TABLA DE CONTENIDOS

INTRODUCCIÓN GENERAL

En la actualidad existe una gran variedad de sistemas usados para controlar procesos que ocurren en el mundo real, y el número de este tipo de sistemas está en constante crecimiento.

Hay una gran cantidad de sistemas para control del tráfico aéreo, redes de fibra óptica, monitoreo de pacientes, pilotos automáticos, control de robots, distribución del fluido eléctrico, cadenas de producción de fábricas, medición, control y seguimiento de satélites, funcionamiento de centrales de energía nuclear y otros sistemas similares.

Este tipo de sistemas, llamados de Tiempo Real, tienen como función principal la comunicación con el mundo físico, en vez de hacerlo con un operador humano, lo cual implica que deben ejecutar sus funciones de acuerdo con los sucesos que ocurren en el mundo real. Por lo tanto, el orden de ejecución depende no sólo de la estructura del programa, sino de lo que ocurre en el entorno.

Si bien existen diferentes definiciones y clasificaciones de los Sistemas de Tiempo Real, las características principales de esta clase de sistemas son:

- Deben responder a diversos eventos externos, asegurando un tiempo de respuesta máximo determinado (llamado deadline o meta de la tarea). El tiempo de respuesta es el que transcurre entre la presentación de un conjunto de entradas al sistema y la obtención de sus resultados asociados.

- La secuencia de ejecución de las tareas del sistema no sólo está determinada por decisiones del sistema, sino por eventos que ocurren en el mundo real.

Un gran número de aplicaciones de Tiempo Real se encuentran en la industria, y en la actualidad existe un número creciente de computadoras controlando procesos industriales. Esto ha incrementado el número de actividades que se automatizaron. No sólo la computadora puede directamente controlar la operación de la planta, sino que además puede proveer a los gerentes e ingenieros de una fotografía del estado de su funcionamiento. Este rol se conoce como Supervisión de Procesos.

A pesar de ello, las principales aplicaciones de Supervisión de Procesos carecen de soluciones genéricas aceptables, lo que ha provocado que el diseño y desarrollo de gran parte de estos sistemas en la actualidad se construyan utilizando técnicas específicas para cada aplicación.

No existe aún demasiados entornos de desarrollo que solucionen todos los problemas relacionados con las restricciones en este tipo de sistemas y por ende, las aplicaciones siguen construyéndose "ad hoc" para cada uno de los problemas. Además existen pocos Sistemas Operativos y muy pocos lenguajes de programación que

provean facilidades relacionadas con la ejecución de tareas con restricciones duras de tiempo.

Con este panorama en vista, el objetivo de esta Tesis es el diseño e implementación de una herramienta que permita el desarrollo de un grupo importante de aplicaciones de Tiempo Real (las de Supervisión de Procesos), facilitando el trabajo para los diseñadores y programadores.

Esta herramienta estará diseñada como un conjunto de clases independientes, para permitir la construcción de diferentes aplicaciones de Supervisión de Procesos con distintas finalidades y con muy poco esfuerzo.

Un Sistema de Supervisión de Procesos, formalmente llamados SCADA (Supervisory Control and Data Adquisition), puede descomponerse en tres niveles de servicio: Interfaz (Interface Service Level - ISL), Particulares (Particular Service Level - PSL) y Básicos (Basic Service Level - BSL). La herramienta a desarrollar se basará en el diseño y construcción del tercer nivel de servicio, BSL, el cual encapsula los servicios básicos de Supervisión que son comunes a todas las aplicaciones SCADA.

En los Sistemas de Tiempo Real se deben asegurar, entre otros, los requisitos de predictibilidad, planificabilidad y estabilidad. Para satisfacer estos requerimientos, se debe implementar una planificación de tareas para poder determinar el orden de ejecución de las mismas, de tal forma que se cumplan las metas (deadlines) de tiempo y recursos. Una tarea, desde el punto de vista del planificador, es una entidad (thread) a planificar, un módulo que se invoca para hacer una función.

La teoría de planificación de tareas (scheduling) en Tiempo Real se relaciona con el hecho de hacer cumplir las restricciones de tiempo de las tareas que ejecutan en el sistema. La complejidad de los algoritmos de planificación reside en la diversidad de restricciones que deben cumplir. Las restricciones de tiempo pueden especificarse en términos de distintos parámetros: el tiempo de llegada, el tiempo de listo, el peor caso de tiempo de ejecución y la meta de la tarea.

Un algoritmo de planificación debe considerar todos estos factores, así como también, las restricciones de recursos, precedencia, concurrencia, comunicación, criticidad y otras más. Para ello se utilizan distintas políticas y técnicas de planificación de tareas, tanto en Sistemas Operativos específicos para Tiempo Real, como en los Sistemas Operativos tradicionales.

En esta Tesis, se implementan diferentes algoritmos de planificación de Sistemas Operativos de Tiempo Real, emulados en una plataforma de Sistemas Operativos de propósito general.

Si bien, seguramente existirán diversas necesidades para instalar, implementar o utilizar Sistemas Operativos de Tiempo Real (RTOS), diferentes factores justifican la

utilización de Sistemas Operativos de Propósito General (GPOS) para aplicaciones de Supervisión de Procesos.

Para desarrollar esta herramienta, se utilizará la plataforma Win32 de Microsoft®, debido a sus características para Sistemas de Tiempo Real, a su popularidad y a su presencia en el mercado y en la industria. Esta plataforma soporta los requerimientos principales para este tipo de sistemas y se analizarán sus diferentes versiones de los Sistemas Operativos: Windows2000, WindowsNT, Windows98 y Windows95.

El lenguaje de programación para desarrollar esta herramienta será C++, dado que es un lenguaje de bajo nivel por poseer todas las características de C y, a su vez, las clases del C++ brindan las facilidades de programación y abstracción necesarias, propias de un lenguaje de alto nivel.

Este trabajo está organizado de la siguiente manera: en los primeros cuatro capítulos se describe la parte de introducción y teoría de esta Tesis; en los capítulos siguientes se estudia la herramienta desarrollada, su diseño e implementación; y por último, se mostrarán los resultados obtenidos, las conclusiones y posibles trabajos futuros.

En el primer capítulo se describirán los Sistemas de Tiempo Real y sus características; en el segundo se estudiarán las funciones de los Sistemas de Supervisión; en el tercer capítulo se estudiará el software de base para Tiempo Real y las características de Win32 para esta clase de sistemas; en el cuarto se describirán los algoritmos para la planificación en Tiempo Real, diferentes técnicas y su emulación en Sistemas Operativos de propósito general.

En el quinto capítulo se especifica el diseño de la herramienta, su diagrama de clases y su interface. En el sexto, se describe la implementación de la herramienta en Win32 y en su familia de Sistemas Operativos, estudiando las ventajas de cada uno de ellos para Tiempo Real. En el séptimo capítulo, se muestra un ejemplo de una aplicación SCADA, la cual utiliza los servicios provistos por la herramienta construida.

En los últimos capítulos se describen los resultados obtenidos, las conclusiones y se plantean posibles trabajos futuros.

1 - SISTEMAS DE TIEMPO REAL

En este capítulo se analizarán las características de los Sistemas de Tiempo Real, sus definiciones, requerimientos y se considerarán diversas clasificaciones de los mismos.

1.1 Características.

Una característica muy importante en los Sistemas de Tiempo Real está relacionada con el tiempo de procesamiento de las aplicaciones. En sistemas tradicionales rara vez es crítico un control exacto de tiempo. Si bien suele ser favorable procesar a mayor velocidad, los resultados siguen siendo válidos si se obtienen con algún retraso. En cambio, en los Sistemas de Tiempo Real, los resultados obtenidos con retraso pueden ser no válidos en el momento de ser utilizados. Por este motivo se dice que estos sistemas deben actuar en "tiempo real".

Otra característica fundamental en los Sistemas de Tiempo Real es que deben atender las diversas demandas a medida que se presentan debiendo considerar los eventos externos, lo cual impide la organización del sistema en base a factores internos.

En resumen, en un Sistema de Tiempo Real se desea obtener respuestas correctas en un tiempo máximo determinado (llamado **meta** o **deadline**) y el orden de ejecución de las tareas depende de la ocurrencia de eventos que suceden en el mundo real.

Típicamente un Sistema de Tiempo Real consta de un sistema de control y de un sistema controlado. A modo de ejemplo: en una Estación Terrena de control satelital, el sistema de control es la Estación Terrena y el sistema controlado son los satélites de telecomunicaciones. Por otro lado, en este mismo ejemplo, el sistema controlado es todo el equipamiento de la planta, las antenas parabólicas, los equipos de transmisión y recepción, los modems de banda base y todos los periféricos necesarios para su operación; y el sistema de control está formado por workstations, procesos, equipamiento de comunicaciones y las interfaces entre las computadoras y la Estación Terrena.

En este tipo de sistemas se destacan tres tipos de funciones bien diferenciadas:

- Monitoreo: deben obtener información acerca del estado actual del entorno físico del sistema.

- Control: deben realizar los cálculos necesarios para permitir controlar el proceso de acuerdo a los valores leídos en la función de monitoreo.

- Actuación: deben alterar el estado actual del mundo real para mantener determinados parámetros dentro de un rango de valores especificados.

Las tareas de monitoreo y actuación realizan su operación a través de una serie de dispositivos de interfaz, incluyendo conversores AD/DA (Analógico-Digital/Digital-Analógico), líneas de Entrada/Salida digitales, generadores de pulso, etc. Para poder operar o manejar cada tipo de dispositivo, se necesita de rutinas especiales de software: los drivers de Entrada/Salida.

El sistema de control interactúa con el entorno utilizando información disponible del mundo real. Por ello es imperativo que los valores de estado del entorno que utiliza el sistema de control, sean consistentes con el estado real. Si esto no es así, el efecto de las actividades del sistema pueden ser desastrosas.

A continuación se enumeran las principales características de esta clase de sistemas [Wai97]:

- Deben responder a diversos eventos externos, asegurando un tiempo de respuesta máximo determinado (*meta* de la tarea). El tiempo de respuesta es el que transcurre entre la presentación de un conjunto de entradas al sistema y la obtención de sus resultados asociados.

- La secuencia de ejecución de las tareas del sistema no sólo está determinada por decisiones del sistema, sino por eventos que ocurren en el mundo real.

- Deben presentar alto nivel de seguridad, dado que es crítico la confiabilidad del sistema así como su respuesta ante situaciones de sobrecarga.

- Las demandas del ambiente externo suelen ser en paralelo, provocando problemas de planificación y prioridades.

- No pueden hacer roll back y reiniciar su ejecución desde un contexto preexistente.

- Son de tiempo infinito, lo que significa que deben estar preparados para recuperarse.

Para medir la performance de estos sistemas se deben considerar patrones diferentes a los usados en los sistemas tradicionales (throughput, turnaround, grado de utilización de recursos), como los que se describen a continuación:

- Predictibilidad: se desea obtener respuesta predecible ante eventos urgentes. No importa la velocidad de dichas respuestas, sino asegurar que las mismas siempre se obtengan antes de una *meta* dada.

- Alto grado de planificabilidad: se trata de maximizar la utilización de los recursos del sistema (especialmente el procesador), asegurando los requerimientos de tiempo de las tareas.

- Estabilidad ante sobrecargas momentáneas: cuando el sistema está sobrecargado y es imposible cumplir las *metas* de todas las tareas, aún se debe garantizar el cumplimiento de algunas tareas críticas elegidas.

- Confiabilidad: las restricciones de tiempo real no pueden cumplirse si los componentes del sistema no son confiables, ya que el costo de una falla del mismo puede exceder la inversión del proyecto y el objeto controlado.

- Adaptabilidad: cuando ocurren cambios en la configuración del sistema, en las especificaciones o en el estado del mismo, el sistema debe ser capaz de adaptarse, de forma tal de seguir cumpliendo con las *metas* de las tareas.

1.2 Definiciones.

Un Sistema de Tiempo Real puede definirse como el que controla un cierto entorno, recibiendo datos, procesándolos y retornando los resultados con una restricción de tiempo, afectando el funcionamiento del medio que controla.

Existen diversas definiciones de distintos autores sobre este tipo de sistemas. Una de ellas llama Sistema de Tiempo Real a un sistema en el cual el **tiempo** es el recurso más precioso a manejar. Las tareas deben ser asignadas y planificadas de tal forma de que puedan ejecutarse antes que sus plazos expiren. Un segundo concepto de un Sistema de Tiempo Real es la **confiabilidad**, que es considerada crítica ya que una falla en este tipo de sistemas puede provocar un desastre económico o la pérdida de vidas humanas.

El término “tiempo real” se utiliza para referirse a sistemas en los cuales la ejecución de las tareas está determinada por el transcurso del tiempo u **ocurrencia de eventos externos**, y los resultados obtenidos pueden depender del momento en que fueron ejecutados o del tiempo en que se demoró en hacerlo.

Es muy importante en este tipo de sistemas el entorno en el cual opera una computadora. Esta última es un componente activo y conjuntamente con su entorno forman una fuerte interrelación entre los tres componentes mencionados, básicamente entre temporalidad y confiabilidad.

Si bien estas definiciones están formuladas en forma genérica, se han propuesto diversas clasificaciones de los Sistemas de Tiempo Real, las que serán analizadas a continuación.

1.3 Clasificaciones.

Existen diversas clasificaciones de Sistemas de Tiempo Real de acuerdo a distintos criterios. A continuación se describen las mismas y sus principales características [Wai97]:

Clasificación Basada en:	Sistemas de Tiempo Real	Principales Características	Ejemplos
Restricciones de tiempo	✓ Duros.	Los cálculos siempre deben terminarse en un tiempo máximo especificado (**meta o deadline**), dado que es extremadamente crítica la obtención de resultados dentro de ese intervalo.	▪ Monitoreo de una planta nuclear. ▪ Monitoreo de pacientes. ▪ Sistemas de Defensa.
	✓ Blandos.	Los cálculos deben terminarse dentro de un tiempo promedio de ejecución inferior a un máximo especificado.	▪ Cajeros automáticos. ▪ Reserva de pasajes.
	✓ Firmes.	Son sistemas duros, en los cuales se toleran pérdidas con una probabilidad de ocurrencia muy baja.	▪ Sistemas de Supervisión de Procesos.
Escalas de tiempo	✓ Basados en eventos.	Las acciones son iniciadas a través de eventos.	▪ Sistemas en los cuales existen distintos eventos (cambios de estados) o señales que disparan actividades.
	✓ Basados en reloj.	La relación entre las escalas de tiempo de los eventos externos y las funciones ejecutadas en la computadora, está dada por el pasaje del tiempo.	▪ Sistemas periódicos en los cuales una señal equidistante en el tiempo inicia todas las actividades.

