;
- Datos y conflictos de recursos;
- Paginación dinámica y I/O; y

- Aborto de transacciones y la restauración de resultados y reinicialización.

Las bases de datos distribuidas tienen problemas adicionales retardos de comunicación y las fallas de sitios.

El análisis fuera de línea de una transacción es deseable porque este provee un estimado de requerimientos de tiempos de cómputo, datos y recursos. Para obtener la información necesaria de una transacción, se deben de tomar en cuenta ***cuatro fuentes de impredecibilidad***. Las transacciones tienen dos fases. La primera fase es la pre-carga, es donde inicia la ejecución de la transacción para obtener los datos necesarios para su procesamiento posterior, no se consideran escrituras y los conflictos con otras transacciones. La demanda de los tiempos de cómputo de la transacción también son determinados en esta fase. Es posible que otras transacciones puedan cambiar los datos sin alterar el progreso de esta fase [25]. Una vez que se han completado los requerimientos se va a la segunda fase donde se puede garantizar la ejecución completa de la transacción en sus restricciones de tiempo. Para garantizar que la transacción se complete en su plazo, se construye un plan de ejecución que toma en cuenta los requerimientos de tiempo, recursos y datos. Si el plan no puede ser construirse, la transacción es abortada, sin reiniciarla incluso [26].

Si al final de la fase de pre-carga se detecta que el estado de los datos ha cambiando, entonces la fase de pre-carga puede ser reejecutada. En cualquier caso, si los datos no cambian se puede garantizar que una transacción se completará en su plazo y ninguna acción de recuperación son necesarias si una transacción se inhabilita para ejecutarse. El precio de esta última situación es la sobrecarga de la fase de pre-carga. Diferentes optimizaciones son posibles, para mas detalles revisar [27].

5.5.2. Tratando con Plazos Críticos

Todas las transacciones con plazos críticos deben satisfacer sus restricciones de tiempo. Por consiguiente el manejo dinámico de transacciones no puede garantizarlo, los datos y el procesamiento de recursos así como el tiempo necesitado por las transacciones

tiene que ser garantizado y estar disponible cuando sea necesario. Esto tiene diferentes implicaciones.

Primero, tenemos conocimiento de cuando las transacciones sean probablemente invocadas. Esta información esta disponible para transacciones periódicas, pero no para las transacciones aperiódicas. La separación mas pequeña de tiempo entre dos ocurrencias de una transacción aperiódica puede ser vista como periodo. De aquí que, podemos tratar a todas las transacciones con plazo crítico de tiempo real como transacciones periódicas.

Segundo, para asegurar a priori que sus plazos se cumplan, tenemos que determinar sus requerimientos de recursos y los peores casos de tiempos de ejecución de una transacción.

Una vez que hemos logrado lo anterior, podemos tratar a las transacciones de manera similar como se tratan a las tareas periódicas en los sistemas de tiempo real, i.e., usando planificadores con manejo de tablas estáticas o manejo de prioridad preemptivo. Además podemos utilizar herramientas para el análisis de la planificalidad, y por consiguiente verificar si el conjunto de transacciones son planificables considerando sus periodos y requerimientos de datos [28].

5.5.3. Tratando con Plazos Suaves

Las transacciones con plazos suaves de tiempo real, tienen mas tiempo para procesarlas por lo tanto no requerimos encontrar los plazos todo el tiempo. La meta es encontrar el plazo de la transacción adoptando políticas de asignación de prioridad y mecanismos para la resolución de conflictos que explícitamente toman en cuenta el tiempo. La asignación de prioridad se dá dentro de la planificación de la CPU y los mecanismos de resolución de conflictos determinan cual de las transacciones que contienden por datos obtendrán el acceso.

5.5.3.1. Asignación de Prioridades y Resolución de Conflictos

Los sistemas de Bases de Datos de Tiempo Real asignan prioridades a las transacciones basadas en plazos y por el valor que concede al sistema. Las posibles políticas de asignación son:

- Earliest-deadline-first
- Highest-values-first
- Highest-value-per-unit-computation-time-first
- Longest-executed-transaction-first

En [33] se muestra que la política de asignación de prioridad tiene impacto significativo en el desempeño y que cuando las diferentes transacciones tienen valores diferentes, plazo y valor deben ser considerados.

Para la resolución de conflictos en sistemas de base de datos en tiempo real, varias extensiones a los protocolos: bloqueo de dos fases, optimista y marcas de tiempo han sido propuestas en [29,30,31,32,33,34,35,36,37,38].

Capítulo 6

Manejador de Base de Datos en Tiempo Real

---

*Un Sistema de Base de Datos en Tiempo Real (**SBDTR**) puede ser definido como un sistema de base de datos que esta diseñado para dar respuesta en tiempo real a las transacciones de aplicaciones con acceso intenso a los datos,* como son: sistemas de cómputo integrados para manufactura, bolsa de valores, bancos, y sistemas de comando y control. Similar a las tareas en los sistemas de tiempo real, las transacciones que se procesan en un SBDTR tienen restricciones de tiempo conocidas como plazos. Lo que hace diferente a un SBDTR de los sistemas de tiempo real convencionales es el requerimiento de preservar la consistencia de los datos garantizando los requerimientos de tiempo real de las transacciones. Como podemos notar esto hace más compleja la planificación de las transacciones en un SBDTR. Los protocolos de control de concurrencia que se usan para preservar la consistencia de los datos en un sistema de base de datos tradicional están basados en el bloqueo y reinicio de la transacción, lo cual hace virtualmente imposible predecir los tiempos de cómputo y hacen difícil establecer planes de ejecución que garanticen los plazos. En este proyecto de tesis se desarrolló un *Manejador de Base de Datos en Tiempo Real* el cual forma parte de un kernel de tiempo real.

En este capítulo se describe la filosofía de diseño que se llevó a cabo para implementar el *Manejador de Base de Datos en Tiempo Real* (MBDTR), así como la implementación de

cada una de sus partes. También se explican las consideraciones de inicialización de la base de datos y algunos otros puntos importantes que ayudaran a comprender mejor como fue el diseño.

6.1. Arquitectura General

En la figura 6.1 se presenta la arquitectura general del sistema de 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 el penúltimo nivel se localiza el *Monitor* y *el Manejador de Base de Datos en Tiempo Real*. En la parte superior, se encuentran los procesos del usuario quienes pueden interactuar con el Kernel o con el MBDTR.

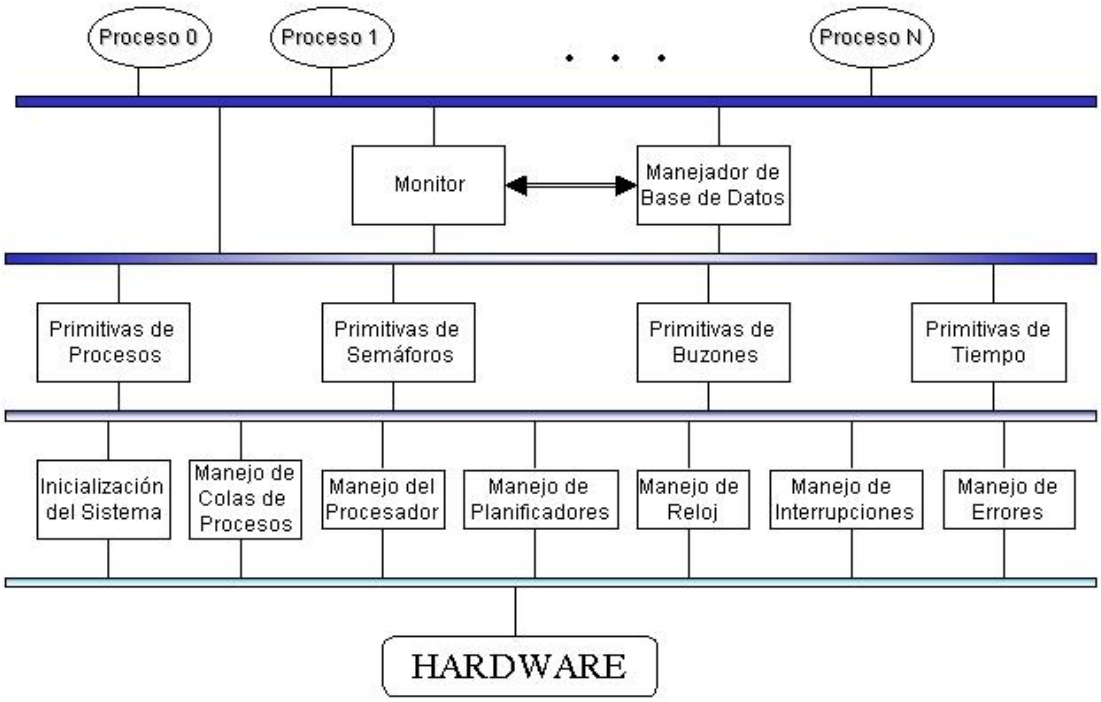

Figura 6.1: Arquitectura General

El esquema de la figura 6.2 presenta la extensión de los bloques Monitor y Manejador de Base de Datos mostrados en la figura 6.1. Estos incluyen: las primitivas del Monitor, las

primitivas del Monitor de Lectores-Escritores y las primitivas del Manejador de Base de Datos en Tiempo Real.

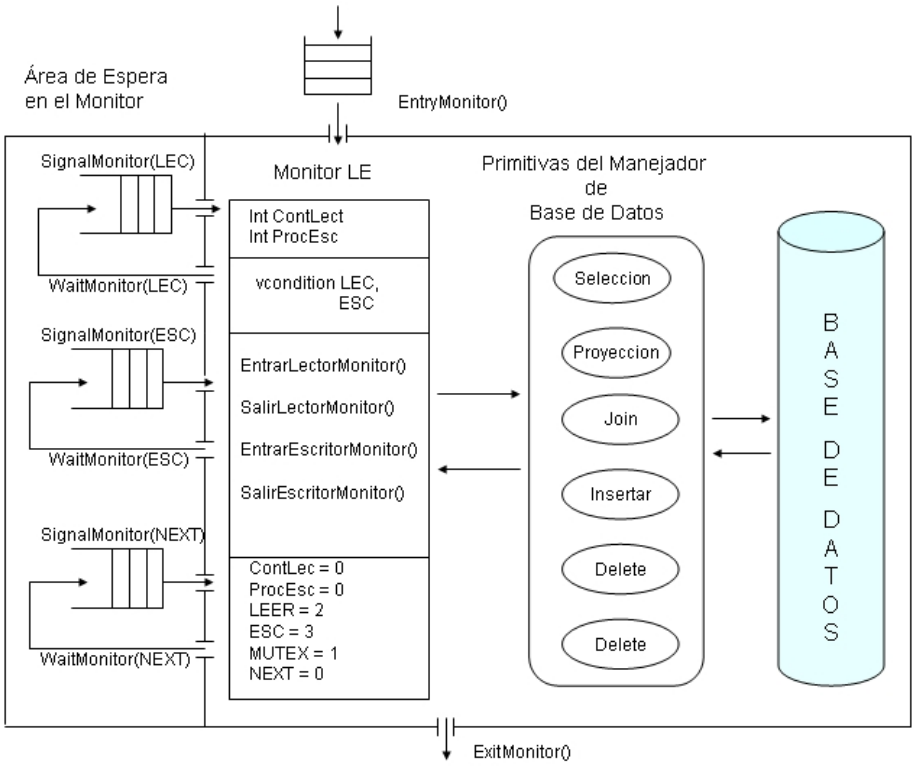

Figura 6.2: Esquema general del Manejador de Base de Datos

6.1.1. Estructura general del Monitor

Un *Monitor* es una construcción de lenguaje de alto nivel que provee una estructura para la eficiente sincronización de procesos. Los monitores encapsulan las representaciones abstractas de los recursos y proveen un conjunto de operaciones que son el único medio por el cual la representación es manipulada. Expresando esto de otra manera, un *Monitor* contiene variables que almacenan el estado de los recursos y los procedimientos que operan sobre el recurso. Una declaración *Monitor* tiene la estructura general como se muestra en la figura 6.3.