Clasificación basada en:	Sistemas de Tiempo Real	Principales Características	Ejemplos
	✓ Interactivos.	Se requiere que un grupo de operaciones se cumplan en un promedio de tiempo determinado.	▪ Un operador ingresando comandos en una terminal.
Integración con el sistema físico	✓ Embebidos.	Se utilizan para controlar hardware especializado en el cual se instala el sistema de computación.	▪ Controlador de un robot.
	✓ No embebidos.	Se subdividen en **orgánicos** si son completamente independientes del hardware en que ejecutan y en **débilmente acoplados** si pueden ejecutar en otro hardware reprogramando ciertos módulos.	▪ Sistemas que se ejecutan en equipos de telemetría y de telecomando.

Para especificar las tareas con requerimientos de tiempo, algunos autores han clasificado los distintos tipos de restricciones como tareas de tiempo crítico. Existen tres tipos de restricciones:

- máximas, en las que existe un tiempo máximo entre la ocurrencia de dos eventos;
- mínimas, en las que debe transcurrir un tiempo mínimo entre dos eventos;
- duracionales, en las que un evento debe ocurrir durante un tiempo.

Las clasificaciones analizadas en esta sección no son estrictas dado que muchos Sistemas de Tiempo Real son combinaciones de varias de ellas. Estas categorías pueden utilizarse como guías al encarar el desarrollo de sistemas de este tipo.

Por último, se enumeran las principales aplicaciones actuales de Sistemas de Tiempo Real, clasificadas de acuerdo al tipo de tarea realizada:

- Sistemas de transporte, control de tráfico vehicular y de tráfico aéreo;
- Soporte para automatización de fábricas y procesos industriales;

- Sistemas de sensores para monitorear patrones de tiempo, datos sísmicos, redes de distribución de energía;

- Satélites, redes de fibra óptica y canales de alta velocidad para transmitir datos, audio y video;

- Monitoreo de pacientes, tomografías computadas, pulmotores, resonancia magnética y otros equipamientos médicos de alta tecnología;

- Vigilancia, comando y control y otros sistemas de defensa.

2 - SISTEMAS DE SUPERVISIÓN DE PROCESOS: SCADA

En este capítulo se estudiarán algunos conceptos relacionados con un conjunto importante de las aplicaciones de Tiempo Real: los Sistemas de Supervisión de Procesos.

2.1 Características.

Todo sistema de control dispone de un conjunto de dispositivos de Entrada/Salida conectados con el entorno, que permiten la interacción con el sistema físico a controlar. El software que los maneja mantiene una imagen del mundo exterior. Para que el sistema de control pueda lograr sus objetivos, esta imagen se debe actualizar a intervalos específicos con las entradas obtenidas desde el entorno. Las tareas de control utilizan los datos de la imagen y como resultados de sus cálculos, actualizan el entorno ajustando estos valores de acuerdo a un valor de referencia o **set point**. Finalmente, los drivers de Entrada/Salida deben transmitir los datos entre la imagen y el mundo exterior para poder afectar al proceso deseado. Todo este proceso debe hacerse en un tiempo máximo, determinado por las características particulares del proceso físico a controlar.

Por otro lado, en muchas aplicaciones la comunicación con el operador es más compleja que un simple panel con indicadores. Así, los ingenieros y gerentes de planta, los controladores de tráfico aéreo y los operadores de fábricas automatizadas, necesitan información detallada de todos los aspectos de la operación de una planta, aeronave, sistema de radar, equipamiento, etc. Se trata que las computadoras y los Sistemas de Tiempo Real no sólo controlen la operación de su entorno, sino que también provean informes y reportes de diferentes niveles y gráficos completos del estado de las operaciones. Estos sistemas de información suelen conocerse con el nombre de **Sistemas de Supervisión de Procesos**.

Un Supervisor es un conjunto de programas encargados de coordinar, monitorear, controlar y prestar servicio a los diversos componentes del sistema. El Supervisor deberá planificar las tareas que el sistema deberá ejecutar, establecer y asignar prioridades entre ellas, monitorear y actualizar la imagen del mundo exterior, controlar la entrada y salida de datos, procesar errores y condiciones de alarmas.

Los Sistemas de Tiempo Real no pueden construirse con un patrón predeterminado de eventos. Una falla en alguno de los componentes de hardware del sistema necesitará procedimientos de emergencia. A su vez, deben existir funciones que les permitan detectar comportamientos anómalos, deben poder comunicar estados de emergencia al operador, a través de alarmas o mensajes y permitir intervenir al operador para solucionar el problema. En algunos casos deben proveer soluciones automáticas.

En la siguiente figura, se puede observar un diagrama de un sistema de control generalizado:

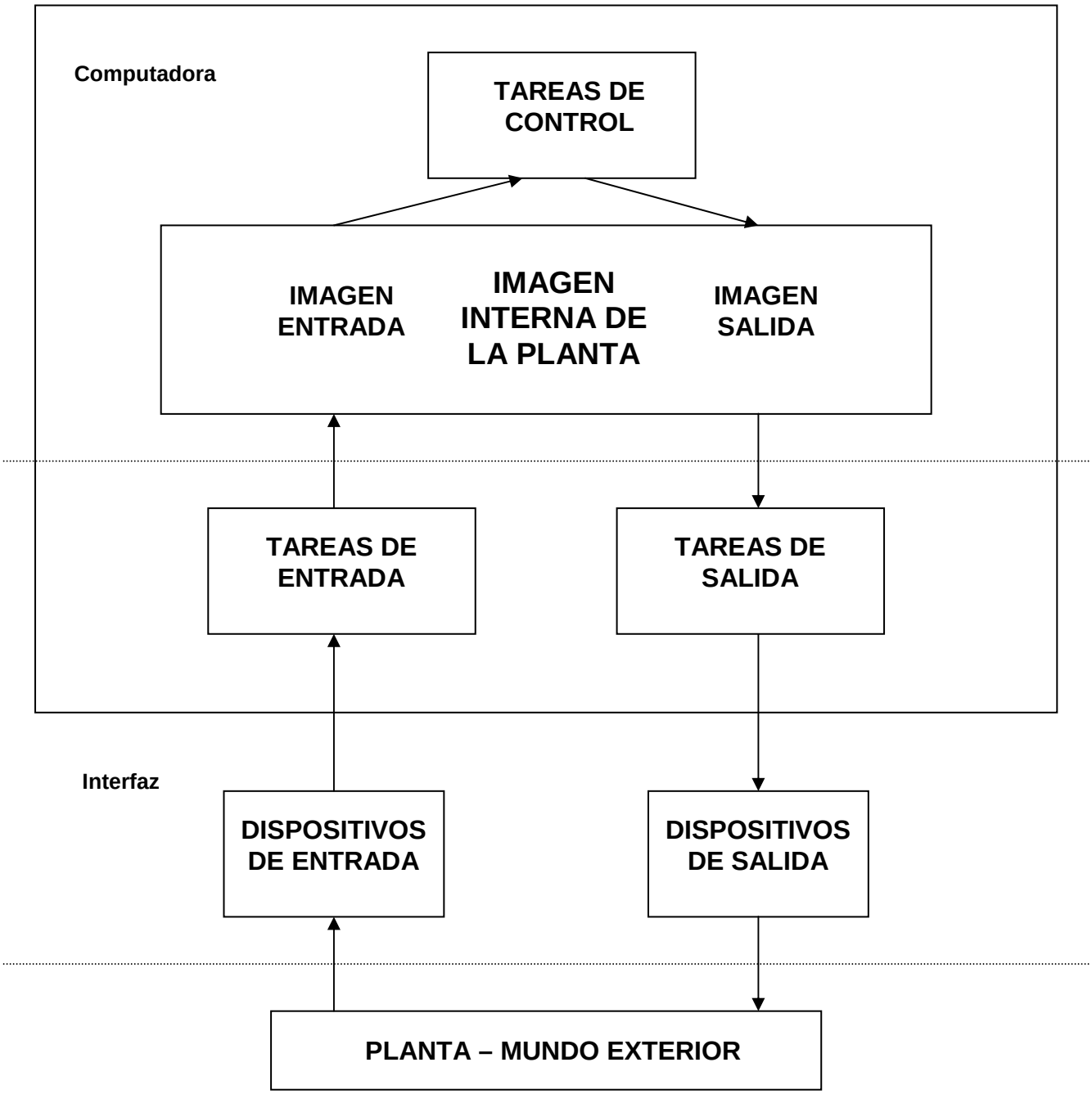


Figura 1 - Sistema de control generalizado.

Muchos de los esfuerzos de diseño y programación para sistemas de control de procesos están dirigidos a las facilidades de supervisión. Una estación típica para el operador suele tener, entre otros, los siguientes componentes de hardware y software [Wai97]:

- Unidades de monitoreo e impresión en donde se despliegan los mímicos (gráficos representando partes de una planta).
- Computadoras personales.
- Teclados diseñados especialmente.
- Equipamiento de multimedia para audio y video.
- Redes de área local, para distribuir la carga y funcionalidad de las operaciones.
- Sistemas Operativos multiusuario.
- Drivers de dispositivos estándar.
- Paquetes de software orientado a objetos para el desarrollo del GUI (Graphical User Interface).
- Bases de Datos relacionales.
- Sistemas de planeamiento y control de la producción.

Como se puede observar, los Sistemas de Supervisión también proveen una fuente importante de desarrollo. Estos sistemas tienen tiempos de respuestas restringidos, aunque sus *metas* suelen no ser duras. En general, en muchas aplicaciones es común la existencia de *metas* firmes e incluso algunas de las tareas de Supervisión de Procesos tienen *metas* blandas (o son simples tareas interactivas sin restricciones estrictas de tiempo).

En la siguiente sección se profundizarán cuáles son las principales funciones que deben ser provistas por esta clase de sistemas.

2.2 Funciones de Supervisión.

En general, los Supervisores tienen funciones comunes, independientemente de la aplicación que estén supervisando. A continuación se enumeran las principales funciones que deben ser provistas por un Sistema de Supervisión de Procesos [Ben97]:

⇒ Funciones de Supervisión del sistema.

- Monitoreo del sistema.
- Control del sistema.

⇒ Funciones de Administración del sistema.

- Asignación de prioridades a las tareas.
- Planificación de las tareas que se deben ejecutar.
- Atención de interrupciones.
- Manejo de excepciones en el sistema.
- Administración de colas y mensajes.

⇒ Funciones de Validación.

- Funciones que tratan de mantener y asegurar la consistencia y correctitud de los datos que se transmiten desde la planta a la computadora.

⇒ Funciones de registro de las operaciones del sistema y funciones de análisis.

- Los datos que comúnmente se almacenan son: estados del sistema y de sus operaciones, cambios en el estado de los equipos y registro de alarmas, eventos y errores.
- Herramientas para generar la creación de informes de acuerdo a las necesidades del operador, brindándole funciones como cálculos de valores promedio, varianzas, desvíos estándar y ordenamiento.

Con respecto al análisis de los datos y tratamiento de casos de excepción, los sistemas más simples dejan toda la tarea de análisis para el operador y los más complejos poseen mecanismos de inferencia y técnicas de Inteligencia Artificial que brindan apoyo para la toma de decisiones.

Los Sistemas de Supervisión de Procesos también se conocen como **sistemas SCADA (Supervisory Control and Data Adquisition)**. Un sistema SCADA es un ambiente computacional, capaz de integrar y analizar diversos procesos, de modo de obtener un diagnóstico de la evolución de los estados del sistema y actuar sobre ellos en caso de ser necesario.

Un punto clave en la adopción de un sistema SCADA es el nivel de facilidades que el sistema provee al operador de la planta. Es importante que este cuente con una interfaz simple y clara para la operación diaria de la planta. El sistema debe ofrecer facilidades para cambiar valores de referencia, ajustar variables, controlar condiciones de alarma, etc.

Los SCADA en general, proveen distintos niveles de información en su interfaz MMI (Man Machine Interface):

- Para el operador de la planta, displays gráficos mostrando estados de alarmas, presentando información de distintas áreas de la planta y facilidades para interactuar con la misma.

- Para el ingeniero de planta, incluye gráficos de tendencias y resúmenes de operaciones pasadas, así como también información para la toma de decisiones, asociadas con el mantenimiento de la planta y reemplazo de componentes.

- Para el gerente de la planta, acceso a cierto tipo de información, en formato de informes que resuman la operatoria diaria de la planta, que presenten datos históricos e informes estadísticos y que faciliten el seguimiento de la operatoria.

Cada usuario del sistema tendrá definido su perfil, es decir, qué operaciones les están permitidas realizar sobre la planta y por lo tanto, qué información deberá brindar el sistema para cumplir con sus tareas.

2.3 Arquitectura de un sistema SCADA.

En todo sistema SCADA, el monitoreo y control de los procesos relacionados deben hacerse en tiempo real. El tiempo necesario para la actualización de un conjunto de variables a partir de los valores obtenidos del proceso analizado o la actuación realizada en base a cambios del SCADA, tienen un máximo permitido para alcanzar sus objetivos.

Para simplificar la organización de estas tareas, los SCADA suelen manejar un conjunto mínimo de información que está definido por una transacción de cambio del estado de las variables del ambiente. Esta mínima unidad de información suele conocerse como **lazo** y está representado por un conjunto de variables y acciones que describen los atributos del sistema que se quiere monitorear.

El concepto más importante de un *lazo* es que puede ser usado para estructurar el proceso de desarrollo de un SCADA. Se puede imaginar a los distintos *lazos* como funciones independientes del sistema que deben ser integradas. Por ello, el sistema SCADA no es otra cosa que un conjunto de tareas que permiten integrar el análisis de varios *lazos* y obtener un diagnóstico de la evolución de los estados del sistema para estos *lazos* [Ben97].

Como ejemplo se puede presentar un sistema de control y supervisión de un edificio. Se pueden encontrar diferentes atributos como ser: alarmas contra invasores,

alarmas contra incendios y control de acceso a determinados lugares del edificio. Para cada uno de estos atributos existen sensores que leen el estado de las variables y se comunican con la estación de supervisión. Por ejemplo, un *lazo* de incendio estará compuesto por detectores de humo y de temperatura que alimentan a un conjunto de variables, las cuales serán transmitidas a los puestos de monitoreo a través de drivers especiales. Luego, se verificará si se está produciendo un incendio y en caso de confirmarse, se deberán accionar las alarmas correspondientes y activar procedimientos especiales como el lanzamiento de agua y espuma, poner fuera de funcionamiento los ascensores, etc. El *lazo* de incendio está conformado por todas las variables involucradas en esta secuencia de eventos. En forma análoga, se pueden definir *lazos* de seguridad, ambientación, etc. y cada uno de ellos puede ser usado como base para almacenar los valores de sus variables en un registro histórico para su posterior análisis.

Si se define a un *lazo* como un conjunto de variables y no se toma en cuenta los equipos que son utilizados para tener sus valores respectivos, entonces un sistema SCADA es un ambiente computacional que integra modelos de varios *lazos*. Un estado del sistema es un conjunto de n-uplas, conteniendo los valores para cada *lazo* [Ben97].

Por otro lado, se puede identificar un conjunto de variables que conforman cada *lazo*. Las variables pueden ser de diferentes tipos pero, en general, se aceptan variables analógicas y digitales. El conjunto de todas las variables del sistema SCADA conforman una Base de Datos y cada componente de la misma se conoce con el nombre de **punto de supervisión**.

La función primordial de los *puntos de supervisión* es el monitoreo de variables del proceso, las cuales se deben muestrear periódicamente. En general, sus *metas* son firmes y suelen ser comunes frecuencias de 100 milisegundos o mayores [Wai97]. Los puntos, también pueden usarse para control, despliegue de datos, ejecución de procesos asociados y cualquier otra función de supervisión.

Todos los puntos de la Base de Datos pueden estar usados por las tareas de supervisión. La Base de Datos puede estar distribuida y cambiar a medida que transcurre el tiempo.

Los sistemas SCADA complejos pueden tener Bases de Datos con miles de puntos. Alguno de ellos se pueden calcular como combinación de valores de varios puntos y otros puntos son filtrados e ingresados como si fueran leídos directamente. Las operaciones típicas que se pueden realizar en tiempo de ejecución son: agregar, eliminar, modificar y consultar los *puntos de supervisión*.

La utilización de una Base de Datos central es primordial en todo sistema SCADA ya que permite almacenar todas las variables de los *lazos*. Puede ser usada por las tareas que manejan el equipamiento de la planta, por las rutinas de conexión con otros sistemas y por el propio sistema SCADA, como se muestra a continuación:

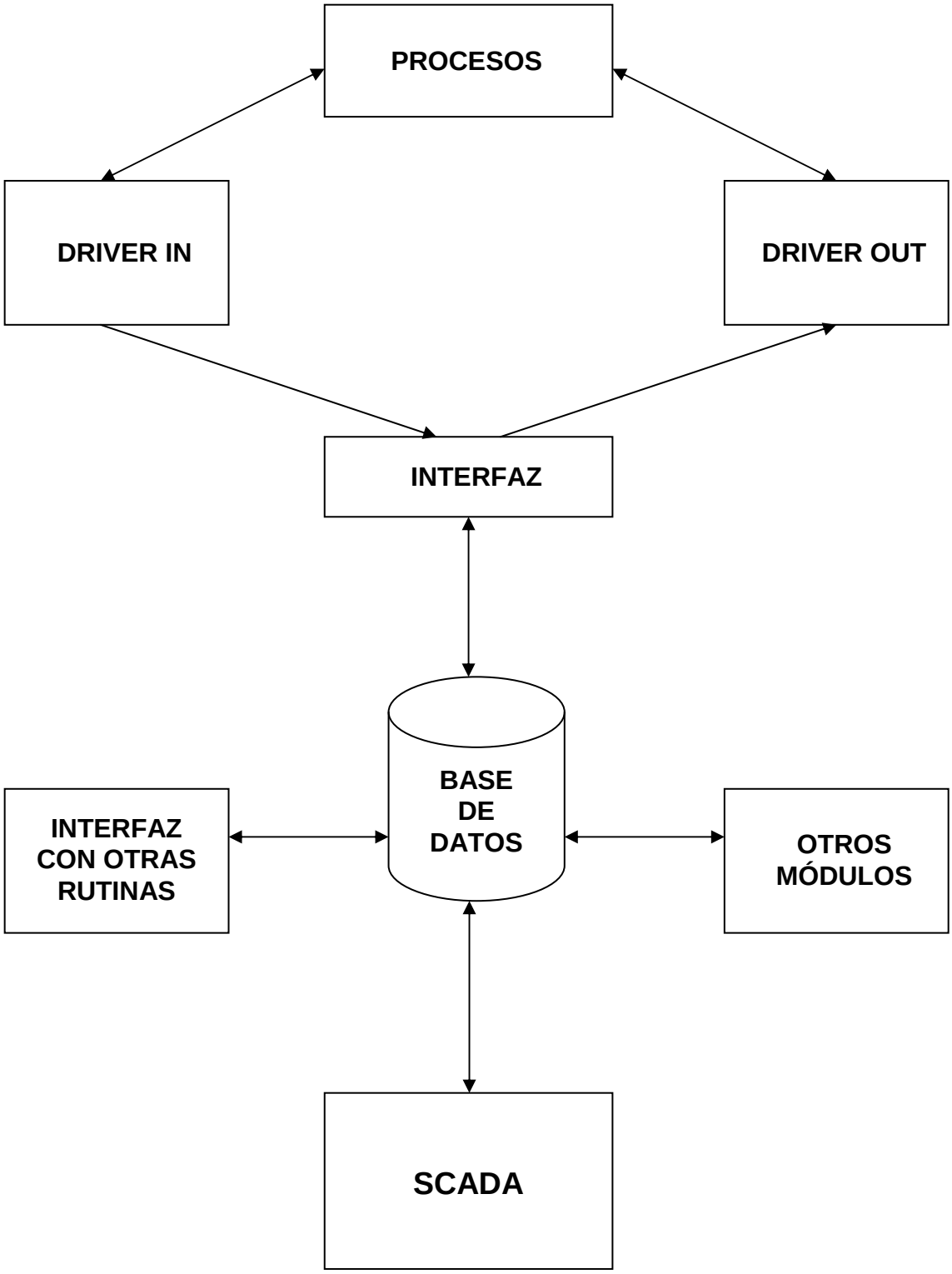


Figura 2 - Arquitectura de un sistema SCADA.

2.4 Niveles y categorías de un sistema SCADA.

Luego de haber estudiado las distintas facilidades que deben ser provistas por los sistemas SCADA, se pueden identificar los siguientes niveles de servicios comunes a todos ellos [Wai97]:

Niveles de Servicio	Características
Interface Service Level – ISL	▪ Encapsula todos los programas que proveen la interfaz entre el Supervisor y el operador. ▪ Provee pantallas gráficas, mímicos, alarmas sonoras, mensajes, etc. ▪ La implementación de este conjunto de programas cambia de acuerdo con las plataformas de hardware y software elegidas para el desarrollo.
Particular Service Level – PSL	▪ Encapsula todos los programas que implementan los requerimientos particulares para un Supervisor específico: rutinas de asistencia de alarmas, manejadores de eventos, rutinas de respuesta a comandos de usuarios, etc. ▪ La implementación cambia para cada Supervisor de acuerdo con las características de servicios particulares.
Basic Service Level – BSL	▪ Encapsula todos los programas que implementan los servicios de supervisión básicos que son comunes a todos los Supervisores: imagen de los *puntos de supervisión*, rutinas de detección de alarmas, almacenamiento histórico, transmisión de mensajes, ejecución de comandos de usuario, planificación de procesos de alto nivel, modelado de procesos del mundo real, etc. ▪ **Desde un punto de vista funcional, este nivel de servicio es el mismo para todo Supervisor.**

A su vez, se pueden definir 5 categorías de sistemas SCADA, teniendo en cuenta su facilidad de uso, flexibilidad del software y su interface con el programador y usuario.

En la siguiente tabla, se describen las categorías con sus principales características, ordenadas de acuerdo al grado de conocimiento que se requiere para su operación, comenzando por las categorías orientadas a usuarios finales hasta las que son más orientadas al programador:

Categorías	Características
Software específico	▪ Está orientado a usuarios finales. ▪ De fácil uso y necesita poca preparación inicial. ▪ Diseñado especialmente para controlar una aplicación en particular. ▪ Si se requiere una nueva funcionalidad, el usuario deberá contactar al proveedor del software y solicitar que se agregue esta nueva facilidad al paquete original.
Facilidades agregadas	▪ Son funciones incluidas en algún entorno de desarrollo conocido (por ejemplo: Planillas de cálculo). ▪ Permiten direccionar los ports de comunicaciones y obtener los datos del entorno, para luego realizar las operaciones necesarias y visualizarlas en forma gráfica.
Instrumentación Virtual	▪ Permite a los usuarios diseñar los instrumentos más apropiados para su aplicación. ▪ Usa el hardware de adquisición de datos disponible de la PC. ▪ Realiza las interfaces entre ambos, utilizando los estándares de arquitecturas abiertas para procesamiento, almacenamiento en memoria y despliegue de la información. ▪ El usuario define la funcionalidad de los instrumentos virtuales, pudiendo agregarle formas elaboradas de control.
Interfaz con un lenguaje	▪ Está orientado a programadores. ▪ Es una colección de subrutinas o llamadas a funciones desde lenguajes de programación convencionales tales como C/C++, Visual Basic, Pascal. ▪ Los programadores deberán escribir, compilar código y realizar el link con la interfaz provista para poder ejecutar las tareas de adquisición de datos. ▪ Provee el acceso al hardware de adquisición de datos a través de simples llamadas a funciones. ▪ Luego de la recolección y del almacenamiento de datos, los programadores deberán realizar el resto de la programación para el manejo y presentación de los mismos o utilizar alguna herramienta de análisis y visualización.
Código fuente	▪ Es la forma más compleja de programar un Supervisor y la más lenta para obtener un sistema funcionando. ▪ Ayuda a reducir el tamaño del software final. ▪ Son archivos usualmente escritos en C, que los programadores pueden compilar con la aplicación para adquirir y controlar datos. ▪ Programación del hardware a nivel de registro.

En la siguiente figura se describe gráficamente los diferentes niveles de servicio y sus interfaces con el entorno:

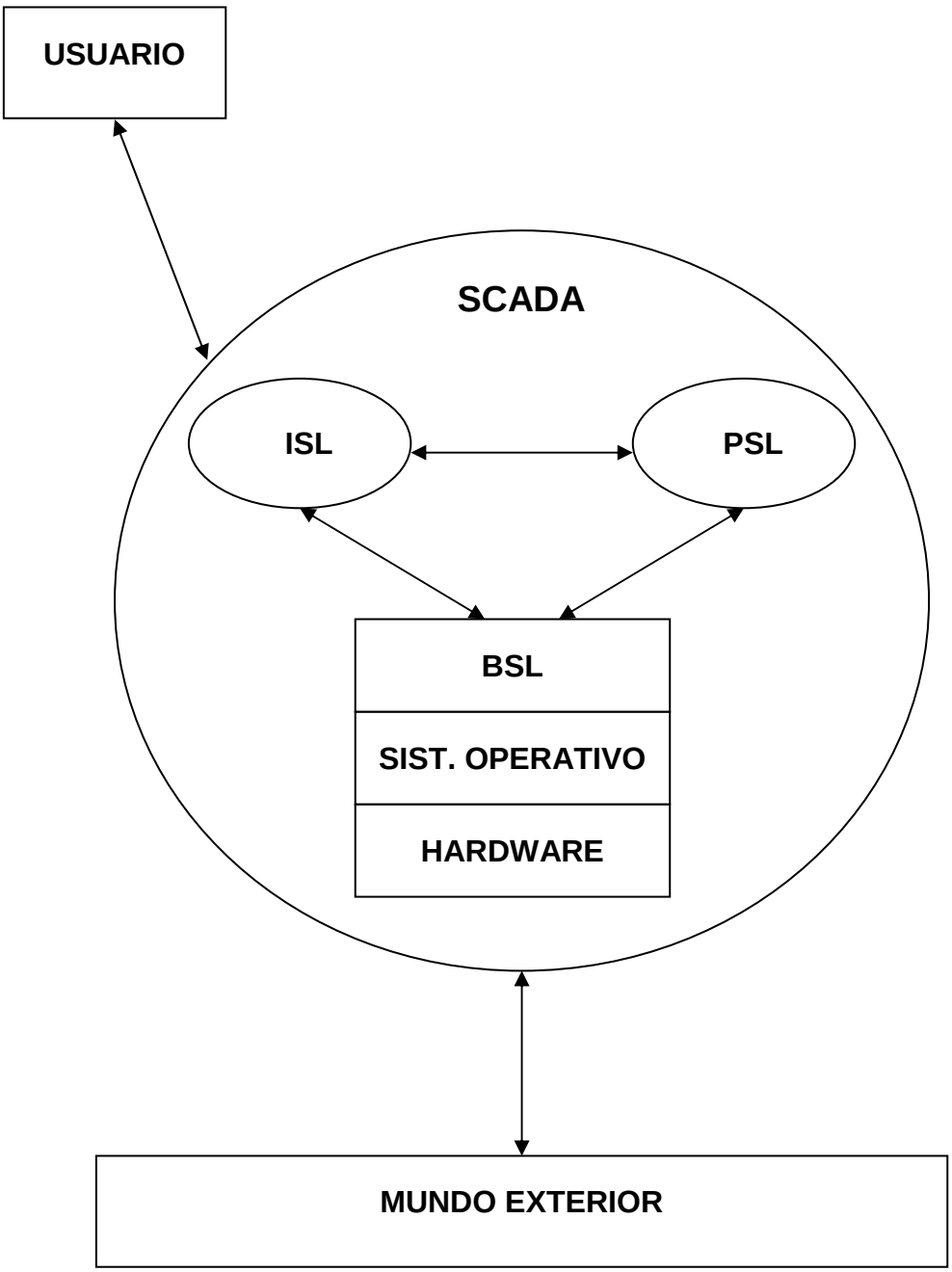


Figura 3 - Los niveles de servicio y sus interfaces.