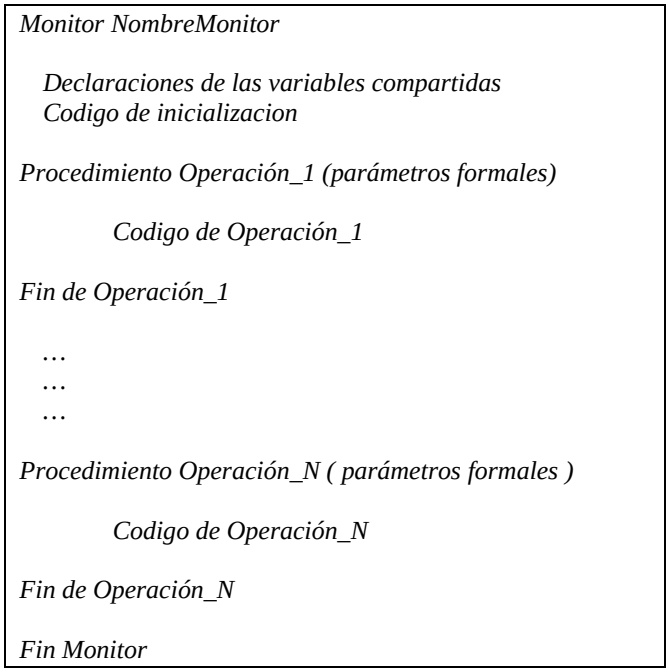

Figura 6.3: Estructura general de un Monitor

Un monitor tiene tres propiedades, las cuales hacen de él un tipo de dato abstracto. Estas son:

- Solo los nombres de los procedimientos son visibles fuera del monitor. Para alterar el estado a un proceso se debe llamar uno de los procedimientos disponibles.
- Los procedimientos en el monitor solo pueden acceder a las variables condición y a las variables locales.
- Las *variables condición* son inicializadas antes de que cualquier procedimiento sea ejecutado.

6.1.2. Sincronización en Monitores

La *exclusión mutua* se consigue asignando el recurso a un solo proceso en cada momento. Si al invocar un procedimiento del monitor se detecta que el recurso lo tiene asignado otro proceso, se bloquearía a la espera de que se libere el recurso y la condición de

sincronización es explícitamente programada usando mecanismos llamados *variables de condición*. Puesto que la ejecución en un monitor es mutuamente exclusiva, los procesos no pueden interferir entre sí cuando son accedidas las variables condición.

Una variable condición es usada para bloquear a un proceso que no puede continuar ejecutándose de forma segura, esto es hasta que el estado del monitor satisfaga alguna condición lógica. Una variable condición también es usada para liberar del bloqueo a un proceso que está suspendido. Una variable condición se define de la siguiente forma:

condicion c;

El valor de *c* es una cola de procesos bloqueados, pero este valor no es visible para el programador. La condición de suspensión es implícitamente asociada con la variable condición por el programador. Así que cada vez que un proceso deba esperar para acceder al recurso compartido, debe existir una variable condición. Es por eso que las variables condición son usadas para sincronizar procesos en los monitores; ya que ellas son declaradas y usadas solo en el monitor.

Para bloquear a un proceso sobre la variable *c*, un proceso ejecuta *cwait(c).* La ejecución de *cwait(c)* provoca dos situaciones, (1) que el proceso en ejecución se bloquee en la cola de la variable condición *c*, y (2) que la exclusión mutua sea liberada, de tal forma que otros procesos puedan entrar uno a la vez al monitor y pueda cambiar el estado de las variables condición y eventualmente puedan desbloquear a otros procesos. Además, la ejecución de *cwait(c)* causa posteriormente que el proceso tenga acceso exclusivo al monitor.

Para desbloquear a un proceso que está en la cola de la variable *condición c*, se ejecuta *csignal(c)*, esta libera al primer proceso en espera en dicha cola y en caso de que la cola no tenga procesos bloqueados, la operación no tiene efecto.

Dado que *csignal(c)* es ejecutado por un proceso dentro de la exclusión mutua, al ejecutar *csignal(c)* se violaría la exclusión mutua por lo que este proceso deberá inmediatamente abandonar el monitor, y una vez fuera del monitor podrá desbloquear a los procesos que están en espera.

6.1.3. Estructura de la Variable Condición

La estructura de la variable condición contiene información necesaria para su implementación. Físicamente, la variable condición esta representada en la tabla 6.1.

```
struct vcondicion {
        int idVC;     /* id de la variable condicion*/
        int sem;      /*semaforo para la variable condicion*/
        int count;    /*Contador de procesos para la variable condicion*/
  };
```

Tabla 6.1: Estructura de las variables condición

Esta estructura está definida en el archivo *PRIMON.DAT*, y es usada para definir variables condición en un monitor. La estructura de una *variable condición* tiene tres miembros:

- *idVC* es un identificador único para la variable condición. Podemos crear hasta *NVC* variables condición en el kernel. *NVC* esta definida en *PRIMON.DAT*.

- *sem* es un identificador único para la cola asociada a la variable condición, y es donde los procesos esperan debido a alguna condición por la que han sido bloqueados.

- *count* es el contador de procesos bloqueados en la cola asociada a la variable condición.

6.1.4. Estructura del Monitor

La estructura de un monitor contiene la información necesaria para su implementación. Físicamente, un monitor está representado en la tabla 6.2.

```
struct Monitor {
    int mutex;                  /*Semaforo para accesar el monitor*/
    int next;                   /*Semaforo para los procesos señalizadores en el monitor*/
    int next_count;             /*Contador de procesos señalizadores en el monitor*/
    struct vcondicion x[NVC];   /*Variables Condicion*/
  };
```

Tabla 6.2 Estructura de los monitores

Esta estructura está definida en el archivo *PRIMON.DAT,* y es usada para definir a un Monitor.

La estructura de un Monitor tiene cuatro miembros:

- *mutex* es el identificador del semáforo para el control de la entrada al monitor.
- *next* es el identificador del semáforo usado por los procesos señalizadores para bloquearse ellos mismos.
- *next_count* es el número de procesos señalizadores bloqueados en la cola del semáforo *next*.
- *vcondicion* es un arreglo de estructuras de variables condición utilizadas por el monitor

6.2. Primitivas

6.2.1. Primitivas del Monitor

Aquí se definen las primitivas para manejo del Monitor como son: "*CreateVarCondition*", "*CreateMonitor*", "*WaitMonitor*", "*SignalMonitor*", "*EntryMonitor*", "*ExitMonitor*", y "*BlockedsInVC*" (Figura 6.4). La estructura de las primitivas del Monitor son las siguientes:

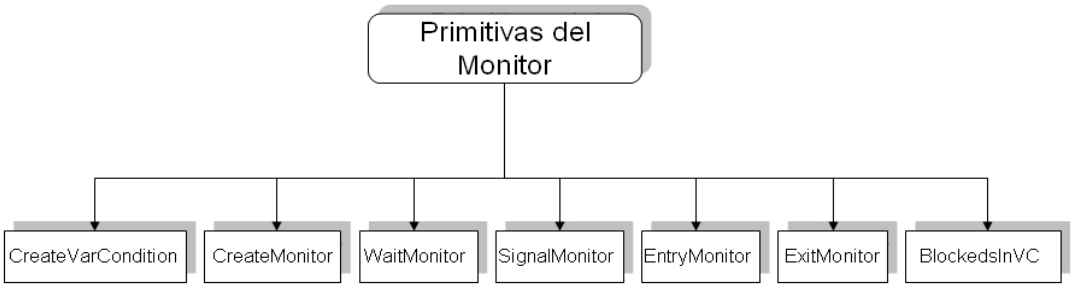

Figura 6.4: Primitivas del Monitor

CreateVarCondition:

NOMBRE: void CreateVarCondition(struct Monitor *m, int idVC, int VC)

FUNCIÓN: Se encarga de crear una variable condición para uso en el monitor.

ENTRADAS: El apuntador a la estructura del monitor (m), el índice del arreglo (IdVC) para la variable condición VC, y el identificador de la variable condición.

SALIDAS : La variable condición creada y lista para utilizarse.

CreateVarCondition: Esta primitiva se encarga de crear una variable condición en el monitor. La variable condición esta constituida por un semáforo *x.sem* y un contador denominado *x.count* que lleva la cuenta de los procesos bloqueados en la cola del semáforo *x.sem*, ambos se inicializan en 0 (ver la tabla 6.3). Además, esta primitiva se usa una sola vez cuando se crea el monitor. Si en el intento de crear la variable condición el procedimiento falla, se aborta la ejecución del kernel.

```
void CreateVarCondition(struct Monitor *m,int idVC, int VC)
{
  m->x[idVC].idVC = idVC; /* índice del elemento en el arreglo de vars cond. */
  m->x[idVC].count = 0;       /*Contador de procesos se pone en cero */
  m->x[idVC].sem = VC;       /*id del semáforo de la variable condición */
  CreaSem(m->x[idVC].sem,0,"VarCond");   /* crea semáforo y se pone en cero */
}
```

Tabla 6.3: Primitiva *CreateVarCondition*

CreateMonitor :

NOMBRE: CreateMonitor (struct Monitor *m,int MUTEX,int NEXT,int VC[])

FUNCION: Crea un Monitor determinado.

ENTRADAS: Un apuntador a la estructura del monitor, un identificador del semáforo que controla el acceso al monitor , un identificador del semáforo que controla a los procesos señalizadores y un arreglo de identificadores de variables condición para este monitor.

SALIDAS: El Monitor creado y listo para utilizarse.

CreateMonitor: Esta primitiva permite crear un monitor en el kernel. En la creación del monitor se crea un semáforo identificado como MUTEX y es inicializado en 1. Este semáforo es utilizado por un proceso antes de entrar al monitor. Además, al monitor se le crea un semáforo NEXT (ver Tabla 6.4) en el cual los procesos señalizadores pueden bloquearse asimismos; el cual es inicializado en 0. Una variable entera *next_count* es inicializada en 0 y es utilizada para contar el número de procesos bloqueados sobre NEXT. Por último se crean las variables condición utilizando el arreglo *VC*, las cuales se utilizarán para controlar el acceso al recurso compartido.

```
void CreateMonitor(struct Monitor *m,int MUTEX,int NEXT,int VC[])
{  int idVC;
    m->mutex=MUTEX;                 /* para cada monitor existe un semáforo MUTEX */
   CreaSem(m->mutex,1,"Mutex");  /* y debe ser inicializado en 1*/
   m->next=NEXT;                      /* Por cada monitor hay un semáforo NEXT para */
   CreaSem(m->next,0,"Next");       /* los procesos señalizadores pudiendo */
                                      /* suspenderse ellos mismos y se inicializa en 0*/
   m->next_count=0;                   /* Contador de procesos suspendidos en NEXT */
   for(idVC=0;idVC<NVC&&VC[idVC]>=0;idVC++) /* variables condición */
       CreateVarCondition(m,idVC,VC[idVC]);
   return;
}
```

Tabla 6.4: Primitiva *CreateMonitor*

WaitMonitor :

NOMBRE: WaitMonitor(struct Monitor *m, int VC)

FUNCION: Mete a la cola asociada a la variable condición al proceso que invoca a esta función y libera la exclusión mutua.

ENTRADAS: Apuntador a la estructura del Monitor y el identificador de la variable condición.

SALIDAS: Proceso bloqueado en la cola asociada a la variable condición y exclusión mutua libre para su uso.