3 - SOFTWARE DE BASE PARA TIEMPO REAL

En este capítulo se analizará el software de base para Sistemas de Tiempo Real, sus requerimientos y características. Por otra parte, también se estudiará la plataforma Win32, su familia de Sistemas Operativos y sus servicios para esta clase de sistemas.

3.1 Requerimientos de Sistemas Operativos para Tiempo Real.

La función principal de los sistemas operativos en general, es la de proveer una "máquina virtual", la cual ejecuta el hardware existente y provee todos los servicios básicos para la programación de aplicaciones, ocultando al programador los detalles de implementación cercanos al hardware.

La segunda función primordial de un sistema operativo está relacionada con la administración de los recursos de un sistema de computación. El principal objetivo en sistemas operativos tradicionales es el de maximizar la utilización de los recursos disponibles y optimizar la ejecución de las distintas tareas que ejecutan en el sistema. Este objetivo dejan de ser un requerimiento para los Sistemas Operativos para Tiempo Real, ya que en lugar de optimizar el uso de los recursos, el requerimiento fundamental para este tipo de sistemas es el de cumplir a tiempo con las *metas* de todas las tareas.

En la actualidad existe una gran variedad de sistemas operativos usados para Tiempo Real, pero éstos suelen ser de propósito general y es muy difícil que los mismos puedan resolver adecuadamente las restricciones de tiempo de las tareas, ya que nativamente no proveen soporte para hacerlo.

Los Sistemas Operativos para Tiempo Real deben facilitar el desarrollo de las aplicaciones con restricciones de tiempo, mediante primitivas y servicios específicos para ese fin: semáforos con tiempo, monitores con tiempo, datagramas de tiempo real, circuitos virtuales de tiempo real y transacciones de tiempo real.

Actualmente, estos sistemas suelen incluir un conjunto mínimo de características y facilidades básicas, las cuales se detallan a continuación [Wai97]:

- Desalojo del procesador (**preemptive***)*: realiza el cambio de contexto (**context switching**) al llegar una tarea con mayor prioridad que la tarea que está ejecutando en ese momento.

- Planificación de tareas por prioridades: asigna el procesador a la tarea que tiene mayor prioridad.

- Multitarea con servicios de comunicación y mecanismos de sincronización: facilita el desarrollo de aplicaciones concurrentes.

- *Cambio de contexto* veloz: minimiza la inhibición de interrupciones que ocurren durante el mismo.

- Soporte de reloj de Tiempo Real: administra y controla el tiempo de ejecución del sistema y de las tareas.

- Primitivas para control de ejecución de tareas: funciones para detener, reanudar y/o retrasar las tareas durante una cantidad fija de tiempo.

- Interrupciones externas: respuesta veloz para este tipo de eventos.

- Núcleo pequeño (*microkernels*): mejora la velocidad de atención de interrupciones, minimiza su funcionalidad y facilita la construcción de Sistemas Distribuidos.

Todo Sistema Operativo para Tiempo Real deberá proveer, como mínimo, funciones de planificación (**scheduling**), despacho, comunicación y sincronización entre tareas. Se llama núcleo (**kernel**) a este conjunto de rutinas del sistema operativo. El planificador (**scheduler**) determina qué tarea debe ser asignada al procesador en cada ranura de tiempo (**time-slice**); el despachador (**dispatcher**) es el encargado de realizar todas las operaciones necesarias para que la tarea elegida comience su ejecución.

En sistemas pequeños, como pueden ser los sistemas embebidos, las funciones del *núcleo* abarcan casi todo el sistema. En otros sistemas más complejos, éstas son funciones mínimas que deberá proveer el sistema operativo, conjuntamente con funciones de administración de memoria y periféricos, protección entre tareas, control de acceso no autorizado al sistema, drivers de dispositivos y Sistemas de Archivos.

Para facilitar el desarrollo de aplicaciones de Tiempo Real, un sistema operativo deberá ser predecible, proveer adaptabilidad, tener capacidad de manejar aplicaciones complejas y, en algunos casos, proveer facilidades para la distribución en múltiples procesadores con un alto grado de cooperación. A su vez, las tareas suelen tener diferentes tipos de restricciones, como pueden ser: de recursos, precedencia, concurrencia, comunicación, ubicación y criticidad. El sistema operativo también deberá proveer facilidades para satisfacer estas restricciones.

3.2 Núcleos.

En esta sección se describen los diferentes tipos de *núcleos* utilizados en la actualidad: Monotarea, Basados en Corrutinas, Basados en Interrupciones y Sistemas

Operativos Multitarea. En la siguiente tabla se describen los tres primeros de ellos con sus principales características y en la siguiente sección se analizará en profundidad los *núcleos* de los Sistemas Operativos Multitarea:

Núcleos	Características
Monotorea	▪ Son los más simples. ▪ Ejecutan una tarea a la vez. ▪ Tienen un cargador del programa en memoria. ▪ No existe planificación ni comunicación de tareas. ▪ Se utilizan para aplicaciones embebidas simples, las cuales generalmente ejecutan en un ciclo infinito.
Basados en Corrutinas	▪ Ejecutan múltiples tareas en forma concurrente, mediante llamadas explícitas entre sí. Estas tareas se llaman *corrutinas*. ▪ La aplicación se divide en dos o más tareas, codificadas como corrutinas, donde cada una ejecuta una fase del problema a resolver. ▪ La sincronización entre corrutinas se realiza mediante las llamadas explícitas entre las mismas. ▪ La comunicación entre corrutinas se hace mediante variables globales. ▪ Ventajas: Puede ser muy sencilla la verificación de las restricciones de tiempo, si cada corrutina invoca a la siguiente a intervalos conocidos. ▪ Desventajas: no siempre la aplicación pueda ser diseñada bajo este concepto y también requiere estricta disciplina por parte de los programadores para que cada corrutina libere el procesador en intervalos regulares.
Basados en Interrupciones	▪ Ejecutan múltiples tareas en forma concurrente, donde cada tarea es una rutina de atención de interrupción. ▪ El mecanismo de prioridades de las interrupciones permite establecer una planificación y un orden de ejecución de las tareas. ▪ Se pueden planificar las tareas en forma periódica, ante la ocurrencia de interrupciones de reloj. ▪ Una extensión a este clase de *núcleos*, se los conoce como Monitores Foreground/Background. ▪ Ventajas: son simples de escribir, tienen bajo overhead y el proceso de planificación se hace por hardware. Son muy usados en sistemas embebidos. ▪ Desventajas: Los sistemas desarrollados de esta forma son difíciles de verificar y validar, debido al flujo asincrónico de las interrupciones. Esto implica que la validación de las restricciones de tiempo es más compleja. No poseen servicios avanzados, como manejo de archivos, drivers de dispositivos, etc.

3.2.1 Sistemas Operativos Multitarea.

Los *núcleos* de estos sistemas operativos ejecutan múltiples tareas, de forma tal que cada una pareciera tener todo el sistema a su disposición. Para ello, el sistema operativo deberá administrar el uso y asignación de los recursos entre todas ellas.

Estas funciones son complejas en un Sistema Operativo de Tiempo Real, ya que varias tareas son dependientes del tiempo y, por lo tanto, deben tener mayor prioridad que otras. Para ello, el sistema operativo implementa un algoritmo de planificación de tareas, el cual asigna las prioridades basado en las restricciones de tiempo de cada una de ellas.

A su vez, las múltiples tareas ejecutando en un mismo sistema, deben poder comunicarse entre sí y compartir información, a través de mecanismos de sincronización para el acceso concurrente a los recursos compartidos. Además, esta clase de sistemas operativos deben proteger los datos privados y el entorno de cada tarea, para evitar que un programa corrompa a otro, ya sea deliberadamente o por error.

En resumen, un Sistema Operativo Multitarea para Tiempo Real, deberá realizar las siguientes actividades y proveer los servicios que se detallan a continuación:

- Planificar las tareas de acuerdo a su algoritmo de planificación y al nivel de prioridad de las mismas.
- Administrar la memoria del sistema y permitir que las aplicaciones de Tiempo Real más críticas puedan mantenerse cargadas en memoria física.
- Manejar las interrupciones del sistema y, en lo posible, tratar de minimizar el tiempo de atención de las mismas, para poder predecir su impacto sobre las tareas de Tiempo Real.
- Cumplir con los requerimientos de tiempo de las tareas.
- Compartir información y proveer comunicación entre las tareas.
- Administrar el acceso a los dispositivos de E/S.
- Proveer un Sistema de Archivos.
- Administrar todos los recursos del sistema.

En la actualidad existen diferentes sistemas operativos utilizados para el desarrollo de aplicaciones de Tiempo Real. De acuerdo a su diseño e implementación particular, se pueden distinguir tres clases básicas de sistemas:

- *Núcleos* propietarios: garantizan la ejecución predecible, sin provisión de servicios extendidos. Suelen ser implementados en sistemas embebidos.

- Extensiones a Sistemas Operativos de propósito general: se construye una capa de servicios, funciones y facilidades de Tiempo Real, sobre el *núcleo* del Sistema Operativo.

- Sistemas Operativos de Investigación: proveen soluciones completas para el desarrollo de Sistemas de Tiempo Real.

3.2.1.1 Administración de Tareas.

En un sistema monoprocesador, sólo una tarea puede estar ejecutando en un instante dado. Para permitir la ejecución de múltiples tareas en forma concurrente, se utiliza una solución conocida con el nombre de Tareas Secuenciales.

En este modelo, cada hilo (**thread**) de ejecución de un programa recibe el nombre de Tarea. La idea básica del modelo se basa en que, cada tarea que ingresa al sistema, pasa por diferentes estados de ejecución independientes. Existen diversas acciones que causan las transiciones entre esos estados, las cuales pueden ser provocadas por el sistema operativo o por los programas en ejecución: solicitudes de servicio al sistema operativo, fin de E/S, llegada de mensajes, recursos que se liberan u ocupan, etc.

La base fundamental para maximizar la concurrencia, es el aprovechamiento de los procesadores de E/S, los cuales ejecutan independiente y asincrónicamente con el procesador central. Esto significa que cuando una tarea solicita E/S, el sistema operativo realiza las siguientes acciones:

1. Suspende la ejecución de la tarea hasta la finalización de la E/S.

2. Ejecuta las instrucciones necesarias para que los procesadores especializados inicien la E/S solicitada por la tarea.

3. Selecciona otra tarea para que comience su ejecución en el procesador central.

4. A partir de instante, los procesadores de E/S y el procesador central, ejecutan sus instrucciones en forma paralela.

Para poder realizar el *cambio de contexto* entre las tareas, el sistema operativo almacena información sobre el estado de ejecución de cada una de ellas: identificador, prioridad, estado actual y los valores de los registros del procesador.

Las tareas deben ser cooperativas, para lo cual el Administrador de Tareas provee diversos mecanismos de comunicación y sincronización. Para controlar el acceso simultáneo a los recursos compartidos, el sistema mantiene tablas de los mismos, conjuntamente con el estado de las solicitudes de cada tarea.

Con toda esta información, el Administrador de Tareas realiza su función principal: repartir el tiempo del procesador entre todas las tareas del sistema. Como fue mencionado en la sección 3.1, esta función es realizada por dos rutinas: el *planificador* y el *despachador* de tareas.

El *despachador* es el encargado de intercambiar las tareas y de preservar el entorno actual. Esta rutina es ejecutada cuando ocurre alguna de las siguientes condiciones de entrada:

- Ocurre una interrupción en el sistema.
- Una tarea solicita un servicio al sistema operativo, como ser: un pedido de E/S, una instrucción de espera por algún evento o una demora hasta que transcurra un tiempo determinado.
- Una tarea finaliza su ejecución.

En todos estos casos, el *despachador* realiza un *cambio de contexto*.

El *planificador* toma las decisiones de asignación de alto nivel y es activado cuando una tarea es suspendida. En ese momento selecciona, de todas las tareas que están listas para ejecutar, la de mayor prioridad.

Hay diversas técnicas de planificación para Sistemas Operativos de propósito general y de Tiempo Real. En la siguiente sección se realizará una Introducción a la Planificación y en el próximo capítulo se estudiarán diferentes algoritmos y políticas de planificación para ambas clases de sistemas operativos.

3.2.1.2 Introducción a la Planificación.

La teoría de planificación no sólo se restringe a los Sistemas de Tiempo Real, sino que comprende también el estudio de diversos sistemas: de fabricación, transporte, control de tareas, etc. Sin embargo, cabe destacar que los problemas de planificación de Tiempo Real son diferentes a los tratados en otras áreas de investigación operativa.

El problema de planificación de tareas (**scheduling**) en Tiempo Real es tan complejo que las soluciones tradicionales no son útiles.

En general, la teoría de planificación para Sistemas de Tiempo Real se relaciona con el hecho de hacer cumplir las restricciones de tiempo de las tareas que ejecutan en el sistema. El objetivo es encontrar planes óptimos estáticos que minimicen el tiempo de respuesta para un conjunto dado de tareas. En esta clase de sistemas, no se desea optimizar la asignación de recursos, sino hacerla predecible.