WaitMonitor : Esta primitiva es invocada por el proceso en ejecución antes de entrar a la sección crítica. Entre sus tareas es incrementar el número de procesos bloqueados en la cola asociada a la variable condición (*x[idVC]->count*), además verifica que no haya procesos señalizadores bloqueados (*m->next_count>0*), y si los hay, entonces saca al primer proceso de la cola de procesos bloqueados (s*ignal(m->next)*) para que reanude su ejecución. En caso de que no haya procesos señalizadores bloqueados se señaliza al semáforo *mutex*, y así permitir que otros procesos que estaban en espera de entrar al monitor lo puedan hacer. Como podemos ver, aquí se puede cambiar el estado de las variables locales, lo cual le permitira al proceso bloqueado reanudar su ejecución posteriormente. Una vez realizado lo anterior, el proceso procede a bloquearse así mismo utilizando *Wait(m->x[idVC].sem)* y el proceso se mantendrá bloqueado hasta que otro proceso lo señalice para que pueda reanudar su ejecución y así logre decrementar en uno el número de procesos bloqueados en la cola de la variable condición (m->*x[idVC].count*) y como consecuencia el proceso en ese momento tendrá acceso a su sección crítica (ver tabla 6.5) .

```
void WaitMonitor(struct Monitor *m,int VC)
{  int idVC,loc=-1;
   for(idVC=0;idVC<NVC;idVC++)
      if( m->x[idVC].sem==VC ) {/*busca la VC en las VC del monitor */
               loc=idVC;  break;
      }
   if(loc==-1)   ERROR(19);
   m->x[idVC].count++;
   if( m->next_count > 0 ) Signal(m->next);
   else  Signal( m->mutex );
   Wait( m->x[idVC].sem );
   m->x[idVC].count--;
}
```

Tabla 6.5: Primitiva *WaitMonitor*

SignalMonitor :

NOMBRE: SignalMonitor(struct Monitor *m, int VC)

FUNCION: Envía una señal a la variable condición VC, saca a un proceso de la cola asociada a la variable condición y lo alista.

ENTRADAS: Apuntador a la estructura del Monitor y el identificador de la variable condición.

SALIDAS: Saca a un proceso de la cola de la variable condición para que reanude su ejecución.

SignalMonitor: Esta primitiva debe ser invocada por el proceso en ejecución al salir de su sección crítica. Si el número de procesos bloqueados en la cola de la variable condición es mayor que cero, entonces se incrementa en 1 el número de procesos señalizadotes. Se desbloquea a un proceso que está en espera en la cola de procesos de la variable condición para después agregarse asimismo a la cola de procesos señalizadores. Este proceso se mantiene bloqueado hasta que otro proceso lo desbloquee y cuando este reanude su ejecución decrementa en 1 el número de procesos señalizadores. Se puede apreciar que en el *caso de que no haya procesos bloqueados en la cola de la variable condición, simplemente no se realiza ninguna acción* (ver tabla 6.6).

```
void SignalMonitor(struct Monitor *m,int VC)
{
  int idVC,loc=-1;
  for(idVC=0;idVC<NVC;idVC++)
     if( m->x[idVC].sem==VC ) {/*busca la VC en las VC del monitor */
                    loc=idVC;  break;
     }
  if(loc==-1)   ERROR(20);
  if( m->x[idVC].count>0 ) {
      m->next_count++;
     Signal(m->x[idVC].sem);
     Wait(m->next);
     m->next_count--;
  }
}
```

Tabla 6.6: Primitiva *SignalMonitor*

EntryMonitor :

NOMBRE: EntryMonitor(struct Monitor m)

FUNCION: Permite a un proceso solicitar entrar al monitor

ENTRADAS: La estructura del Monitor

SALIDA: Proceso en Monitor

EntryMonitor: Esta primitiva permite a un proceso solicitar entrar al monitor. Por cada proceso que solicite entrar al monitor, debe llamar antes de entrar a esta primitiva. Lo único que hace es invocar a *Wait(m.mutex)* con el fin de verificar si el acceso al monitor está disponible y pueda entrar a él. En otro caso se bloquea hasta que otro proceso que está dentro del monitor señalice al semáforo MUTEX (ver tabla 6.7).

```
void EntryMonitor(struct Monitor m)
{
  Wait(m.mutex);
}
```

Tabla 6.7: Primitiva *EntryMonitor*

ExitMonitor :

NOMBRE: ExitMonitor(struct Monitor m)

FUNCION: Permite a un proceso abandonar el monitor

ENTRADAS: La estructura del Monitor

SALIDA: Proceso en Monitor

ExitMonitor: Esta primitiva permite a un proceso abandonar el monitor. Todo proceso que haya terminado de ejecutar su código de la sección crítica deberá llamar a esta primitiva. Como podemos apreciar en el código (ver Tabla 6.8) en caso de no haber procesos señalizadores bloqueados, se libera el acceso al monitor para que otros procesos que están en espera puedan entrar al monitor.

```
void ExitMonitor(struct Monitor m)
{
 if( m.next_count>0 ) Signal(m.next);
 else Signal(m.mutex);
 }
```

Tabla 6.8: Primitiva *ExitMonitor*

BlockedsInVC:

NOMBRE: BlockedsInVC(struct Monitor *m, int VC)

FUNCION: Obtiene el número de procesos en la cola de la variable condición VC.

ENTRADAS: Apuntador a la estructura del Monitor y el identificador de la variable condición.

SALIDA: Número de procesos bloqueados en la cola de la variable condición

BlockedsInVC: Esta primitiva permite obtener el número de procesos bloqueados en la cola asociada a la variable condición. Si el número de procesos bloqueados es cero, la primitiva regresa 0, sí la cola asociada a la variable condición no esta vacía la primitiva regresa el número de procesos que están bloqueados en su cola. Si ocurre un error debido a que el identificador de la variable condición no existe, el kernel aborta su ejecución (ver Tabla 6.9).

```
int BlockedsInVC(struct Monitor *m, int VC)
{  int idVC,loc=-1;
   for(idVC=0;idVC<NVC;idVC++)
      if(m->x[idVC].sem==VC) {
                loc=idVC; break;
      }
   if(loc==-1)    ERROR(21);
   return (m->x[loc].count);
}
```

Tabla 6.9: Primitiva *BlockedsInVC*

6.2.2 Primitivas Lectores – Escritores

Aquí se definen las primitivas para el control de concurrencia, las cuales están basadas en el problema de lectores-escritores, siendo estas: *EntrarLectorMonitor*, *SalirLectorMonitor*, *EntrarEscritorMonitor*, *SalirEscritorMonitor* e *InicializaMonitorLE* (ver figura 6.5). La estructura de las primitivas para *Lectores – Escritores* son las siguientes:

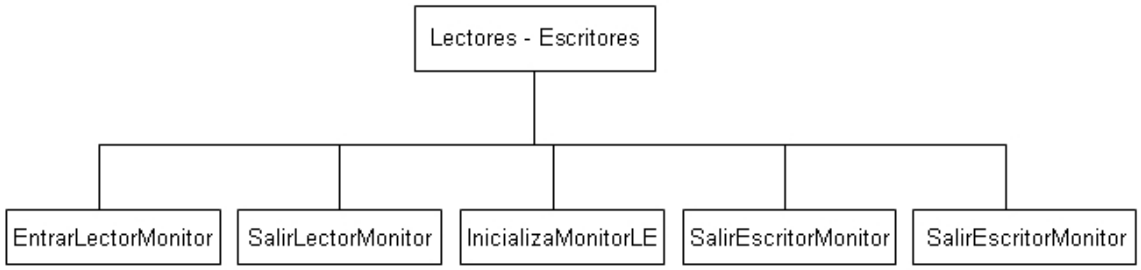

Figura 6.5: Primitivas del monitor *Lectores – Escritores*

InicializaMonitorLE:

NOMBRE: InicializaMonitorLE()

FUNCION: Se encarga de Inicializar la estructura de datos y variables del Monitor Lectores-Escritores (las variables locales, variables condición, semáforos y un apuntador a la base de datos) .

ENTRADAS: Ninguna

SALIDA: Estructura del monitor inicializada y lista para utilizarse.

InicializaMonitorLE : Esta primitiva permite al usuario inicializar y crear el monitor *lectores – escritores*. El monitor LE maneja hasta 4 variables condición (*NVC*). El monitor LE tiene a una variable global *LecEsc* que apunta a la estructura del monitor *lectores-escritores*, además cuenta con una variable local *ContLect* que le permite llevar la cuenta de los procesos lectores que están en el monitor, esta es inicializada en 0. La variable local *ProcEsc* indica según su estado, si un proceso está escribiendo en la base de datos, esta es inicializada en 0 que indica que no hay procesos en el monitor escribiendo en la base de datos. El monitor LE tiene además dos variables condición, ambas para el control de los procesos lectores y escritores, cada una de ellas está identificada por un número mayor o igual a cero (la asignación de estos identificadores de semáforos depende de cuales estén disponibles en el kernel), estas son identificadas con 2 y 3 respectivamente y el resto de las variables condición que no se usen deben ponerse en -1. Una vez que se establecieron los parámetros anteriores se procede a creación del monitor, la primitiva *CreateMonitor* es usada para ello. Los parámetros que se le pasan a esta primitiva como podemos ver son: la dirección de la estructura del monitor LE, *MUTEX* que es puesto 0, *NEXT* que es puesto en 1, y el arreglo de variables condición. Una vez creado el monitor LE, se actualiza el apuntador *BD* para que apunte a la memoria del sistema donde se almacenan las tablas que constituyen a la base de datos (ver Tabla 6.10).

```
void InicializaMonitorLE()
{ int VC[NVC];                    /* arreglo de variables condición */

  LecEsc=(struct LectorEscritor*)&Lector_Escritor;
  LecEsc->ContLect=0;             /* no hay lectores leyendo en el monitor */
  LecEsc->ProcEsc=0;              /* no hay escritor escribiendo en el monitor */
  LecEsc->LEER=VC[0]=2;           /* id del semaforo para LEER */
  LecEsc->ESCRIBIR=VC[1]=3;       /* id del semaforo para ESCRIBIR */
  VC[2]=-1;
  VC[3]=-1;
  CreateMonitor(&LecEsc->LE,0,1,VC); /* 0=MUTEX & 1=NEXT */
  LecEsc->BD=(struct MANTAB*)&MemTab;    /*BD apunta a las tablas de la Base datos*/
}
```

Tabla 6.10: Primitiva *InicializaMonitor*

EntrarLectorMonitor :

NOMBRE: void EntrarLectorMonitor()

FUNCION: Se permite a un proceso lector entrar al Monitor Lectores-Escritores.

ENTRADAS: Ninguna

SALIDA: El proceso dentro del monitor lectores-escritores, listo para realizar la lectura en la base de datos.

EntrarLectorMonitor : Esta primitiva permite a un proceso solicitar entrar al monitor *lectores-escritores* para realizar una lectura de datos. Lo primero que realiza esta primitiva es solicitar entrar al monitor LE, una vez que el proceso esta dentro del monitor, verifica si no está ocupado en alguna operación de lectura o de escritura. Si está ocupado el monitor, él proceso debe esperar en la cola de la variable condición LEER. En caso de estar disponible el monitor LE, se incrementa el número de procesos lectores (*ContLect*), y se señaliza a un proceso que esta en espera en la cola de la variable condición LEER para que este pueda continuar su ejecución y realice la lectura en la base de datos (ver Tabla 6.11).

```
void EntrarLectorMonitor()
{
  EntryMonitor(LecEsc->LE); /* entrar al monitor de lectores-escritores*/
  if(LecEsc->ProcEsc||BlockedsInVC(&LecEsc->LE,LecEsc->ESCRIBIR))
     /* si un escritor esta escribiendo o hay escritores esperando */
      WaitMonitor(&LecEsc->LE,LecEsc->LEER); /*se hace esperar al lector*/
  LecEsc->ContLect++; /*entra a leer, incrementa el num de procs. lecs.*/
  SignalMonitor(&LecEsc->LE,LecEsc->LEER);/*señaliza a un lector y pueda*/
  ExitMonitor(LecEsc->LE); /* entrar a leer al monitor de lectores-escritores*/
  /* LEER */
}
```

Tabla 6.11: Primitiva *EntrarLectorMonitor*

SalirLectorMonitor :

NOMBRE: void SalirLectorMonitor()

FUNCION: Se permite a un proceso lector abandonar el Monitor Lectores-Escritores.

ENTRADAS: Ninguna

SALIDA: El monitor de lectores-escritores disponible para que otro proceso pueda realizar alguna operación de lectura o escritura en la base de datos.