El objetivo principal de un *planificador* de tareas es predecir la ejecución de las mismas y cumplir con todas sus restricciones de tiempo y recursos. Para ello, deberá establecer un plan de ejecución, el cual especifica un orden para ejecutar las tareas. El *planificador* deberá determinar si existe tal plan y si es posible de encontrar.

Una tarea, desde el punto de vista del *planificador*, es una entidad (**thread**) a planificar, un módulo que se invoca para hacer una función. Se dice que una tarea es planificable si existe un plan con el cual se puedan cumplir sus restricciones de tiempo. Se dice que un algoritmo garantiza a una nueva tarea, si puede encontrar un plan para garantizar todas las tareas anteriores y también cumplir con la *meta* de la nueva tarea.

Es necesario que un algoritmo de planificación pueda asegurar los requerimientos analizados en la sección 1.1: predictibilidad, planificabilidad y estabilidad, entre otros.

La predictibilidad de un Sistema de Tiempo Real sólo puede garantizarse si se conocen las condiciones de peor caso de carga antes de su ejecución. Por otro lado, interesa el grado de planificabilidad del sistema. Dado que los cálculos se basan en los peores casos de ejecución, si todas las tareas cumplieran sus *metas* aún en situaciones de sobrecarga, el sistema resultará altamente subutilizado.

Además, los Sistemas de Tiempo Real deben ejecutar eficiente y correctamente ante situaciones de sobrecarga. En estos casos, la solución recae en políticas de planificación estables que garanticen que las tareas más críticas siempre cumplan sus *metas*. Sin embargo, puede no haber correlación entre las *metas* de las tareas y su criticidad. Planificar tareas de forma tal de maximizar el número de tareas críticas que cumplen sus *metas*, es un problema no trivial. Ante tales situaciones, hay que asegurar que un subconjunto crítico de las tareas del sistema cumplirán con sus *metas*.

Un algoritmo de planificación deberá considerar todos estos factores, así como también, las restricciones de recursos, precedencia, concurrencia, comunicación, criticidad y otras más. Para ello se utilizan distintas políticas, técnicas y algoritmos de planificación de tareas, tanto en Sistemas Operativos tradicionales como en Sistemas Operativos específicos para Tiempo Real, los cuales serán estudiados en el capítulo 4.

3.2.1.3 Cooperación y Comunicación entre Tareas.

Los Sistemas de Tiempo Real suelen construirse como un conjunto de tareas concurrentes que cooperan y se comunican entre sí.

Las tareas deben satisfacer un propósito común y para lograrlo, deben poder comunicarse entre sí. También pueden competir por los recursos del sistema y esta competencia debe estar regulada por la intervención del sistema operativo o por la intercomunicación de las tareas, a través de algún mecanismo de sincronización entre las mismas.

En esta sección se analizan diversas formas para solucionar este tipo de problemas. Para ello, se describirán los conceptos de la programación concurrente y se realizará una introducción a las diferentes técnicas de comunicación y sincronización de tareas.

Uno de los elementos básicos de la programación concurrente es el concepto de Tarea Secuencial, como fue descripto en la sección 3.2.1.1. La ejecución de un programa dividido en tareas concurrentes puede verse como una secuencia de acciones atómicas, cada una de las cuales es resultante de la ejecución de una operación indivisible.

Si las variables de una tarea no pueden ser afectadas por otras, entonces la tarea puede verse como una única función que termina en un tiempo finito y de forma determinística. En este caso, se dice que las tareas son independientes.

Esta situación cambia de forma considerable si una tarea puede modificar las variables de otra, ya que los resultados obtenidos dependerán de la velocidad relativa entre las tareas. En estos casos, el acceso concurrente a variables compartidas produce interferencias entre las ejecuciones de cada tarea. Para evitar cualquier comportamiento erróneo o inconsistencia en los datos, se deberá sincronizar la ejecución para restringir ciertas intercalaciones en sus acciones.

Para ello, una aplicación subdividida en tareas concurrentes deberá utilizar mecanismos de comunicación y sincronización, que permita una cooperación efectiva entre las mismas.

Los mecanismos de comunicación permiten que una tarea influya en la ejecución de la otra, mientras que los mecanismos de sincronización son un conjunto de restricciones en el orden de los eventos. El programador puede usar estos mecanismos para retardar la ejecución de una tarea de modo de satisfacer tales restricciones [Wai97].

La comunicación y sincronización pueden basarse en el mecanismo de memoria compartida o en el pasaje de mensajes. En forma introductoria, a continuación se describirán cada uno de ellos, con sus ventajas y desventajas.

3.2.1.3.1 Memoria Compartida.

En este mecanismo, las tareas intercambian información y se sincronizan por medio de lecturas y escrituras de un conjunto de variables compartidas entre todas las tareas.

Los problemas de esta técnica pueden provocar inconsistencia en los datos compartidos entre las tareas. A modo de ejemplo, una tarea comienza a leer las variables del ambiente y, antes de terminar de leer todo el bloque de entrada, es interrumpida. Durante esta interrupción, otra tarea comienza su ejecución y modifica todo el bloque de entrada de datos, ya que es compartido entre todas las tareas. Cuando la tarea reanuda su ejecución, termina de leer las variables pendientes y realiza los cálculos correspondientes, los cuales producirán resultados incorrectos.

Estos problemas se solucionan con técnicas de **exclusión mutua**, creando una sección crítica, es decir, una secuencia de comandos que debe ser ejecutada como una operación indivisible. El término exclusión mutua se refiere a la ejecución mutuamente excluyente de cada región crítica.

Otra técnica empleada en el mecanismo de memoria compartida es la denominada bloqueo (**locking**). Cuando una tarea desea acceder a un área compartida, debe solicitar permiso al sistema operativo, por medio de primitivas específicas. Luego, cuando la tarea termina de utilizar las variables compartidas, debe informar al sistema que el área compartida está nuevamente disponible para su uso. Un problema grave de confiabilidad que surge al utilizar esta técnica, ocurre cuando un conjunto de tareas forman un ciclo esperando que cada una se desbloquee para poder seguir ejecutando. Este estado es conocido como abrazo mortal (**deadlock**).

Los problemas de acceso concurrente a memoria compartida, podrían ser solucionados por medio de la deshabilitación de interrupciones durante el acceso a la misma. En esta técnica, si bien es muy simple, tiene varias desventajas. En primer lugar, podrían ocurrir serios problemas de estabilidad y confiabilidad en el sistema, si la tarea entrara en un ciclo infinito debido a un error de programación. Otra desventaja es que podría ocurrir inanición (**starvation**), es decir, que las tareas nunca lleguen a ejecutar por más que dispongan de todos los recursos para hacerlo.

Existen diversas técnicas para implementar el mecanismo de memoria compartida, cada uno con sus ventajas y desventaja: secciones críticas, bloqueo, semáforos, monitores, etc. En general, en estas soluciones suelen ser aceptables para tareas concurrentes cooperativas que no se encuentran en un ambiente de Tiempo

Real. Pero en un sistema en el cual existen restricciones de tiempo y criticidad para las tareas, aparecen otros problemas que deberán ser considerados.

Uno de estos problemas es la inversión de prioridades de las tareas. Este fenómeno ocurre cuando una tarea de prioridad alta está bloqueada esperando el acceso a un recurso que está siendo utilizado por una tarea de menor prioridad. En este caso, pueden existir otras tareas de mayor prioridad que consumen todo el tiempo del procesador y la tarea que está bloqueada no puede continuar su ejecución debido a que la tarea de menor prioridad nunca llegará a ejecutar. Esto puede provocar que ciertas tareas críticas no cumplan con sus *metas*.

Si bien en algunas aplicaciones de Tiempo Real, como en los sistemas SCADA, las *metas* suelen ser firmes o blandas, la acumulación de demoras en las ejecuciones de cada tarea puede provocar retrasos en su frecuencia de activación. Para evitar estos problemas, hay que analizar detalladamente las restricciones de tiempo, relaciones de precedencia y prioridades de todas las tareas.

3.2.1.3.2 Pasaje de mensajes.

Estos mecanismos tratan de lograr comunicación y sincronización entre las tareas a través de intercambio de mensajes y no por medio de lecturas y escrituras de variables compartidas.

La comunicación se da porque una tarea recibe un mensaje enviado por otra y la sincronización se da porque un mensaje sólo puede ser recibido luego de haber sido enviado, lo cual restringe el orden en el cual deben ocurrir los eventos.

Diversos autores analizan las primitivas usadas para el pasaje de mensajes entre tareas, desde los puntos de vista sintáctico y semántico. Se consideran los siguientes cuatro aspectos diferentes:

- Tipos de sincronización.
- Nominación de los canales de comunicación.
- Tipos de mensajes.
- Tratamiento y recuperación de errores.

Los tipos de sincronización entre tareas, se pueden dividir en 2 categorías: comunicación sincrónica y comunicación asincrónica. En el siguiente cuadro, se describen ambas categorías con sus principales características:

Tipo de sincronización	Características
Comunicación Sincrónica	▪ Las primitivas de esta categoría son **bloqueantes**. ▪ La primitiva *send* bloqueará a la tarea emisora hasta que el mensaje sea recibido. La primitiva *receive* también bloqueará a la tarea receptora, hasta que llegue un mensaje. En este caso, la primitiva *send* sincronizará tanto al emisor como al receptor y transferirá la información entre ellos. ▪ Las primitivas de esta categoría se pueden dividir en 3 clases: ▪ Rendez-vouz: las primitivas send y receive son bloqueantes. A este tipo de comunicación también se lo denomina simétrica. Este tipo de primitivas permite calcular de forma simple las restricciones de tiempo. ▪ Rendez-vouz extendido: es una comunicación asimétrica, en la cual la tarea emisora se conecta con la tarea receptora. En este caso, cada tarea tiene un rol específico: la tarea emisora (**client**) realiza un pedido y la tarea receptora (**server**) ofrece un servicio. Al recibir un mensaje, el *servidor* ejecuta un grupo de sentencias sobre el pedido del *cliente* y luego le envía la respuesta. El *cliente* quedará bloqueado hasta que el *servidor* le haya provisto el servicio solicitado. Es una solución para la mayoría de los problemas de multiprogramación. Ofrece un buen desempeño para los Sistemas de Tiempo Real, aunque es crítico la demora en el *servidor*. ▪ Llamada a Procedimientos Remotos (RPC, Remote Procedure Call): es una variante al Rendez-vouz extendido, en el cual la tarea emisora no queda bloqueada. Los procedimientos invocados ejecutan con exclusión mutua, en forma intercalada y no determinística con la tarea que los invocó. En este caso, aumenta el grado de concurrencia pero disminuye la predictibilidad.
Comunicación asincrónica	▪ Las primitivas de esta categoría son **no-bloqueantes**. ▪ Cuando una tarea realiza un *send*, continuará inmediatamente su ejecución, aunque la tarea receptora todavía no haya recibido el mensaje. De la misma forma, cuando una tarea realiza un *receive*, continuará su ejecución aunque todavía no haya ningún mensaje disponible. ▪ La ventaja principal de esta técnica es que maximiza el paralelismo en la ejecución de las tareas. ▪ Se deberán mantener colas para almacenar los mensajes. ▪ La predictibilidad de las tareas se complica, debido a que se desconoce el momento en que se recibirán los mensajes.

Con respecto a la nominación de los canales de comunicación, existen diversas formas de dar nombre o de especificar a los mismos.

El esquema más simple es el llamado *nominación directa*. En este mecanismo, las primitivas emplean el nombre de las tareas para designar el origen y el destino del mensaje. Esta técnica es útil en configuraciones en las cuales la salida de una tarea es la entrada de otra.

Otro esquema para las comunicaciones de tipo *cliente/servidor*, es el uso de casillas de correo (**mailboxes**). Una *casilla de correo* es un nombre global, el cual puede aparecer tanto en primitivas *send* como en primitivas *receive.* Los mensajes enviados a una casilla pueden ser recibidos por cualquier tarea que acceda para leerlos. Este mecanismo es simple de implementar cuando las tareas comparten una memoria común, pero no lo es en el caso de Sistemas Distribuidos.

Un tipo especial de *casilla de correo*, es el de **port**. Los *ports* se utilizan para crear conexiones dinámicas entre el *cliente* y el *servidor*. La secuencia de ejecución es la siguiente: primero, el *cliente* solicita un servicio al *servidor*, especificando un número de *port*; luego el *servidor* acepta la conexión; y por último se crea dinámicamente un nuevo *port*, estableciendo un nuevo canal de comunicación entre ambos.

En los Sistemas de Tiempo Real, cuando se establece un canal de comunicación entre dos tareas usando *nominación directa*, se pueden predecir los tiempos de ejecución y analizar las relaciones de precedencia, si se conocen los tiempos de transmisión. En cambio, si se utilizan *casillas de correo*, pueden ocurrir problemas relacionados con la predictibilidad, dado que un mensaje puede permanecer en la casilla por un tiempo indeterminado.

El tercer aspecto semántico relacionado con el pasaje de mensajes, está asociado con el tipo de los mismos utilizados en la comunicación. Los mensajes pueden ser de tamaño fijo o variable. En el caso de Sistemas de Tiempo Real, la mayoría de los mensajes son relativamente cortos y de tamaño fijo, conteniendo los valores de las variables del entorno o informaciones de estado.

Por último, debe ser considerado el tratamiento y recuperación de errores en la comunicación entre tareas. En este caso, es crítico el uso de las primitivas sincrónicas, ya que una falla puede provocar altos riesgos en la estabilidad y confiabilidad del sistema.

En los Sistemas de Tiempo Real, la forma más común para manejar este tipo de fallas, es el uso de primitivas de espera a la ocurrencia de un evento, asociadas a un contador de tiempo. Con la utilización de estos servicios se evita el problema de tener *abrazo mortal*, dado que en el peor caso, la tarea continuará ejecutando luego que expire el contador respectivo.