SalirLectorMonitor : Esta primitiva permite a un proceso que a terminado de realizar una lectura en la base de datos, salir del monitor *lectores-escritores*. La primitiva al iniciar su ejecución solicita entrar al monitor LE. Una vez que el proceso está dentro del monitor, decrementa el contador de procesos lectores (*ContLect*), y verifica si no hay procesos lectores en espera de leer la base de datos; si este fuera el caso se señaliza a la variable condición *ESCRIBIR* para sacar a un proceso de su cola de espera para que este pueda reanudar su ejecución. Finalmente el proceso abandona el monitor LE dejándolo en estado disponible para que otro proceso pueda hacer uso de él (ver Tabla 6.12).

```
void SalirLectorMonitor()
{
  EntryMonitor(LecEsc->LE); /* entrar al monitor de lectores-escritores*/
  LecEsc->ContLect--;        /* decrementa el numero de procesos lectores */
  if(LecEsc->ContLect==0)  /* si no hay proceso en espera de leer */
     SignalMonitor(&LecEsc->LE,LecEsc->ESCRIBIR);/*señaliza a un escritor*/
  ExitMonitor(LecEsc->LE);   /* salir del monitor lectores-escritores */
  return;
}
```

Tabla 6.12: Primitiva *SalirLectorMonitor*

En la figura 6.6 podemos observar a tres lectores activos en el monitor accesando la base de datos. También tenemos un lector que puede entrar al monitor y una tarea escritora que tiene que esperar hasta que en el monitor no haya ningún lector activo.

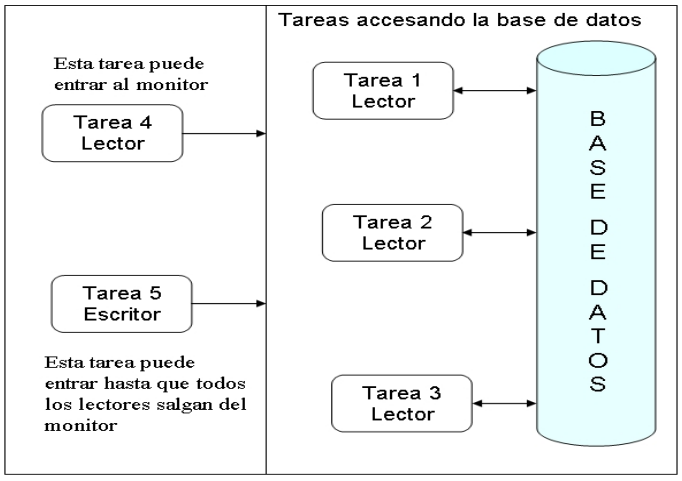

Figura 6.6: Monitor lectores-escritores con tres lectores activos.

EntrarEscritorMonitor :

NOMBRE: void EntrarEscritorMonitor()

FUNCION: Se permite a un proceso escritor entrar al Monitor Lectores-Escritores.

ENTRADAS: Ninguna

SALIDA: El proceso dentro del monitor lectores-escritores, listo para realizar la escritura en la base de datos.

EntrarEscritorMonitor: Esta primitiva permite a un proceso solicitar entrar al monitor *lectores-escritores* para realizar una escritura en la base de datos. Lo primero que realiza esta primitiva es solicitar entrar al monitor LE. Una vez que el proceso está dentro del monitor, verifica si hay procesos lectores o escritores activos. Si existe algún proceso activo, el proceso escritor debe esperar hasta que esté disponible el monitor LE. Una vez que el proceso puede entrar al monitor, pone en 1 la variable *ProcEsc* que indica que el monitor está ocupado por el proceso en una escritura en la base de datos. Después de esto, el proceso sale de esta primitiva dejando en estado disponible al monitor LE y el proceso solicitante en ese momento puede realizar la escritura o actualización de datos en la base de datos (ver Tabla 6.13).

```
void EntrarEscritorMonitor()
{
  EntryMonitor(LecEsc->LE);   /* entrar al monitor lectores-escritores */
  if(LecEsc->ContLect>0||LecEsc->ProcEsc)/*lector en espera o escritor activo*/
     WaitMonitor(&LecEsc->LE,LecEsc->ESCRIBIR);/*espera a que lo se¤alicen*/
  LecEsc->ProcEsc=1; /*entra a escribir y edo. proc. esc. se pone en uno */
  ExitMonitor(LecEsc->LE);   /* salir del monitor lectores-escritores */
  /* ESCRIBIR */
}
```

Tabla 6.13: Primitiva EntrarEscritorMonitor

SalirEscritorMonitor :

NOMBRE: void SalirEscritorMonitor()

FUNCION: Se permite a un proceso escritor abandonar el Monitor Lectores-Escritores.

ENTRADAS: Ninguna

SALIDA: El monitor de lectores-escritores disponible para que otro proceso pueda realizar alguna operación de lectura o escritura en la base de datos.

SalirEscritorMonitor : Esta primitiva permite a un proceso que ha terminado de realizar una escritura o actualización en la base de datos salir del monitor *lectores-escritores*. La primitiva al iniciar su ejecución solicita entrar al monitor LE. Una vez que el proceso está dentro del monitor, pone la variable *ProcEsc* en 0. Esto indica que el monitor está disponible para que otro proceso pueda entrar al monitor LE. Una vez realizado lo anterior, se verifica si hay procesos lectores en espera de leer la base de datos. Si este fuera el caso, se señaliza a la variable condición *LEER* para sacar a un proceso de su cola de espera para que pueda reanudar su ejecución. Pero si no hay procesos lectores esperando, y hubiera procesos escritores en espera se señaliza la variable condición *ESCRIBIR* para sacar a un proceso de su cola de espera y para que reanude su ejecución, una vez realizado lo anterior el proceso abandona el monitor LE dejándolo en estado disponible para que otro proceso pueda hacer uso de él (ver Tabla 6.14).

```
void SalirEscritorMonitor()
{
  EntryMonitor(LecEsc->LE);   /* salir del monitor lectores-escritores */
  LecEsc->ProcEsc=0;/*el proceso actual indica que  termino la escritura */
  if(BlockedsInVC(&LecEsc->LE,LecEsc->LEER)) /* cola con procesos lectores? */
     SignalMonitor(&LecEsc->LE,LecEsc->LEER); /* si, se¤aliza un lector */
  else                                  /* si no, en caso de haber */
     SignalMonitor(&LecEsc->LE,LecEsc->ESCRIBIR); /* escritores saca uno */
  ExitMonitor(LecEsc->LE);  /* salir del monitor lectores-escritores */
  return;
}
```

Tabla 6.14: Primitiva SalirEscritorMonitor

En la figura 6.7 podemos observar a un escritor activo en el monitor accesando la base de datos, y tenemos a una tarea lectora y a una escritora esperando entrar al monitor una vez que la tarea activa termine de realizar la escritura y abandone el monitor.

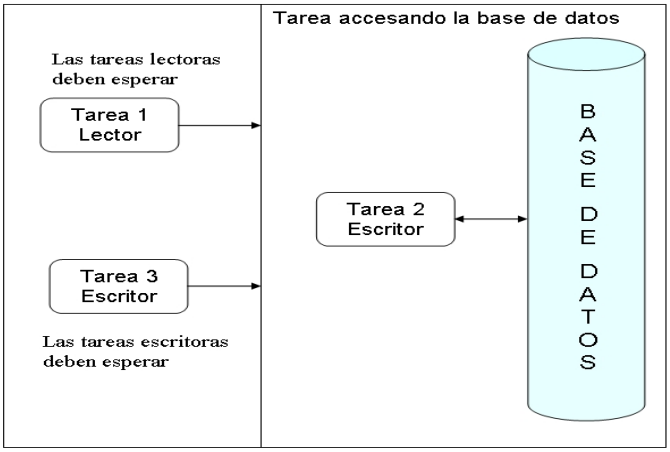

Figura 6.7: Monitor lectores-escritores con un escritor activo.

6.2.3. Primitivas del Manejador de Base de Datos

Aquí se definen las primitivas para el Manejador de Base de Datos como son: "*Seleccion*", "*Proyeccion*", "*Join*", "*Insertar*", "*Delete*", y "*Update*" (Figura 6.8). La estructura de las primitivas del Manejador de Base de Datos son las siguientes:

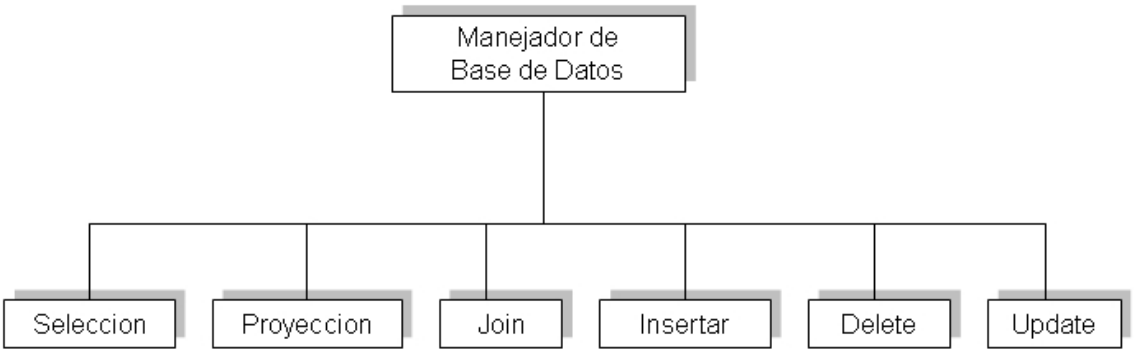

Figura 6.8: Primitivas del Manejador de Base de Datos

Las primitivas del Manejador de Base de Datos se auxilian de las primitivas del monitor lectores-escritores para mantener integridad y consistencia de los datos en la base de datos, así que cuando se realiza alguna operación de lectura en la base de datos, esta debe ir entre estas dos primitivas: *EntrarLectorMonitor()* al inicio y al final *SalirLectorMonitor()*. Las

operaciones de escritura deben ir entre estas dos primitivas, *EntrarEscritorMonitor()* primero, y al último *SalirEscritorMonitor()*.

Seleccion :

NOMBRE: unsigned long Seleccion(char *tabla,char *cond,int *mid)

FUNCION: Realiza la operación Selección sobre una tabla de la base de datos.

ENTRADAS: El nombre de la tabla origen de la Base de Datos, la condición y un apuntador a un entero para almacenar el identificador de la tabla resultado.

SALIDA: Número de registros recuperados, una tabla de registros recuperados y el identificador de la tabla resultado. En caso de fallo se aborta el kernel.

Selección : Esta primitiva (ver tabla 6.15) permite seleccionar un conjunto de registros sin repetición de una tabla de la base de datos que cumplen una condición especifica.

```
unsigned long Seleccion(char *tabla,char *cond,int *mid)
{
 struct MANTAB mt;
 struct HEADT t;
 Uchar *Registro;
 int id,poscampo, regok,idxc;
 long n,ldat;
 unsigned int lenreg,ncampos;
 unsigned long nregs,contregs;
 char tipoval,opercond[3],campocond[MAXLCAMPO];
 Uchar valcond[MAXLDAT];
 double fval;

 EntrarLectorMonitor();
   id=GetIdTable(tabla); /*Se accesa la BD para obtener el id de la tabla */
   if(id<0)
     ERROR(100); /* error la tabla no existe */
   mt.cnt=LecEsc->BD[id].cnt;          /* se lee el tama¤o ocupado de la tabla */
   mt.ptrtab=LecEsc->BD[id].ptrtab;    /* se obtiene la direccion de los datos */
   mt.pribloq=LecEsc->BD[id].pribloq;  /* se obtiene el indice del ier bloque*/
 SalirLectorMonitor();
 n=ObtenCamposCond(cond,campocond); /* se obtiene el nombre del campo (izq)*/
 ObtenOperVal(cond+n,opercond,valcond,&tipoval); /*se obtiene el operador y el
                                                          valor del campo (der)*/
 mt.idx=LNOMTAB;  /* salta el nombre de la tabla */
```

Tabla 6.15: Primitiva *Selección*

```
EntrarLectorMonitor();
  /* calcula la posicion de los datos del campo en el registro */
    poscampo=GetPosField(&t,campocond,&idxc,id,&ncampos,&nregs,&lenreg);
 SalirLectorMonitor();
/* Se verifican los tipos de datos del valor del campo especificado
   con el tipo del dato del campo que esta en la tabla */
 if(tipoval==0&&(t.tipo=='I'||t.tipo=='L'||t.tipo=='F')) {
     switch( t.tipo ) {
            case 'I':
            case 'L': ldat= AtoN(valcond);
                        break;
            case 'F': fval= Atof(valcond);
                        break;
     }
 }
 else fval=ldat=0;
 if(poscampo>=0) {
    contregs=0L;
    *mid=CreaTablaBD(tabla,NULL,"selec");/*crea una tabla con todos los campos*/
    if(*mid<0)
       ERROR(102);
    n=0L;
    /* Indice del manejador de la tabla de la BD se pone al inicio de datos */
    mt.idx= LNOMTAB+LNC+LNREGS+LLREG+(ncampos*(MAXLCAMPO+4));
    while(n<nregs) {
              EntrarLectorMonitor();
                Registro=reg_select;
               /*lee un registro de la tabla de la BD con identificador id*/
                if(mtRead(Registro,lenreg+1,&mt)<(lenreg+1)) {
                        if(LecEsc->BD[id].cnt!=mt.cnt)
                          ERROR(101); /* error tabla corrupta */
                        break;
                }
              SalirLectorMonitor();
              regok=ValidaDatos(Registro,opercond,poscampo,t.longitud,
                                        t.tipo,valcond,ldat,fval);
              if(regok) {
                /* escribe registro encontrado en la tabla de la BD */
                EntrarEscritorMonitor();
                        if(InsertaRegSinRepe(*mid,Registro)>=2)   /*ordena*/
                          contregs+=1L;
                SalirEscritorMonitor();
              }
              n+=1L;
     }
}
 return contregs;
}
```

Tabla 6.15: Primitiva *Selección* (continuación)

Proyeccion :

NOMBRE: unsigned long Proyeccion(char *tabla,char const *cols, int *mid)

FUNCION: Realiza la operación Proyeccion sobre una tabla de la base de datos.