3.2.1.3.3 Estimulación cruzada.

Este mecanismo de sincronización facilita el control entre las tareas, como pueden ser las funciones para comenzar, demorar o parar la ejecución de otras o de sí mismas. Para realizar estas operaciones se utilizan dos clases de primitivas:

- Wait (*evento*): la tarea que ejecuta esta primitiva, suspende su actividad hasta que reciba una señal de que ha ocurrido ese *evento*.

- Signal (*evento*): la tarea que ejecuta esta primitiva, indica que ha ocurrido ese *evento*, habilitando a las tareas en espera para que puedan continuar su ejecución.

El mecanismo de estimulación cruzada puede verse como una operación de manejo de semáforos binarios con sus dos estados: *signaled* y *non-signaled*. En esta técnica no es necesario que exista memoria compartida y puede verse como un pasaje de un mensaje sin información, que sólo se utiliza para hacer sincronización entre tareas.

En los Sistemas de Tiempo Real, el mecanismo de memoria compartida es, en general, más simple y eficiente de implementar que el pasaje de mensajes. Sin embargo, este último es la única alternativa cuando se controla un sistema con varias computadoras conectadas a través de una red de comunicaciones. El mecanismo de estimulación cruzada, puede combinarse con ambos esquemas con facilidad.

3.3 La plataforma Win32 para Tiempo Real.

En esta sección se analiza la plataforma Win32 de Microsoft® y su familia de Sistemas Operativos, enfocada principalmente a sus características para Sistemas de Tiempo Real.

La plataforma Win32 fue la elegida para desarrollar la herramienta de esta Tesis, debido a sus características para Tiempo Real, a su popularidad y a su presencia en el mercado y en la industria.

Esta plataforma está orientada a Sistemas Operativos de propósito general, pero cumple con varios de los requerimientos principales de los Sistemas Operativos para Tiempo Real:

- Planificación de tareas con desalojo del procesador (*preemptive*) y por prioridades.

- Multitarea con servicios de comunicación y mecanismos de sincronización.
- Primitivas para control de ejecución de tareas.
- Soporte de reloj de Tiempo Real.
- Atención de interrupciones por prioridades y con desalojo (*preemptive*).
- Núcleo pequeño (*microkernel*).

3.3.1 Los Sistemas Operativos y la Win32 API.

La plataforma Win32 soporta varios Sistemas Operativos, cada uno de ellos con sus similitudes y diferencias entre sí.

Esta familia de Sistemas Operativos, está compuesta por diferentes sistemas Microsoft® Windows, todos ellos basados en la arquitectura de 32-bit. De acuerdo a sus características y servicios, se pueden clasificar en 3 tipos de tecnologías:

- Windows2000 (Server & Professional) / WindowsNT (Server & Workstation).
- Windows98 / Windows95.
- Windows, versión 3.1, con el soporte de Win32s.

Todos ellos conforman una interface consistente y uniforme, llamada **Win32 API (Application Program Interface)**. A través de ella, se accede a todos los servicios ofrecidos por los diferentes Sistemas Operativos. Esta capa de acceso está formada por una colección de funciones, estructuras de datos, mensajes, macros e interfaces.

Al utilizar esta interface, se pueden desarrollar aplicaciones para que se ejecuten en todos sus Sistemas Operativos y, a su vez, aprovechar las facilidades únicas de cada uno de ellos.

Las diferencias de implementación en cada uno de los componentes, dependen de las características subyacentes de cada Sistema Operativo. Las mayores diferencias se encuentran en las tecnologías más poderosas, como lo son Windows2000 y WindowsNT.

En el siguiente cuadro se describen las principales similitudes y diferencias entre las dos primeras clases de Sistemas Operativos. El sistema Windows versión 3.1 con la integración de Win32s, no se estudiará debido a sus características y a su obsolecencia en el mercado.

Característica	Windows2000 / NT	Windows98 / 95
Arquitectura:		
Multiprocesadores.	Sí.	No.
Soporte para otras arquitecturas, no basadas en Intel®.	Sí.	No.
Administración de tareas:		
Kernel de 32-bit.	Sí.	Código de 32-bit y 16-bit para compatibilidad con Windows 3.1 y con DOS.
Multitarea y Multithreading.	Sí.	Sí.
Planificación preemptive y por prioridades.	Sí.	Sí, pero las tareas de 16-bit, se planifican en forma cooperativa.
Servicios de comunicación y mecanismos de sincronización entre threads.	Sí.	Sí.
Administración de memoria:		
Modelo de memoria flat 32-bit.	Sí.	Sí.
Espacio de direccionamiento separado por cada proceso.	Sí.	Sí, pero las tareas de 16-bit, comparten un espacio de memoria común.
***Interface Gráfica**:*		
Windows y controles standard.	Sí.	Sí.
DirectX 7.x.	Sólo en Windows2000.	Sí.
Redes:		
Sockets.	Sí.	Sí.
Servicios de Servidor y Routing.	Sí.	No.
Clustering.	Sí.	No.
Seguridad:		
Seguridad Certificada C2.	Sí.	No.
Logging de eventos del Sistema.	Sí.	No.
Misceláneos:		
Plug & Play.	Sólo en Windows2000.	Sí.
Administrador de Servicios.	Sí.	No.
Soporte para Unicode.	Sí.	No.

3.3.2 Arquitectura de WindowsNT.

Dada la variedad de capas, módulos y componentes en cada Sistema Operativo de la familia Win32, en esta sección en particular se estudia la arquitectura de uno de ellos: WindowsNT. La arquitectura de este sistema está dividida en dos secciones principales: *kernel mode* y *user mode* [Msr00].

- *Kernel mode*, es el modo más privilegiado de operación del sistema, en el cual tiene acceso a todo el hardware, incluyendo la memoria y todo el espacio de direccionamiento de todos los procesos. Esta parte se llama *WindowsNT Executive* e incluye: *HAL* (Hardware Abstraction Layer), el *Microkernel* y los *Executive Services*.

- *User mode*, es el modo de operación menos privilegiado del sistema, el cual no tiene acceso al hardware y sólo puede acceder dentro de su propio espacio de direccionamiento. Para acceder al hardware y a los servicios del sistema, invoca las funciones de la Win32 API.

A continuación se muestra un diagrama de su arquitectura y los módulos que la componen:

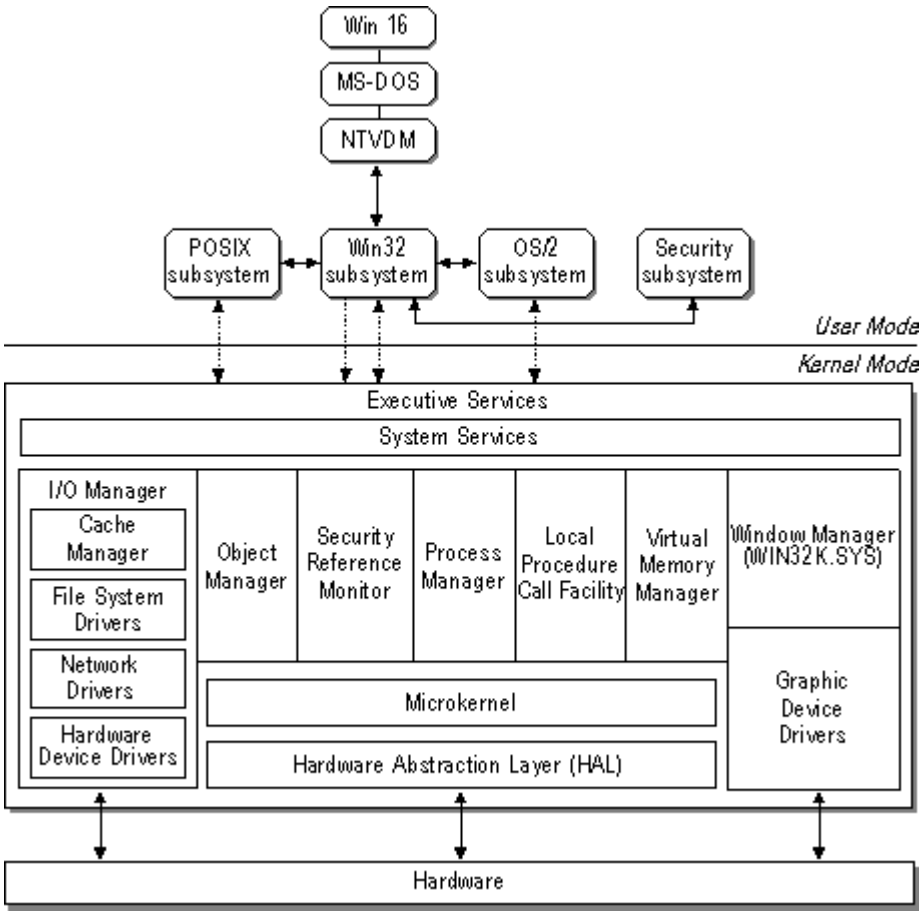


Figura 4 - Arquitectura modular de WindowsNT.

La *HAL* (Hardware Abstraction Layer) es una biblioteca de rutinas que ejecutan en *kernel-mode*, las cuales acceden y manipulan el hardware. Esta capa se encuentra en el nivel más bajo del WindowsNT Executive, entre el hardware y el resto del Sistema Operativo. Está compuesta por funciones desarrolladas por Microsoft® o por proveedores de hardware.

Esta capa de software oculta y abstrae las características de la plataforma subyacente y provee puntos de acceso común para todas las arquitecturas, de forma tal que el Sistema Operativo pueda ejecutar en diferentes plataformas con diferentes procesadores.

Las rutinas de *HAL*, son llamadas desde los drivers de dispositivos de más alto nivel y, sobre todo, del *Microkernel*, el corazón de WindowsNT.

El *Microkernel* planifica y despacha los threads, maneja las interrupciones y las excepciones del sistema. Para ello, ejecuta la política de planificación implementada por el *WindowsNT Executive*. También, realiza los *cambios de contexto*, remueve las tareas de los procesadores y selecciona las tareas de mayor prioridad para que ejecuten.

Si el sistema de computación es multiprocesador, el *Microkernel* también sincroniza la actividad entre todos los procesadores para optimizar la performance. En este caso, se ejecuta simultáneamente en todos los procesadores. A su vez, como su rol principal es mantener los procesadores tan productivos y cargados como sea posible, también implementa las técnicas de *SMP* (Symmetric Multiprocessing) y *soft affinity*.

El *multiprocesamiento simétrico* permite que los threads de cualquier proceso, incluyendo el Sistema Operativo, puedan ejecutar en cualquier procesador. Incluso, un proceso puede ejecutar varios de sus threads en diferentes procesadores al mismo tiempo. Por otro lado, utiliza la técnica de *afinidad por software*, esto es, poder asignar un thread a un procesador vía software, o sea, a través de una llamada a la Win32 API.

Para poder desalojar (*preempts*) y despachar los threads, el código del Sistema Operativo deberá ser *reentrante*. Esto significa, que será capaz de ser interrumpido y luego reasumir su ejecución sin que se corrompa y podrá ser compartido por diferentes threads, ejecutando las mismas líneas de código en diferentes procesadores.

El *Microkernel*, técnicamente no se ejecuta en threads y, además, es la única parte del Sistema Operativo que no se puede desalojar del procesador ni paginar a disco. El resto del sistema, incluyendo *WindowsNT Executive*, es totalmente *reentrante* y puede ser desalojado de la CPU, con el objeto de maximizar la eficiencia de todo el sistema.

Por otro lado, el *Microkernel* sincroniza las actividades del *WindowsNT Executive Services*, como son el I/O Manager y el Process Manager. También, los componentes del *WindowsNT Executive* acceden a los *Microkernel objects*, varios de los cuales serán analizados en las secciones de Comunicación y Sincronización entre threads.

El módulo *Virtual Memory Manager* es el encargado de la administración de memoria del sistema. WindowsNT trabaja con el sistema de memoria virtual, con paginado por demanda y está basado en un espacio de direccionamiento plano y lineal de 32-bit.

El espacio de direccionamiento virtual total es de 4 GB de direcciones (2^{32}) y se encuentra dividido en 2 particiones de la siguiente manera [Msw00]:

- *Low Memory*: esta partición de 2 GB se encuentra en las posiciones más bajas del sistema: desde la posición 0x00000000 hasta la 0x7FFFFFFF. Esta partición está dedicada al direccionamiento privado de cada proceso.

- *High Memory*: esta partición de 2 GB se encuentra en las posiciones más altas del sistema: desde la posición 0x80000000 hasta 0xFFFFFFFF. Esta partición está reservada al direccionamiento del sistema.

A continuación se observa el esquema de memoria virtual y particiones de WindowsNT:

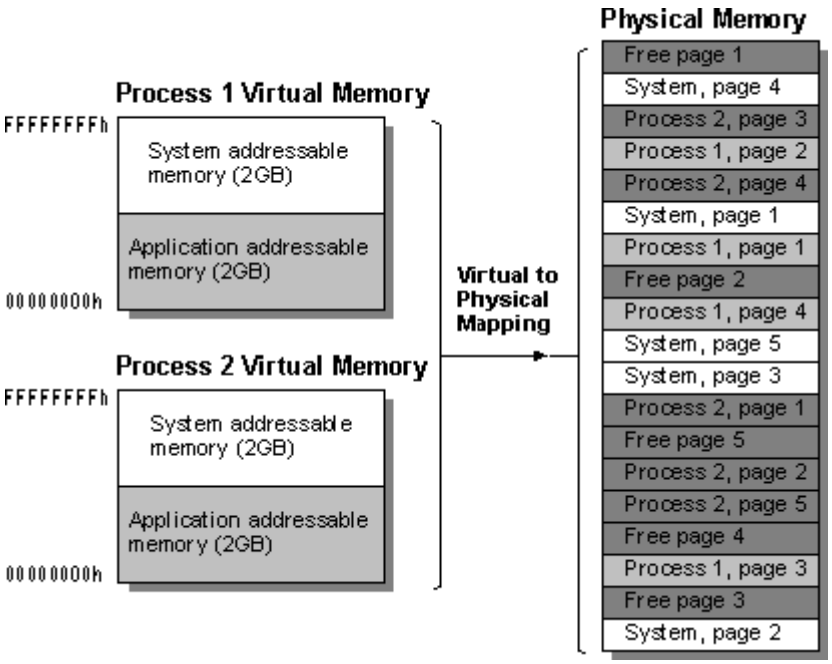


Figura 5 - Esquema de Memoria Virtual de WindowsNT.