ENTRADAS: El nombre de la tabla origen de la Base de Datos, la condición y un apuntador a un entero para almacenar el identificador de la tabla resultado.

SALIDA: Número de registros recuperados, una tabla en la base de datos donde están almacenados los registros recuperados y el identificador de la tabla resultado. En caso de fallo se aborta el kernel.

Proyeccion : Esta primitiva (ver Tabla 6.16) permite seleccionar un conjunto de registros de una tabla de la base de datos. Los registros contienen únicamente los valores sin repetición de las columnas especificadas.

```
unsigned long Proyeccion(char *tabla, const char *cols,int*mid)
{
 struct MANTAB mt;
 struct HEADT c;
 Uchar ubuf[MAXLDAT];
 long int n;
 unsigned long nrecups,nregs;
 unsigned int lenreg,bcampos,ncampos,lreg;
 int flag,lec,id,i,k,idx,pos;
 unsigned char *Registro;
 unsigned char *RegProyect;

 EntrarLectorMonitor();
   id=GetIdTable(tabla); /*Se accesa la BD para obtener el id de la tabla */
   if(id<0)
     ERROR(200); /* error la tabla no existe */
   mt.cnt=LecEsc->BD[id].cnt;          /* se lee el tama¤o ocupado de la tabla */
   mt.ptrtab=LecEsc->BD[id].ptrtab;    /* se obtiene la direccion de los datos */
   mt.pribloq=LecEsc->BD[id].pribloq;
 SalirLectorMonitor();
 bcampos=ObtenCampos((char*)&campos[0], cols); /* campos a recuperar */

 mt.idx=LNOMTAB;
 EntrarLectorMonitor();
   mtRead((Uchar*)&ncampos,2,&mt);  /* numero de campos      */
   mtRead((Uchar*)&nregs,4,&mt);    /* numero de registros   */
   mtRead((Uchar*)&lenreg,2,&mt);   /* longitud del registro */
 SalirLectorMonitor();

 if(bcampos>0) {
   *mid = CreaTablaBD(tabla,(char*)&campos[0],"proyecc");
   if(*mid<0)
     ERROR(202);
   mt.idx=LNOMTAB+2+4+2+(ncampos*14); /*apunta a los datos*/
   n=nrecups=0;
   while(n<nregs) {
        EntrarLectorMonitor();
          Memset(xbuf,0,MAXLREG);
          Registro=xbuf;
          lec=mtRead(Registro,lenreg+1,&mt);
```

Tabla 6.16: Primitiva *Proyeccion*

```
        if(lec<(lenreg+1)||LecEsc->BD[id].cnt!=mt.cnt)
                ERROR(201);  /* error tabla corrupta */
        SalirLectorMonitor();
        idx=mt.idx;
        RegProyect=reg_proyect;
        RegProyect[0]=' ';
        RegProyect[1]=0;
        /* Obtiene solo los campos especificados en la proyeccion*/
        for(k=0;k<bcampos;k++) {
           pos=1;
           flag=0;
           mt.idx=LNOMTAB+2+4+2;
           for(i=0;i<ncampos&&flag<bcampos;i++){
                   EntrarLectorMonitor();
                     lec=mtRead((unsigned char *)&c,sizeof(struct HEADT),&mt);
                     if(lec<sizeof(struct HEADT)||LecEsc->BD[id].cnt!=mt.cnt)
                        ERROR(201);
                   SalirLectorMonitor();
                   if(Strncmp(campos[k],c.campo,Longitud(campos[k]))==0){
                     Memcpy(ubuf,Registro+pos,c.longitud);
                     ubuf[c.longitud]=0;
                     Strcat(RegProyect,ubuf);
                     flag++;
                     break;
                   }
                   /*actualiza la posicion de los datos*/
                   pos+=c.longitud;
           }
        }
        if(flag) {
          /* Inserta en la tabla con eliminacion de registros repetidos */
          EntrarEscritorMonitor();
           if(InsertaRegSinRepe(*mid,RegProyect)>=2)
                   nrecups+=1L;
          SalirEscritorMonitor();
        }
        mt.idx=idx;
        n+=1L;
   }
 }
 return nrecups;
}
```

Tabla 6.16: Primitiva *Proyeccion* (continuación)

Join :

NOMBRE: unsigned long Join(char *T1,char *T2, char *cond, int *mid)

FUNCION: Realiza la operación Join Natural de dos tablas de la base de datos.

ENTRADAS: Los nombres de dos tablas de la Base de Datos, una condición y un apuntador a un entero para almacenar el identificador de la tabla resultado.

SALIDA: Número de registros recuperados, una tabla nueva en la base de datos donde están almacenados los registros recuperados y su identificador. En caso de fallo se aborta el kernel.

Join : Esta primitiva (ver Tabla 6.17) permite obtener los registros sin repetición de la reunión de dos tablas que cumplan con la condición de igualdad de dos columnas dadas.

```
unsigned long Join(char *T1,char *T2,char *cond,int *mid)
{struct MANTAB mt,mu;
 struct HEADT t1,t2,t2a;
 int id_t1,id_t2;
 Uchar *Reg_t1;
 Uchar *Reg_t2;
 int i,k,regok,Pos,pos;
 long int n,nn;
 unsigned long nregs;
 unsigned int lenr_t1,lenr_t2,nc_t1,nc_t2,nc_t3;
 unsigned long nregs_t1, nregs_t2,nrecups;
 char Atr_T1[MAXLCAMPO+1],Atr_T2[MAXLCAMPO+1];
 char opercond[3];
 char tdatr_t1;
 char campos[2];
 int idx,poscampo_t1,poscampo_t2;
 int ic_t1,ic_t2;

 EntrarLectorMonitor();
 id_t1=GetIdTable(T1);
   if(id_t1<0)   ERROR(600); /* error T1 no existe */
   mt.idx = 0;
   mt.pribloq=LecEsc->BD[id_t1].pribloq;
   mt.cnt = LecEsc->BD[id_t1].cnt;
   mt.idx = LNOMTAB;
   mt.ptrtab = LecEsc->BD[id_t1].ptrtab;
   id_t2=GetIdTable(T2);
   if(id_t2<0)   ERROR(601); /* error T2 no existe */
   mu.idx = 0;
   mu.pribloq=LecEsc->BD[id_t2].pribloq;
   mu.cnt = LecEsc->BD[id_t2].cnt;
   mu.idx = LNOMTAB;
   mu.ptrtab = LecEsc->BD[id_t2].ptrtab;
 SalirLectorMonitor();
 i=ObtenCamposCond(cond,Atr_T1);    /* parte izq de la condicion   */
 opercond[0]=cond[i++];           /* operador                    */
 i=ObtenCamposCond(cond+i,Atr_T2);  /* parte der de la condicion   */
 EntrarEscritorMonitor();
   /* Obtiene la posicion de los datos del campo en el registro */
   poscampo_t1=GetPosField(&t1,Atr_T1,&ic_t1,id_t1,&nc_t1,&nregs_t1,&lenr_t1);
   /* Obtiene la posicion de los datos del campo en el registro */
   poscampo_t2=GetPosField(&t2,Atr_T2,&ic_t2,id_t2,&nc_t2,&nregs_t2,&lenr_t2);
 SalirEscritorMonitor();
   campos[0]=0;
   *mid=CreaTablaBD(T1,(char*)&campos[0],"join");
   if(*mid<0)  ERROR(605);  /*la tabla solicitada no se pudo crear, falta memoria*/
   nc_t3=JuntaCamposTablasBD("join",T2,Atr_T2);
```

Tabla 6.17: Primitiva *Join*

```
EntrarEscritorMonitor();
  if(poscampo_t1>=0) {
    n=nrecups=0L;
    mt.idx=LNOMTAB+LNC+LNREGS+LLREG +((MAXLCAMPO+4)*nc_t1);
    while(n<nregs_t1) {
      Reg_t1=xbuf;
      if(mtRead(Reg_t1,lenr_t1+1,&mt)<lenr_t1)  ERROR(602);  /*lee registro de T1 */
      nn=0L;
       mu.idx=LNOMTAB+LNC+LNREGS+LLREG+(nc_t2*(MAXLCAMPO+4));
      while(nn<nregs_t2) {/*todos los registros de la tabla T2*/
            Reg_t2=kbuf;
            if(mtRead(Reg_t2,lenr_t2+1,&mu)<lenr_t2)   ERROR(602); /*lee reg de T2*/
            regok=Strncmp(Reg_t1+poscampo_t1+1,Reg_t2+poscampo_t2+1,
                          t1.longitud);/*compara Reg T1 y Reg T2*/
              if(regok==0) { /*son =? obtiene los valores de los campos*/
                Pos=0;
                Memcpy(_buf_,Reg_t1,lenr_t1+1);/*1a. parte del reg de T1*/
                Pos+=lenr_t1+1;  /*actualiza la longitud del registro */
                pos=1;
                idx=mu.idx;
                mu.idx=12+8;
                for(i=0;i<nc_t2;i++) {
                   if(mtRead((Uchar*)&t2a,sizeof(t2a),&mu)<sizeof(t2a)) ERROR(602);
                    k=Strncmp(Atr_T2,t2a.campo,MAXLCAMPO);
                    if(k) {/*agrega los valores d campos excepto Atr_T2*/
                       Memcpy(_buf_+Pos,Reg_t2+pos,t2a.longitud);
                       Pos+=t2a.longitud;/*act long del registro*/
                    }
                    pos+=t2a.longitud;/*actualiza la posicion de los datos*/
                }
                 mu.idx=idx;
                /* inserta registro a la tabla sin repeticion */
                if(InsertaRegSinRepe(*mid,_buf_)>=2)  nrecups+=1L;
                  }
              nn+=1L;
      }
      n+=1L;
    }
  }
SalirEscritorMonitor();
return nrecups;    /*retorna el numero de registros recuperados*/
}
```

Tabla 6.17: Primitiva *Join* (continuación)

Insertar :

NOMBRE: int Insertar(char *tabla, Uchar *valores)

FUNCION: Realiza la inserción de un registro en la base de datos.

ENTRADAS: El nombre de la tabla de la base de datos donde se insertará el registro y los valores de las columnas separados por tabulador.

SALIDA: Registro insertado en la tabla de datos especificada, en caso de fallar la inserción se aborta el kernel.