A su vez, el Sistema Operativo mantiene tablas con las páginas de cada proceso y, de esta manera, puede localizar la ubicación real de cada una de ellas:

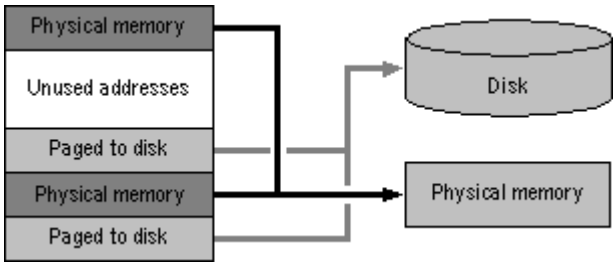


Figura 6 - Tablas del Sistema, para localizar las páginas de cada proceso.

En WindowsNT, el *Microkernel* y la *HAL*, están optimizados para la atención de interrupciones y el despacho de eventos externos. El *Microkernel* opera con 32 niveles de interrupciones, en forma *preemptive* y con la facilidad de enmascarar diferentes niveles.

Cuando ocurre una interrupción de mayor prioridad en el sistema, el *Microkernel* suspende toda la ejecución relativa a un nivel inferior de prioridad y comienza inmediatamente la atención de la nueva interrupción.

Para optimizar la performance del sistema y la eficiencia en la atención de interrupciones, los drivers de dispositivos están compuestos por cuatro componentes [Msb00]:

- Una rutina de inicialización que prepara y configura el hardware a ser controlado.
- Una rutina de servicio de interrupción (*ISR, Interrupt Service Routine*), la cual maneja la interrupción del dispositivo.
- Una o más llamadas a procedimientos diferidos (*DPC, Deferred Procedure Call*), que realizan todo el procesamiento no-crítico del driver.
- Un thread del sistema (*System thread*), en donde se realiza las funciones de muy baja prioridad.

El driver de un dispositivo comienza con la ejecución de su rutina de inicialización. Este módulo registra el driver en el sistema, realiza la configuración inicial y luego registra la *ISR* para ese dispositivo. El driver quedará en espera, consumiendo solamente recursos de memoria. Cuando ocurre una interrupción, la *ISR* es la

encargada de atender al dispositivo. Su código deberá ser escrito de forma tal que realice el trabajo mínimo necesario y libere al dispositivo para que pueda interrumpir nuevamente. Sus funciones principales son: salvar el contexto de la actividad suspendida; atender la interrupción; llamar a una rutina *DPC*, la cual se encarga de realizar todo el trabajo no-crítico de I/O; y luego retornar:

```
ISR()
{
salvar el contexto();
atender la interrupción();
DPC;
}
```

La ejecución de la *ISR* sólo podrá ser interrumpida por otra de mayor prioridad. Es por esto, que la latencia de interrupción (**interrupt latency**) es difícil de predecir en WindowsNT.

Cuando la *ISR* finaliza su trabajo, la rutina *DPC* podrá ejecutar inmediatamente o no, dependiendo si hay más *ISR* pendientes en el sistema. La mayor parte del procesamiento de la interrupción se realiza durante la ejecución de la *DPC*. En esta etapa, se ejecutan todas las tareas no-críticas del driver y, además, se pueden crear *System threads* para realizar otras funciones de menor prioridad.

En WindowsNT, todos los drivers de dispositivos se ejecutan dentro de un proceso de sistema (System Process), el cual contiene múltiples *System threads* y, por lo tanto, tienen acceso directo a todo el hardware a través de la *HAL*.

Otro mecanismo muy poderoso para las aplicaciones de Tiempo Real, es la comunicación I/O asincrónica (*Asynchronous I/O*). Una aplicación puede encolar su solicitud de I/O e inmediatamente continuar su procesamiento sin tener que esperar o responder por algún evento de finalización. A su vez, existen mecanismos para verificar si se completó la solicitud de I/O. La aplicación puede consultar por estos eventos en cualquier otro momento en el futuro.

Una forma de notificar la finalización de I/O, es a través de una *APC (Asynchronous Procedure Call)*. Esta función se ejecuta en forma asincrónica en el contexto de un thread en particular. Cada aplicación tiene su propia cola de *APCs* y existen diferentes funciones de la Win32 API para poder encolar una *APC* a un thread y para esperar por la ocurrencia de algún evento que dispare la ejecución de una *APCs*. Cada vez que se encola una *APC* a un thread, el sistema genera una interrupción por software y la función solicitada se ejecutará cuando el thread sea planificado nuevamente.

En la siguiente tabla se muestra la jerarquía de interrupciones y sus niveles de prioridad [Liu99]:

Nivel ó prioridad	Interrupción – Descripción
31 – Alta.	Error de hardware (Hardware Error interrupt, NMI). Esta interrupción es NO enmascarable y cuando ocurre, se detiene inmediatamente el sistema.
30	Falla en la alimentación de energía (Power Failure interrupt).
29	Entre procesadores (Inter-Processor interrupt).
28	Reloj del Sistema (System Timer interrupt).
27-12	Estos niveles están asignados para dispositivos de Entrada/Salida y se corresponden con los niveles IRQ (Interrupt Request Line) 0-15 del hardware (pueden ser PCs u otras arquitecturas).
11-4	Estos niveles están reservados para dispositivos de Entrada/Salida y generalmente no se usan.
3	Programa de debugger (Software Debugger interrupt).
2	Interrupción por software. En este nivel se ejecutan las llamadas a procedimientos diferidos (DPC, Deferred Procedure Call) y el *dispatcher/scheduler*
1	Interrupción por software. En este nivel se ejecutan las llamadas a procedimientos asincrónicos (APC, Asynchronous Procedure Call).
0 – Baja.	En este nivel se ejecutan todos los procesos y threads, los cuales son planificados por el Sistema Operativo.

En las próximas secciones, se analizarán los servicios básicos y funciones que provee la Win32 API para Tiempo Real. Las descripciones se realizarán en forma genérica para toda la familia de sus Sistemas Operativos y, cuando corresponda, se especificarán las funciones que son únicas para un sistema en particular.

3.3.3 Reloj del Sistema.

Los Sistemas Operativos de la plataforma Win32 requieren que la arquitectura provea un Reloj de Tiempo Real (*Real-Time Clock, RTC*), el cual contiene una memoria no volátil para almacenar la fecha y hora en forma permanente.

Cuando el O/S realiza el *bootstrap*, consulta al *RTC* e inicializa la fecha y hora del sistema (*System Time*).

A su vez, el O/S provee una rutina de software especial, llamada Reloj del Sistema (*System Timer*), la cual se ejecuta periódicamente en cada interrupción de reloj (interrupción 28). A cada invocación de esta rutina se la llama *tick* de reloj.

El *System Timer* es un componente fundamental para el O/S y para las aplicaciones de Tiempo Real. Sus principales funciones son:

- Actualizar y mantener la fecha y hora del sistema (*System Time*).
- Calcular el *time-slice* de procesamiento de cada thread.
- Administrar y controlar diferentes timers establecidos por el O/S y por las aplicaciones.
- Mantener e incrementar el contador global de *ticks* del sistema.

La resolución del *System Timer* o, en forma equivalente, el período de la interrupción de reloj, es diferente por cada O/S [Msw00]:

- En Windows2000/NT: es aproximadamente 10 ms.
- En Windows98/95: es aproximadamente 55 ms.

La Win32 API provee una función llamada *GetTickCount(..)*, la cual retorna la cantidad de milisegundos que transcurrieron desde que se inició el sistema. La resolución de esta función depende de la granularidad del *System Timer*.

Dependiendo de la plataforma de hardware subyacente, se pueden utilizar las funciones *QueryPerformanceCounter(..)* y *QueryPerformanceFrequency(..),* para medir el transcurso del tiempo con mayor resolución. Estas rutinas pueden trabajar con una precisión de 0.8 microsegundos (0.0008 ms).

3.3.4 Planificación y multithreading.

Toda aplicación Win32 consiste en uno o más procesos. Un *proceso*, en forma simplificada, es un programa en ejecución. Uno o más threads se ejecutan en el contexto de un proceso. Un *thread* es la unidad básica de ejecución para la cual el O/S asigna el procesador. Un thread puede ejecutar cualquier parte del código del proceso, incluso partes que están siendo ejecutadas por otros threads. Una *fiber* es una unidad de ejecución que puede ser planificada por la aplicación. Las fibers se ejecutan en el contexto de los threads que los planifican [Msw00].

Un *job object* permite que un grupo de procesos pueda ser manejado como una unidad. Las operaciones realizadas sobre un job object afectan a todos los procesos asociados con él. A un job object se le asigna un nombre y se lo puede configurar con atributos de seguridad, parámetros de ejecución y optimización comunes para todos sus procesos.

Cada proceso provee los recursos necesarios para ejecutar un programa. Un proceso tiene un espacio de direccionamiento virtual, código ejecutable, datos, un

conjunto de manejadores de objetos, variables del entorno, una *base priority* y un mínimo y máximo working set.

Un proceso puede crear a otros procesos, los cuales se ejecutan en forma independiente del proceso que los creó. Por simplicidad, a esta relación se la llama padre-hijo *(parent-child)*.

Cada proceso es creado con un thread, llamado el *primary thread*. A partir de este thread, se pueden crear múltiples threads, y éstos a su vez, pueden crear a otros threads y así sucesivamente (*multithreading*). La cantidad de threads que un proceso puede crear está limitada solamente por la disponibilidad de memoria y recursos del sistema.

Los threads de un proceso comparten su espacio de direccionamiento y los recursos del sistema. Además, cada thread mantiene manejadores de excepción, una *scheduling priority* y un conjunto de estructuras que el sistema usará para salvar su contexto. El *thread context* incluye: los valores de los registros del procesador, el *kernel* stack, un bloque de entorno y un user stack en el espacio de direccionamiento del proceso.

Todos los Sistemas Operativos de la plataforma Win32 son *preemptive multitasking*. Esto significa, que sus núcleos son multitarea y trabajan con desalojo del procesador.

Cada thread en el sistema tiene asignada una *scheduling priority*. El *scheduler* del sistema mantiene una cola por cada nivel de prioridad (*dispatch queue* o *eligible queue*). Cada una de ellas, contiene los threads listos para ejecutar (*ready threads*).

El algoritmo utilizado por el *scheduler* para planificar los threads, es el llamado *highest priority ready thread first*. Esto significa que asignará el procesador al primer thread de mayor prioridad que esté listo. En un sistema multiprocesador, se elegirán los N threads de mayor prioridad listos para ejecutar [Msb00].

Con este algoritmo, el *scheduler* determina cuál de todos los *ready threads* recibe el próximo *time-slice* del procesador. La longitud del *time-slice* es de aproximadamente 20 ms., pero depende del O/S y del procesador en particular.

Un aspecto muy importante para las aplicaciones de Tiempo Real es la política utilizada para seleccionar los threads de igual prioridad (*intralevel scheduling*). En Win32, se implementa la técnica de *round-robin*, en donde los threads de una misma cola (igual nivel de prioridad), se ejecutan en forma cíclica, obteniendo un *time-slice* por cada vuelta.

Una de las características más poderosas para Tiempo Real en Windows2000, es que provee la posibilidad de seleccionar el *intralevel scheduling* entre las políticas de

round-robin y *FIFO*. Esta última técnica es muy importante, dado que se puede asignar la máxima prioridad a un thread de tiempo crítico y asegurarse que cuando comience su ejecución, nunca será desalojado del procesador por ningún thread de su mismo nivel de prioridad [Liu99]. En este caso en particular, se dice que el thread está ejecutando en forma *non-preemptable*.

El O/S realiza un *context switch*, cada vez que necesita asignar el procesador a otro thread. Los eventos que causan este cambio de contexto son los siguientes, enumerados de acuerdo a su prioridad:

- Ocurre una interrupción en el sistema.
- Un thread de mayor prioridad está listo para ejecutar.
- El *time-slice* del thread que está ejecutando ha expirado.
- El thread que está ejecutando, necesita esperar por algún evento o finaliza su ejecución.

Para realizar un *context switch*, el O/S ejecuta los siguientes pasos:

- Almacena el contexto del thread que estaba ejecutando.
- Coloca al thread al final de la cola de su nivel de prioridad (en el caso de *round-robin*).
- Busca el thread de mayor prioridad listo para ejecutar.
- Remueve el thread de la cola, carga su contexto y lo ejecuta.

3.3.5 Prioridades.

A cada thread se le asigna una *scheduling priority*. La prioridad de cada thread está dada por su *base priority* y su *dynamic priority*. La *base priority* está formada por los siguientes 2 componentes:

- La *priority class* del proceso.
- El *priority level* del thread, relativo a la *priority class* de su proceso.

Cada proceso pertenece a alguna de las siguientes *priority classes*:

- IDLE_PRIORITY_CLASS.

- BELOW_NORMAL_PRIORITY_CLASS **(*)**.
- NORMAL_PRIORITY_CLASS.
- ABOVE_NORMAL_PRIORITY_CLASS **(*)**.
- HIGH_PRIORITY_CLASS.
- REALTIME_PRIORITY_CLASS.

(*) Sólo disponible en Windows2000.

Un proceso se crea un mediante la función *CreateProcess(..)* y, si no se especifica alguna prioridad en particular, el O/S asume NORMAL_PRIORITY_CLASS. La *priority class* de un proceso se puede cambiar en cualquier momento, a través de la función *SetPriorityClass(..)*.