Insertar : Esta primitiva (ver Tabla 6.18) permite insertar un registro en una tabla de la base de datos.

```
int Insertar(char *tabla,Uchar *valores)
{int id;
 struct HEADT t;
 Uchar valor[MAXLDAT],sep[2];
 Uchar *Registro;
 unsigned int lenreg;
 unsigned int ncampos;
 unsigned long int nregs;
 int i,l,k;
 int indexc;
 EntrarLectorMonitor();
   id=GetIdTable(tabla);
   if(id<0)   ERROR(300);                    /* error tabla no existe */
   LecEsc->BD[id].idx=LNOMTAB;
   mRead((Uchar*)&ncampos,LNC,id);   /* numero de campos      */
   mRead((Uchar*)&nregs,LNREGS,id);  /* numero de registros   */
   mRead((Uchar*)&lenreg,LLREG,id);  /* longitud del registro */
SalirEscritorMonitor();
 l=Longitud(valores);       /* longitud de la cadena de los valores */
 EntrarEscritorMonitor();
   Registro=_buf_;
   Registro[0]=' ';
   Registro[1]=0;
         indexc=i=0;
         do { i=GetToken(valores,i,valor,MAXLDAT);  /* valor del campo */
          /* Lee los atributos del campo, especificando su indice */
          k=GetDatIndexField(&t,indexc,id);
          if(k>=0) {
            i=GetToken(valores,i,sep,1); /* \t es el separador de valor */
            if((*sep=='\t'||*sep==0)&&i<=l) {
              if(t.tipo=='I'||t.tipo=='L'||t.tipo=='F')
               AjustaValorIzq(valor,t.longitud,'0');
              else  AjustaValorDer(valor,t.longitud,' ');
              Strcat(Registro,valor); /* se agrega el valor al registro */
            }  else
                 if(*sep!=0)  ERROR(304); /* error no se encontro fin de datos */
          } else  ERROR(305); /* error: el campo no existe */
          indexc++; /* siguiente campo */
       }  while(i<l&&indexc<ncampos&&*sep=='\t');
      if(*sep!=0 )    ERROR(304); /* error en el formato de los datos    */
```

Tabla 6.18: Primitiva *Insertar*

```
        if(Longitud(Registro)==lenreg+1) {  /* longitud del reg. es exacta */
          if(mWrite(Registro,lenreg+1,id)!=-1) {/* se inserta el nuevo registro */
             nregs+=1L;                /* se actualiza el numero de registros */
             mRWrite((Uchar*)&nregs,4,id);
          } else   ERROR(307);  /* error, el registro no se puedo agregar a la tabla */
  } else ERROR(306); /* el registro no se inserta, excede el tamaño permitido */
SalirEscritorMonitor();
return 1;
}
```

Tabla 6.18: Primitiva *Insertar* (continuación)

Delete :

NOMBRE: int Delete(char *tabla, char *cond)

FUNCION: Realiza la eliminación de registros de una tabla de la base de datos que cumplan con una condición especifica.

ENTRADAS: El nombre de la tabla de la base de datos en la cual se eliminarán los registros y la condición.

SALIDA: Registros eliminados de la tabla de la base de datos, en caso de fallar la eliminación se aborta el kernel.

Delete : Esta primitiva (ver Tabla 6.19) permite eliminar registros de una tabla de la base de datos que cumplan con una condición especifica. La condición tiene la forma: <*columna*> <*operador*> <*valor*>, donde *columna* es el nombre de la columna de la tabla, *operador* es cualquiera de estos operadores: =,>,<, y *valor* que debe de ser del mismo tipo de dato que la columna comparada.

```
int Delete(char *tabla, char *cond)
{int id;
 struct HEADT t;
 Uchar *ptrtab;
 int i,idxc;
```

Tabla 6.19: Primitiva *Delete*

```
 long n;
 unsigned int idx,posreg, lenreg,ncampos;
 unsigned long nregs,contregs=0L;
char campocond[MAXLCAMPO+1];
 char opercond[3];
 Uchar valcond[MAXLDAT];
 char tipoval;
 int poscampo, regok;
 long int ldat;
 double fval;
 Uchar *Registro;

 EntrarLectorMonitor();
   id=GetIdTable(tabla);
if(id<0) {
SalirLectorMonitor();
       ERROR(401); /* error tabla no existe */
   }
 SalirLectorMonitor();
 i=ObtenCamposCond(cond, campocond); /* parte izq de la condicion */
 ObtenOperVal(cond+i, opercond,valcond, &tipoval); /* parte der de la condicion */
 EntrarLectorMonitor();
 /* calcula la posicion de los datos del campo en el registro */
  poscampo=GetPosField(&t,campocond,&idxc,id,&ncampos,&nregs,&lenreg);
 SalirLectorMonitor();
 if( poscampo == -1 )    ERROR(400); /* no se encontro el campo */
 if( tipoval==0 && (t.tipo=='I' || t.tipo=='L' || t.tipo=='F') ) {
    switch( t.tipo ) {
       case 'I':
       case 'L': ldat= AtoN(valcond); break;
       case 'F': fval= Atof(valcond);   break;
       default:  fval=ldat=0;  break;
    }
 }
 n=contregs=0; /* contador de registros borrados */
 EntrarEscritorMonitor();
   Registro=_buf_;
   LecEsc->BD[id].idx=LNOMTAB+8+(ncampos*(MAXLCAMPO+4));
   while(n<nregs) {
        posreg=LecEsc->BD[id].idx;
        if(mRead(Registro,lenreg+1,id)<(lenreg+1))   break;
        regok=ValidaDatos(Registro,opercond,poscampo,t.longitud,t.tipo,
                                valcond,ldat,fval);
        if(regok) {
          idx=LecEsc->BD[id].idx;
          LecEsc->BD[id].idx=posreg;
          Registro[0]=0; /*se marca el registro para borrar*/
          mRWrite(Registro,1,id);
          LecEsc->BD[id].idx=idx;
```

Tabla 6.19: Primitiva *Delete*

```
            contregs+=1L;
          }
          n+=1L;
  }
    if(contregs) {   /* se eliminan todos los registros marcados */
      if(EliminaReg(id,nregs,lenreg+1,ncampos)!=contregs)  ERROR(401);
       else { /* actualiza el numero de registros en la tabla */
            nregs-=contregs;
            LecEsc->BD[id].idx=LNOMTAB+LNC;
            mRWrite((Uchar*)&nregs,LNREGS,id);
            LecEsc->BD[id].cnt-=contregs*(lenreg+1);
       }
    }
     SalirEscritorMonitor();
  return contregs;
}
```

Tabla 6.19: Primitiva *Delete* (continuación)

Update :

NOMBRE: int Update(char *tabla, char *cond)

FUNCION: Realiza la actualización de registros de una tabla de la base de datos que cumplan con una condición especifica.

ENTRADAS: El nombre de la tabla de la base de datos donde se actualizarán los registros, las columnas con los nuevos valores y la condición.

SALIDA: Registros actualizados en la tabla de la base de datos.

Update : Esta primitiva (ver Tabla 6.20) permite actualizar registros de una tabla de la base de datos con nuevos valores que cumplan con una condición especifica. La forma de especificar la actualización (*set*) es: <*columna*> <*signo igual*> <*nuevo valor*>, esta especificación puede repetirse hasta *NUM_ACT* veces, para esta configuración del kernel se tiene un valor de 4, cada una de estas especificaciones deben ir separadas por coma. La condición (*where*) tiene la forma: <*columna*> <*operador*> <*valor*>, donde *columna* es el nombre de la columna de la tabla, *operador* es cualquiera de estos operadores: =,>,<, y *valor* el cual debe de ser del mismo tipo de dato que la columna comparada.

```
int Update(char *tabla,char *set,char *where)
{
 int id;
 struct HEADT t;
 int i,idxc;
 long n;
 unsigned int lenreg,ncampos;
 unsigned long nregs,contregs=0L;
 Uchar campocond[MAXLCAMPO+1];
 char opercond[3];
 Uchar valcond[MAXLDAT];
 char tipoval;
 int poscampo, posc,regok;
 long int ldat;
 double fval;
 Uchar *Registro;

 EntrarLectorMonitor();
   id=GetIdTable(tabla);
 SalirLectorMonitor();
 if( id<0 )   ERROR(500);
 i=ObtenCamposCond(where, campocond); /* parte izq de la condicion */
 ObtenOperVal(where+i, opercond,valcond, &tipoval); /* parte der de la condicion */
 /* calcula la posicion de los datos del campo en el registro */
   poscampo=GetPosField(&t,campocond,&idxc,id,&ncampos,&nregs,&lenreg);
 if(tipoval==0&&(t.tipo=='I'||t.tipo=='L'||t.tipo=='F')) {
    switch( t.tipo ) {
       case 'I':
       case 'L': ldat=AtoN(valcond);
                 break;
       case 'F': fval=Atof(valcond);
                 break;
       default:  fval=ldat=0;
                 break;
    }
 }
 EntrarEscritorMonitor();
   GetFieldsValsUpdate(tabla,update,set); /*obtiene campos y valores a act. */
     n=0L;
        if(poscampo>=0) {
         LecEsc->BD[id].idx=LNOMTAB+8+(ncampos*(MAXLCAMPO+4));
         contregs=0; /* contador de registros actualizados */
         while(n<nregs) {
           Registro=_buf_;
           posc=LecEsc->BD[id].idx;
           if(mRead(Registro,lenreg+1,id)<(lenreg+1) ) break;    /*lee registro*/
           regok = ValidaDatos(Registro,opercond,poscampo,t.longitud,t.tipo,
                               valcond,ldat,fval); /* compara valores */
```

Tabla 6.20: Primitiva *Update*

```
            if( regok ) {
                    ActualizaCampos(Registro,update); /*actualiza datos en reg.*/
                    LecEsc->BD[id].idx=posc;
                    mRWrite(Registro,lenreg+1,id);/*sobreescribe el registro*/
                    contregs+=1L;
            }
          n+=1L;
       }
        LecEsc->BD[id].idx=0;
    }
 SalirEscritorMonitor();
 return contregs;
}
```

Tabla 6.20: Primitiva *Update* (continuación)

6.2.4. Descripción de la Estructura de las Tablas

En esta sección se describen cada uno de los elementos que constituyen a una tabla de la base de datos, así como su organización en la memoria del sistema. La información de entrada a la tabla. Esta se muestra en orden secuencial y con el número de bytes que ocupa para almacenarse (ver Tabla 6.21).

Posición	**Tamaño en bytes**	**Descripción**
1	*12*	*El nombre de la tabla*
2	*2*	*Número de columnas*
3	*4*	*Número de Registros*
4	*2*	*Longitud del registro*

Tabla 6.21: Entrada a la Tabla de la Base de Datos

La siguiente parte se refiere a la descripción de las columnas. Esta tiene la organización en la memoria del sistema tal como se muestra en la tabla 6.22. Puede notar que esta información es para cada una de las columnas que forman una tabla de la base de datos.

Posición	Tamaño en bytes	Descripción
1	*10*	*Nombre de la columna*
2	*1*	*Tipo de datos*
2	*2*	*Longitud del dato*
3	*1*	*Decimales*

Tabla 6.22: Organización de los campos en una tabla de la Base de Datos

Después de la descripción de las columnas, enseguida viene un campo que sirve para indicar el estado del registro, en este caso se usa principalmente para marcar registros como listos para ser eliminados de la tabla, después de este campo, vienen los datos del registro.

Los tipos de datos que se pueden asignar a las columnas de una tabla son:

- *Cadena (S)*
- *Entero (I)*
- *Booleano (B)*
- *Fecha (D)*
- *Hora (T)*
- *Real (F)*

6.2.5. Organización de la memoria del MBDTR

El manejador de base de datos de Tiempo Real utiliza un área de memoria de tamaño fijo para la base de datos, y su manipulación es dinámica. Esto se organiza dividiendo esta área de memoria en bloques de un tamaño específico. En esta implementación el tamaño del bloque es de 512 bytes y hay un total de 20 bloques. Estos están representados por un arreglo denominado *BloquesBD*. Cada bloque está constituido por: *el identificador de la tabla a la que pertenece el bloque, un número secuencial de bloque, número o índice del bloque y los datos* (ver figura 6.9 inciso b). Para hacer uso de estos bloques que forman las tablas, se creo una estructura denominada *MEMTAB* la cual permite hacer el manejo de la tabla. Este manejador está constituido por: *número o índice del primer bloque, un índice*

para uso local del contenido de la tabla, un contador de bytes ocupados y un apuntador que se usa para accesar a los datos del bloque (ver figura 6.9 inciso a). En la figura 6.9 inciso c, se muestran dos ejemplos de cómo esta organizada la memoria.

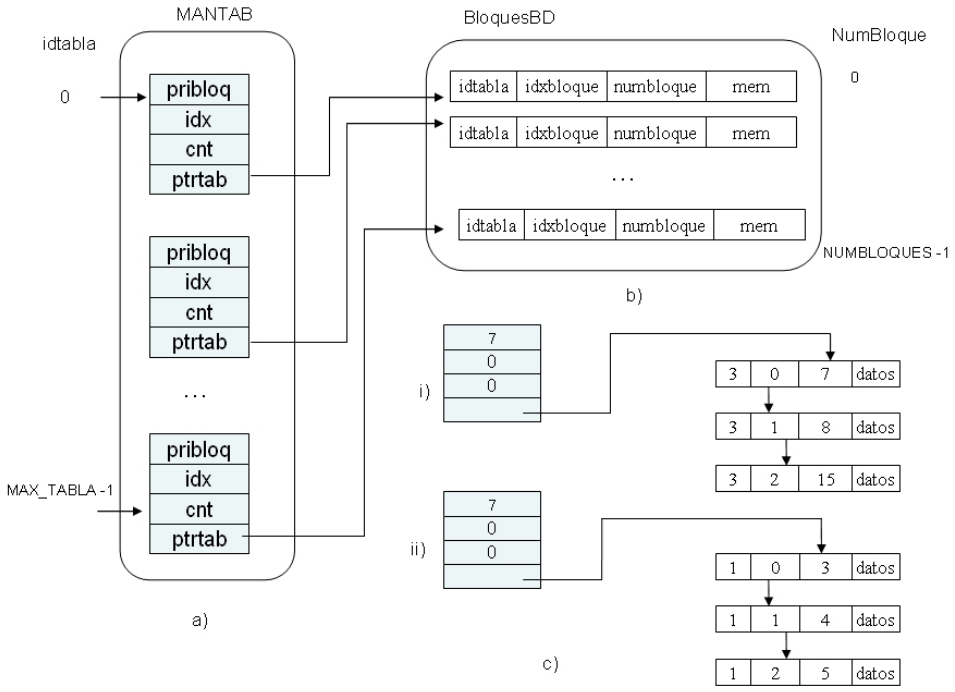

Figura 6.9: Organización y Manipulador de una tabla de la Base de Datos

6.3. Manejo de Errores en el Manejador de Base de Datos en Tiempo Real

Cuando un proceso invoca a una primitiva del Manejador de Base de Datos, antes de ejecutarla, el MBDTR lleva a cabo una serie de validaciones para detectar posibles parámetros erróneos ó eventos que no se esperaban. Para esto, en el Kernel se tiene implementado un manejador de errores de tal forma que si llega a detectar un error, el sistema manda un mensaje de error de acuerdo a un código de error perteneciente a una de las primitivas del MBDTR, y el Kernel debe terminar su ejecución debido a que en un

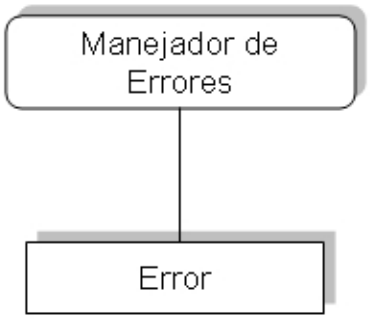

Figura 6.10: Procedimiento del Manejador de Errores

sistema de tiempo real el manejo de los procesos son críticos y cualquier falla puede causar una catástrofe, es por eso que el Kernel no debe continuar. El MBDTR hace uso de una primitiva del manejador de errores (ver Figura 6.10) para mostrar los errores que sucedan y esta tiene la siguiente estructura:

Error :

NOMBRE: void Error(unsigned int E)

FUNCION: Se encarga de mostrar al usuario en caso de que ocurriera un error, la posible causa que lo generó, y restaura la rutinas de interrupción del teclado y el reloj por las que tenia el sistema antes de iniciar el Kernel.

ENTRADAS: El código que identifica al error (E) que ocurrió.

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

A continuación se una descripción de cada uno de los posibles errores que pueden ocurrir en la ejecución del MBDTR en el Kernel.

Códigos de Error

Primitiva Selección

ERROR 100: Ocurre cuando se intenta abrir una tabla que no existe en la base de datos.

ERROR 101: Este error es provocado debido a que la tabla en la base de datos esta corrupta.

ERROR 102: Este error se genera debido a que la tabla no se pudo crear en la base de datos.

ERROR 103: Este error ocurre cuando el identificador de la tabla esta fuera de rango.

ERROR 104: Este error ocurre cuando la tabla no se puede crear en disco.

ERROR 105: Sucede cuando hay un error de escritura en disco.

Primitiva Proyeccion

ERROR 200: Este error ocurre cuando se intenta abrir una tabla de la base de datos que no existe.

ERROR 201: Este error sale cuando se detecta que la tabla en la base de datos esta corrupta.

ERROR 202: Este error sucede cuando la tabla no se puede crear en la base de datos.

ERROR 203: Este error ocurre cuando el nombre de tabla es errónea.

ERROR 204: Este error ocurre cuando el identificador de la tabla esta fuera de rango.

ERROR 205: Este error se genera cuando una tabla de la base de datos se intenta crearla en disco.

ERROR 206: Sucede cuando hay un error de escritura en disco.

Primitiva Insertar

ERROR 300: Este error ocurre cuando se intenta abrir una tabla de la base de datos que no existe.

ERROR 303: Este error sale cuando el nombre de tabla es errónea.

ERROR 304: Este error sucede cuando no se encuentra el separador del valor de la columna.

ERROR 305: Este error sale cuando no se encuentra el nombre de la columna especificada.

ERROR 306: Este error ocurre cuando la longitud del registro esta excedida del tamaño permitido.

ERROR 307: Este error sale cuando en la tabla no se pueden insertar mas registros debido a que la base de datos esta llena.

Primitiva Delete

ERROR 400: Este error ocurre cuando la columna especificada no existe en la tabla de la base de datos.

ERROR 401: Este error se genera cuando el número de registros eliminados no coincide con el número de registros marcados para eliminar.

ERROR 402: Este error ocurre cuando el identificador de la tabla esta fuera de rango.

Primitiva Update

ERROR 500: Este error sale cuando se intenta abrir una tabla de la base de datos que no existe.

ERROR 501: Este error ocurre cuando en la condición no se encontró el símbolo = después del nombre de la columna.

ERROR 502: Este error se genera cuando un nombre de columna no existe.

ERROR 503: Este error ocurre cuando el valor de la cadena no esta bien terminada.

ERROR 504: Este error sucede cuando el valor de la columna de tipo real no tiene punto decimal.

ERROR 505: Este error ocurre cuando el tipo de dato de la columna no existe.

Primitiva Join

ERROR 600: Este error ocurre cuando se intenta abrir la tabla T1 que no existe en la base de datos.

ERROR 601: Este error ocurre cuando se intenta abrir la tabla T2 que no existe en la base de datos.

ERROR 602: Este error se genera cuando la tabla T1 de la base de datos esta corrupta.

ERROR 603: Este error se genera cuando la tabla T2 de la base de datos esta corrupta.

ERROR 605: Este error sucede cuando la tabla del resultado no se pudo crear en la base de datos.

ERROR 607: Este error ocurre cuando el identificador de la tabla esta fuera de rango.

ERROR 608: Este error sucede cuando una tabla no se pudo crear en disco.

ERROR 609: Este error sale cuando hubo en error de escritura en disco.

Otros códigos

ERROR 800: Este ocurre cuando el proceso no puede cargar la tabla a la base de datos.

6.4. Configuración e Inicialización del Kernel y Manejador de Base de Datos en Tiempo Real

6.4.1. 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 principales de configuración que el Manejador de Base de datos utiliza 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 que puede manejar el kernel depende de (MAXPRO=NTAREAS+2), se suman dos procesos a NTAREAS por la primera y última tarea.

3.- El número máximo de niveles de prioridad manejados por el Kernel (MAXPRIO).

4.- El número total de semáforos que pueden utilizarse (MAXSEM).

5.- El tiempo de computo permitido en el mecanismo de planificación FIFO Round Robin (QUANTUM).

6.- El número de interrupciones por segundo (NUMINTSS)

Para que el Manejador de Base de Datos pueda ejecutarse en el Kernel, debe estar configurado de la siguiente manera:

- Utiliza Planificador FIFO Round Robin
- Quantum utilizado es de 5.
- El usuario puede crear hasta 5 tareas
- Maneja hasta 7 niveles de prioridad
- El número de interrupciones por segundo para el kernel es de 20.

6.4.2. Configuración del Manejador de Base de Datos en Tiempo Real

El Manejador de Base de Datos de Tiempo Real cuenta con un archivo de configuración llamado CONST_BD.H, en el cual el usuario puede modificar el contenido de las variables para que MBDTR se adapte a sus necesidades. Entre las cosas que puede modificar se encuentran:

1.- El número máximo de tablas en la base de datos (MAX_TABLA).
2.- El número máximo de columnas en una tabla (MAXCAMPOS)
3.- La longitud máxima del registro en una tabla (MAXLREG)
4.- La longitud máxima del dato de una columna (MAXLDAT)
5.- El tamaño del área de memoria ocupada por la Base de Datos (MAXMEM)
6.- El tamaño del bloque de memoria para el manejo de la tabla (TAMBLOQUE)
7.- El número máximo de bloques de memoria (NUMBLOQUES)

Por defecto el Manejador de Base de Datos en Tiempo Real se encuentra configurado de la siguiente manera:

- Utiliza 15 bytes para el nombre de las columnas
- Permite una longitud por registro de 1024 bytes
- La longitud del dato para una columna es 255 bytes
- El área de memoria asignada al MBDTR es de 10000 bytes
- La memoria del MBDTR esta organizada en bloques de tamaño de 512 bytes
- Maneja hasta 20 bloques para el MBDTR

6.4.3. Inicialización del Kernel y del Manejador de la Base de Datos en Tiempo Real

Inicialización del Kernel

Para obtener el programa ejecutable MKOS.EXE es necesario crear el proyecto *MKOS.PRJ*. Inicialmente este proyecto incluye los siguientes archivos:

LIBRERIA.CPP	*PRIPRO.CPP*	*PRITAREA.CPP*
TECLADO.CPP	*PRITIM.CPP*	*KERNEL.CPP*
ERRORES.CPP	*PRISEM.CPP*	
MANREG.CPP	*PRIBUZ.CPP*	
MANCOLAS.CPP	*TIMER.CPP*	
MANCPU.CPP	*ULTIMTAR.CPP*	

El programa *KERNEL.CPP* es el programa principal (*main()*), aquí es donde se inicializa el sistema y se indican las tareas o procesos que se ejecutarán en el Kernel. El proceso de inicialización del sistema contempla lo siguiente:

1.- Se cambia la resolución del “timer” para configurar al temporizador del procesador para que genere 20 interrupciones por segundo (*NUMINTSS=20*) .

2.- La primitiva *inicializa()* la cual se encarga de:

- Inicializar cada una de las tareas poniendo su estado en “NUEVO”.
- Inicializar colas de procesos utilizadas por el Kernel (cola de procesos listos, retrasados, de semáforos y de procesos bloqueados por buzón).
- Inicializar la Base de Datos
- Inicializar el Monitor Lectores-Escritores

3.- Crear los procesos primero y último.

4.- Crear los procesos hechos por el usuario.

5.- Activar a los procesos primero y ultimo

6.- Mandar a ejecutar la primer tarea.

Inicialización del Manejador de Base de Datos en Tiempo Real

Los archivos que deben de incluirse en el proyecto creado (MKOS.PRJ) y que tienen que compilarse junto con los archivos del Kernel, son los siguientes:

GVARMBD.CPP *LIBMBD.CPP* *MANBD.CPP* *PRIMON.CPP*
LECESC.CPP *PROC1.CPP*

Como se puede notar en la inicialización del Kernel, esta implícita la inicialización del MBDTR. Esta inicialización consiste explícitamente en lo siguiente:

- Se utiliza el procedimiento *inicializa_tab()* llamado desde la primitiva *inicializa()*. Este procedimiento se encarga de inicializar los manejadores de tablas de la Base de Datos y también inicializa el área de memoria fija asignada a la Base de Datos, la cual esta se organiza en bloques de tamaño fijo.

- También es posible que se puedan crear tablas o cargar tablas almacenadas en disco a la menoria de la Base de Datos. Aquí se crean dos tablas: *Deptos* y *Emps*.

- Se inicializa el monitor Lectores-Escritores, esto es:

  a) Se inicializan las variables locales del monitor: *LecEsc* apunta a la estructura *LECTOR_ESCRITOR*, el contador de lectores activos en el monitor *ContLect = 0*, la variable de estado que indica si hay o no escritor activo en el monitor *ProcEsc = 0* y la variable *BD* que apunta a el área de memoria de la Base de Datos (*MemTab*).

  b) Se inicializan las variables condición: *LEER = 2* y *ESCRIBIR = 3*.

  c) Se inicializan los semáforos *MUTEX = 0* y *NEXT = 1*;

d) y se crea al monitor en el sistema utilizando la primitiva *CreateMonitor().*

Con respecto a los procesos del usuario estos se pueden hacer en uno o más archivos independientes, después estos archivos simplemente se agregan al proyecto y se compilan juntos con el Kernel y el MBDTR para obtener el programa ejecutable (*MKOS.EXE*).

En esta configuración, el archivo *PROC1.CPP* contiene dos procesos de usuario que realizan consultas y actualizaciones a la Base de Datos. Estos son creados al iniciar el Kernel cuando se ejecuta la *primer tarea.*

Una vez que esta listo todo lo necesario para que se ejecute el Kernel y con él, el Manejador de Base de Datos. El Kernel se ejecuta iniciando con la ejecución de la primera tarea. Una vez que es ejecutada la primer tarea esta activa todos los procesos que 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 ó FIN.

Capítulo 7

Conclusiones y Trabajo Futuro

7.1. Conclusiones

En la actualidad las bases de datos de tiempo real son parte de muchos tipos de sistemas los cuales son usados tanto en ámbito domestico, industrial, gubernamental, militar, etc. donde se demanda alta confiabilidad, por lo tanto los resultados que estos ofrecen deben de ser correctos, predecibles y precisos.

Como se mostró en esta tesis, las pruebas experimentales realizadas involucran a un sistema de base de datos en tiempo real, las cuales están hechas con un conjunto de tareas concurrentes que incluyen transacciones sobre una base de datos, donde las transacciones pueden realizar operaciones de lectura y actualización a una base de datos, garantizando con mecanismo de control de concurrencia; la integridad y la consistencia de los datos en la base de datos.

En forma resumida, en el desarrollo de esta tesis obtuvieron los siguientes resultados:

1.- Se creó un Manejador de Base de Datos en Tiempo Real como parte de un Kernel de tiempo real con capacidad de ser portado a cualquier otra plataforma portátil o empotrada,

es capaz de ejecutar procesos concurrentes los cuales pueden manipular de forma controlada una base de datos garantizando la integridad y la consistencia de los datos.

2.- Se estudio y se desarrollo los algoritmos para los monitores que sirvieron de bases para desarrollar los algoritmos para el monitor de lectores y escritores, utilizando este para mantener el control de acceso de tareas concurrentes a una base de datos compartida.

3.- Se obtuvo el Manejador de Base de Datos en Tiempo Real como parte de un Kernel de tiempo real con un código fuente fácil de entender y modificar, por lo que permite que se le puedan hacer mejoras y extensiones de futuros algoritmos de control de concurrencia, restricciones de precedencia y herencia de prioridades.

4.- Se hicieron pruebas experimentales de funcionalidad respecto al control de concurrencia sobre una base datos, utilizando un conjunto de procesos que interactúan concurrentemente sobre una base de datos y que realizan diferentes operaciones de lectura y actualización de datos que involucran a las primitivas del manejador de base datos y se comprobó que la ejecución de estas se realizó correctamente, así como también se hizo la verificación de cada uno de los resultados obtenidos.

7.2. Trabajo Futuro

El área de los sistemas de tiempo real es muy amplia, y actualmente uno de los más importantes, son los sistemas de base datos en tiempo real. Ahora que contamos con este Manejador de Base de Datos en Tiempo Real cuyo código fuente conocemos podemos realizar los cambios y extensiones de acuerdo a las necesidades de nuestra aplicación. A continuación se detallan algunos de estos cambios y extensiones.

- **Extensión a estrategias de indexamiento de datos.** Se puede mejorar el desempeño de las transacciones que se hacen a la base de datos, extendiendo la funcionalidad para el soporte de indexamiento de datos, utilizando alguna estrategia de indexado para realizar búsquedas de datos mas eficientes como son B+-tree y T-tree, la elección de la estrategia dependerá de las necesidades de la aplicación que se desarrolle.

- **Extensión a Sistemas Distribuidos y a Sistemas con Múltiples Procesadores.** El Manejador de Base de Datos ha sido diseñado para que funciones sobre un Kernel de tiempo real que se ejecuta en una computadora con un sólo procesador, sin embargo, se le puede mejorar agregándole primitivas que permitan la comunicación entre tareas de kernels que se estén ejecutando en distintas computadoras con la finalidad de poder controlar, monitorear, comunicar y administrar sistemas de tiempo real distribuidos. Otra mejora importante, es que la base de datos sea compartida y accesada con tareas que se ejecutan en una computadora con 2 o más procesadores.

- **Extensión para el uso de un lenguaje de consulta estructurado (SQL).** Muchos de los sistemas de base de datos en tiempo real hoy en día tienen incorporado un lenguaje de consulta estructurado como es SQL. Por lo que podría ser una alternativa utilizar este sistema para experimentar el desarrollo de un lenguaje como SQL.

Bibliografía

[1] Pervasive Software Inc. *http://www.pervasive.com*

[2] Polyhedra Plc. *http://www.polyhedra.com*

[3] MBrane Ltd. *http://www.mbrane.com*

[4] Sleepycat Software Inc. *http://www.sleepycat.com*

[5] TimesTen Performance Software. *http://www.timesten.com*

[6] P.S. Yu, K. Wu, K. Lin, and S. H. Son. *On Real-Time Databases: Concurrency Control and Scheduling*. Proceedings of the IEEE, 82(1):140-157, January 1994.

[7] F. Baothman, A. K. Sarje, and R. C. Joshi. *On Optimistic Concurrency Control for RTDBS*. In preceedings IEEE Region 10 International Conference on Global connectivity In Energy, Computer Society, December 1998.

[8] B. Adelberg, B. Kao, and H. Garcia-Molina. *Overview of the Stanford Real-Time Information Processor (STRIP)*. SIGMOD Record, 25(1):34-37, 1996.

[9] S. F. Andler, J. Hansson, J. Eriksson, J. Mellin, M. Berndtsson, and B. Eftring. *DeesDS Towards a Distributed and Active Real-Time Database System*, ACM SIGMOD Record 25(1):38-40, 1996.

[10] J. A. Stankovic, S. H. Son, and J. Liebeherr. *Real-Time Databases and Information Systems*, chapter BeeHive: Global Multimedia Database Support for Dependable, Real-Time Applications, pages 409-422. Kluwer Academic Publishers.

[11] J. Zimmermann and A. P. Buchmann. REACH. In N. Paton (ed): *Active Rules in Database Systems*, Springr-Verlag, 1998.

[12] J. Taina and K. Raatikinen, RODAIN: *A Real-Time Object-Oriented Database System for Telecommunications*. In Proceedings of the work. Shop on Databases: active an real-time, pages 10-14. ACM Press, November 1996.

[13] Y-K. Kim, M. R. Lehr, D. W. George, and S.H. Song. *A Database Server for Distributed Real-Time Systems; Issues and Experiences*. In Preceedings of the Second IEEE Workshop on Parallel and Distributed Real-Time Systems, pages 66-75. IEEE Computer Society, April 1994.

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

[15] Silberschatz, Galván and Gagne. *Operating Systems Concepts, Sixth Edition*, Ed. John Wiley and Sons, Inc. 2002.

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

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

[18] José Ismael Ripoll. *Planificación en Sistemas de Tiempo Real*. Universidad Politécnica de Valencia. Departamento de Informática de Sistemas y Computadores. 2002.

[19] 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.

[20] 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.

[21] 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.

[22] K. Ramamritham and P. Chrysanthis. *"In Search of Acceptability Criteria: Database Consistency Requeriments and Transaction Correctness Properties", in Distributed Object Management,* Ozsu, Dayal, and Valduriez Ed., Morgan Kaufmann Publishers, 1992.

[23] H. F. Korth, Soparkar, Silberschatz. A. *"Triggered Real-Time databases with consistency constraints"*, Proceedings of the Conference on Very Large Data Bases, 1990.

[24] U. Dayal, et. Al. *"The HiPAC Project: Combining Active Database and Timing Constraints"*, SIGMOD Record, 17, 1, March 1988, 51-70.

[27] P. O'Neil,K. Ramamritham, and C. Pu. *"Towards Predictable Transaction Executions in Real-Time Databases Systems"*, Technical Report 92-35, University of Massachusetts, August, 1992.

[25] Peter A. Franaszek,John T. Robinson, and Alexander Thomasian. *"Access Invariance and its Use in High Contention Environments"*, Proceedings of the Sixtth International Conference on Database Engineerings, 1990, pp. 47-55.

[26] K. Ramamritham, J. Stankovic, and P. Shiah. *"Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems,"* IEEE Transactions on Parallel and Distributed Systems, Vol. 1, No. 2, April 1990, pp.184-194.

[28] L. Sha, R. Rajkumar, and J. P. Lehoczky. *"Concurrency Control for Distributed Real-Time Databases,"* ACM SIGMOD Record, March 1988.

[29] R. Abbott and H. Garcia-Molina. *"Scheduling Real-Time Transactions: A Performance Evaluation,"* Proceedings of the 14th VLDB Conference, Aug. 1988.

[30] R. Abbott and H. Garcia-Molina. *"Scheduling Real-Time Transactions with Disk Resident Data,"* Proceedings of the 15th VLDB Conference, Aug. 1989.

[31] A. P. Buchmann, D. R. McCartty, M. Chu, and U. Datal. *"Time-Critical Database Scheduling: A Framework for Integrating Real-Time Scheduling And Concurrency Control"*, Proceedings of the Conference on Data Enginnering, 1989.

[32] J. R. Haritsa, M. J. Carey and M. Livny. *"On Being Optimistic about Real-Time Constraints,"* Proceedings of ACM PODS, 1990.

[33] J. Huang, J. A. Stankovic, D. Towsley and K. Ramamritham. *"Experimental Evaluation of Real-Time Transaction Processing,"*, Proceedings of the Real-Time Systems Symposium, Dec. 1989.

[34] J. Huang, J. A. Stankovic, D. Towsley and K. Ramamritham, "On using Priority Inheritance in Real-Time Databases," Proceedings of the Real-Time Systems Symposium, December 1991.

[35] J. Huang, J. A. Stankovic, D. Towsley and K. Ramamritham. *"Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes,"* Proceedings of the Conference on Very Large Data Bases, Dec. 1991.

[36] J. E. B. Moss. *Nested Transactions: An Approarch to reliable distributed computing*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, April 1981.

[37] S. H. Son, Y. Lin, and R. P. Cook. *"Concurrency Control in Real-Time Database Systems", in Foundations of Real-Time Computing: Scheduling and Resource Management*, edited by Andre van Tilborg and Gary Koob, Kluwer Academic Publishers, pp. 185-202, 1991.

[38] J. A. Stanckovic, k. Ramamritham, and D. Towsley. *"Scheduling int Real-Time Transaction Systems,"* in Foundations of Real-Time Computing: Scheduling and Resource Management, edited by Andre van Tilborg and Gary Koob, Kluwer Academic Publishers, pp. 157-184, 1991.