A su vez, cada thread tiene su *priority level*:

- THREAD_PRIORITY_IDLE.
- THREAD_PRIORITY_LOWEST.
- THREAD_PRIORITY_BELOW_NORMAL.
- THREAD_PRIORITY_NORMAL.
- THREAD_PRIORITY_ABOVE_NORMAL.
- THREAD_PRIORITY_HIGHEST.
- THREAD_PRIORITY_TIME_CRITICAL.

Todos los threads son creados con la función *CreateThread(..)* y con el *priority level* THREAD_PRIORITY_NORMAL. El *priority level* de un thread se puede cambiar en cualquier momento, a través de la función *SetTheadPriority(..)*.

A su vez, cada thread tiene una *dynamic priority*, la cual inicialmente es igual a la *base priority*. Dado que el O/S trabaja con planificación por prioridades, posee un mecanismo para incrementar (*boost*) la *dynamic priority* de los threads que tienen menor prioridad. Esto permite asegurar una distribución equitativa del tiempo del procesador, mejorar la respuesta del sistema y evitar que los threads queden en un estado de *inanición*. Esta técnica no se aplica a los threads que pertenecen a REALTIME_PRIORITY_CLASS.

El O/S incrementa o disminuye la *dynamic priority*, de acuerdo a los siguientes eventos:

- Cuando un proceso de NORMAL_PRIORITY_CLASS se convierte en *foreground* (esto es, el proceso que está asociado con la ventana activa), el *scheduler* incrementa su *priority class* hasta llegar a ser igual o mayor que los demás procesos que ejecutan en *background*. Una vez que el proceso deja de estar en *foreground*, su *priority class* es restaurada a su configuración original.

- Cuando una ventana recibe input del usuario (teclado, mouse) o eventos de tiempo, el *scheduler* incrementa la prioridad del thread que maneja esa ventana.

- Cuando se satisfacen las condiciones de espera de algún thread que está bloqueado, el *scheduler* incrementa su prioridad. Por ejemplo, cuando finaliza un solicitud de I/O, el thread que estaba bloqueado recibe un incremento de prioridad para poder atender a su requerimiento.

Luego de incrementar la prioridad de un thread, el *scheduler* reduce su *priority level* en uno por cada vez que el thread completa un *time-slice*, hasta llegar a su *base priority*. La *dynamic priority* nunca podrá ser menor que la *base priority*.

Solamente en Windows2000/NT se puede deshabilitar este mecanismo de ajuste dinámico de prioridades, a través de las funciones *SetProcessPriorityBoost(..)* y *SetThreadPriorityBoost(..)*.

Como fue estudiado en la sección anterior, Windows2000 soporta la posibilidad de configurar el *intralevel scheduling*. Para utilizar esta nueva característica, el O/S introduce 9 *job scheduling classes.* Por default, el O/S planificará a todos los threads con la técnica *round-robin*. Para que un proceso y todos sus threads puedan ser planificados con la política *FIFO*, su *priority class* deberá ser REALTIME_PRIORITY_ CLASS y deberá pertenecer a un *job object* que tenga habilitado el flag JOB_OBJECT_LIMIT_SCHEDULING_CLASS y su *job scheduling class* sea 9 (la más alta) [Liu99].

Todos los threads que ejecutan en el sistema, se encuentran en el nivel 0 de la jerarquía de interrupciones. Esto significa que, independientemente de la *scheduling priority* que tenga un thread, solamente será planificado cuando no exista ninguna interrupción pendiente de ejecución.

Por otro lado, en el nivel 2 de la jerarquía de interrupciones, se ejecutan los procedimientos *DPCs* y el *dispatcher/scheduler*. Dentro de este nivel, los *DPCs* tienen mayor prioridad y, por lo tanto, el *scheduler* sólo podrá ejecutar cuando la cola de *DPCs* esté vacía [Liu99].

En la siguiente tabla se muestran los 32 niveles de prioridades con los cuales trabaja la plataforma Win32. La *base priority* de los threads se corresponde con un número entero entre 0 (la más baja y reservada para el sistema) y 31 (la más alta), dependiendo de su combinación de *priority class* y *priority level* [Msw00]:

Base Priority	Process Priority Class	Thread Priority Level
1 - Más baja.	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
1	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
1	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
1	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
1	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
2	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
3	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
4	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
4	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
5	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
5	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
5	Background NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
6	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
6	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
6	Background NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
7	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
7	Background NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
7	Foreground NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
8	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
8	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
8	Foreground NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
8	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
9	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
9	Foreground NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
9	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
10	Foreground NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
10	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
11	Foreground NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
11	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
11	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
12	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
12	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
13	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
14	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
15	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
15	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_TIME_CRITICAL
15	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_TIME_CRITICAL
15	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_TIME_CRITICAL
15	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_TIME_CRITICAL
15	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_TIME_CRITICAL
16	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
17	REALTIME_PRIORITY_CLASS	-7
18	REALTIME_PRIORITY_CLASS	-6
19	REALTIME_PRIORITY_CLASS	-5
20	REALTIME_PRIORITY_CLASS	-4
21	REALTIME_PRIORITY_CLASS	-3
22	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
23	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
24	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
25	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
26	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
27	REALTIME_PRIORITY_CLASS	3
28	REALTIME_PRIORITY_CLASS	4
29	REALTIME_PRIORITY_CLASS	5
30	REALTIME_PRIORITY_CLASS	6
31 - Más alta.	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_TIME_CRITICAL

Las clases BELOW_NORMAL_PRIORITIY_CLASS y ABOVE_NORMAL_PRIORITIY_CLASS y los valores -7, -6, -5, -4, -3, 3, 4, 5 y 6 son sólo soportados por Windows2000.

3.3.6 Comunicación entre threads.

La Win32 API provee mecanismos que facilitan la comunicación y permiten compartir información entre aplicaciones. En forma colectiva, las actividades involucradas por estos mecanismos se las llama *Interprocess Communications (IPC)*.

Algunos de estos mecanismos facilitan el trabajo entre procesos de una computadora y otros facilitan la división del procesamiento entre computadoras de una red.

En forma genérica, las aplicaciones que usan *IPC*, se categorizan en *clients* y *servers*. Un *client* es una aplicación o proceso que solicita un servicio a otra aplicación o proceso. Un *server* es una aplicación o proceso que responde a la solicitud del *client*. Muchas aplicaciones actúan como *clients* y *servers*, dependiendo de la situación.

A continuación se describen los mecanismos de *IPC* que soporta la Win32 API, con sus principales características [Msw00]:

- **Clipboard:** este mecanismo actúa como un repositorio central de datos, para ser compartidos entre todas las aplicaciones. Es un medio de intercambio de información débilmente acoplado, en donde las aplicaciones sólo deben acordar el formato de los datos. Las aplicaciones pueden residir en una computadora o en diferentes computadoras en una red.

- **COM:** las aplicaciones que usan OLE (Object Link and Embedding) manejan *compound documents*, esto es, documentos formados por datos provenientes de diferentes aplicaciones. La base de OLE es el Component Object Model (COM). Los programas que usan COM, pueden comunicarse con una gran variedad de otros componentes, los cuales interactúan como *objects* y *clients*. Distributed COM extiende su funcionalidad para trabajar en red.

- **DDE:** es un protocolo que habilita a las aplicaciones a intercambiar datos en un variedad de formatos, mediante una sola operación o en forma continua. Es una extensión de Clipboard y generalmente se usa para aplicaciones que están más fuertemente acopladas. Se puede utilizar en una misma computadora y en procesos que ejecutan en diferentes computadoras en una red.

- **File Mapping:** este mecanismo habilita la posibilidad de manejar el contenido de un archivo como si fuera un bloque de memoria en el espacio de direccionamiento del proceso. Dos o más procesos pueden acceder a este mismo archivo, pero deben utilizar algún mecanismo de sincronización para prevenir la corrupción de datos. En caso de necesitarse una *named shared*

memory entre procesos, se especifica el system swapping file como objeto de file-mapping y se trata como un *shared memory block*. Otros procesos pueden acceder a esta área compartida en memoria, abriendo el mismo objeto de file-mapping. Este mecanismo sólo puede ser utilizado entre procesos que ejecutan en una misma computadora.

- **Mailslot:** este mecanismo provee comunicación en una sola dirección. Cualquier proceso que crea un mailslot es un *mailslot server*. Otros procesos, los *mailslot clients*, envían mensajes al *mailslot server* escribiendo los mensajes en su mailslot. Cada mensaje entrante es agregado al final del mailslot. Un proceso puede cumplir ambos roles, por lo tanto, puede establecer una comunicación bidireccional, usando múltiples mailslots. Este mecanismo permite trabajar en red y enviar mensajes de tipo broadcast a todas las computadoras de un dominio de red.

- **Pipes:** este mecanismo provee 2 tipos de comunicaciones: *anonymous pipes* y *named pipes*. *Anonymous pipes* se utilizan para transferir información en una sola dirección, entre procesos padre-hijo y se utilizan siempre en forma local. Por otro lado, los *named pipes*, son utilizados para transferir datos entre procesos que no están relacionados y en procesos en diferentes computadoras. Un *named pipe*, se puede utilizar para comunicaciones bidireccionales entre un *pipe server* y múltiples *pipes clients*.

- **RPC:** mediante este mecanismo, las aplicaciones pueden llamar a funciones para que se ejecuten remotamente. Puede operar en una misma computadora o en diferentes computadoras en una red. Este mecanismo es compatible con el standard Open Software Foundation (OSF), Distributed Computing Environment (DCE). Facilita el desarrollo de aplicaciones *client/server* de alta performance y fuertemente acopladas.

- **Windows Sockets:** es una interface independiente del protocolo, capaz de soportar las capacidades de redes actuales y futuras. Toma las ventajas y características propias del protocolo subyacente. Este mecanismo está basado en los sockets del Berkeley Software Distribution (BSD). Una aplicación que usa Windows Sockets, puede comunicarse con otra implementación de sockets en sistemas diferentes.

- **WM_COPYDATA:** Este mecanismo es simplemente un tipo de mensaje de sistema, con una estructura de datos asociada. Se puede utilizar para enviar un mensaje de una aplicación a otra.

Además de estos mecanismos de comunicación, existen otras técnicas para compartir información entre threads de un mismo proceso, de una manera muy simple y eficiente:

- **Variables Globales:** dado que todos los threads de un proceso comparten el mismo espacio de direccionamiento, este mecanismo provee un área de memoria compartida entre todos ellos.

- **Open Handles:** los threads pueden utilizar los mismos recursos compartidos en forma concurrente: archivos, pipes y ports de comunicaciones.

Dado que proveer acceso compartido a recursos puede provocar diferentes conflictos, es necesario que los múltiples threads sincronicen su ejecución. En la próxima sección, se describirán diferentes técnicas y métodos para realizar la sincronización entre threads de Win32.

3.3.7 Sincronización entre threads.

La plataforma Win32 provee diferentes formas de coordinar la ejecución de múltiples threads. Las funciones descriptas en esta sección, proveen mecanismos que son utilizados por threads para sincronizar el acceso a recursos compartidos (propios o del sistema).

Para sincronizar el acceso a un recurso compartido, se utiliza alguno de los *synchronization objects*, en alguna de las *wait functions*. El estado de un *synchronization object* es *signaled* o *non-signaled.* Por otro lado, las *wait functions* permiten que un thread bloquee su propia ejecución hasta que un objeto que está en un estado *non-signaled* pase a un estado *signaled*.

A continuación se describen los tres tipos de *wait functions*, con sus principales características [Msw00]:

- **Single-Object:** esta clase de funciones trabajan con un solo *synchronization object*. A modo de ejemplo: la función *WaitForSingleObject(..)* bloquea la ejecución del thread que la llama, hasta tanto el objeto especificado pasa al estado *signaled* o hasta que expira un contador de tiempo especificado. En situaciones particulares, puede no especificarse un contador de tiempo, con lo cual, retornará solamente si el objeto pasa a un estado *signaled*.

- **Multiple-Object:** en esta clase de funciones, se especifica un arreglo de *synchronization objects*. Las funciones bloquean la ejecución de los threads que las llaman, hasta tanto alguno o todos los objetos fueron señalizados al estado *signaled*, o cuando expira el contador de tiempo (en caso de haber sido especificado).

- **Alertable:** esta clase de funciones permiten que las *wait functions* retornen cuando se cumplan los criterios especificados, como así también cuando el

sistema encola una rutina de finalización de I/O o una *APC* para que se ejecute en el thread que estaba esperando.

Un *synchronization object* es un objeto del sistema, el cual se especifica en alguna de las *wait functions* para coordinar la ejecución de múltiples threads. Más de un thread puede tener un *handle* al mismo *synchronization object*, haciendo posible la sincronización entre threads.

Los siguientes *synchronization objects* son provistos por el sistema, exclusivamente para la sincronización entre threads [Msw00]:

- **Event:** notifica a uno o más threads que un evento ha ocurrido. Las funciones *SetEvent(..) y ResetEvent(..)* señalizan a un *event object* al estado *signaled* y *non-signaled* respectivamente.

- **Mutex:** en un instante dado, un solo thread puede tener acceso a este objeto. De esta forma, se coordina el acceso mutuamente excluyente del recurso compartido.

- **Semaphore:** mantiene un contador entero entre 0 y algún valor máximo, limitando el número de threads que pueden estar esperando por el recurso compartido.

- **Waitable timer:** notifica a uno o más threads que están en espera, que el tiempo especificado ha transcurrido.

A su vez, también están disponibles otros objetos que no son específicos para sincronización, pero que se utilizan en varias *wait functions*: *Thread, Process, Job, Console input y Change notification*. A modo de ejemplo: cuando un proceso necesita esperar por la finalización de algún thread en particular, puede llamar a la función *WaitForSingleObject(..)*, especificando al objeto *Thread* como parámetro. En este caso, el proceso quedará bloqueado hasta tanto el thread no haya terminado.

Cuando una *wait function* retorna de su llamada, puede modificar los estados de algunos de los *synchronization objects*. Estas modificaciones ocurren sólo en los objetos cuyo estado *signaled* fue el que causó que la *wait function* haya retornado. Las situaciones en las cuales ocurren estos cambios de estado, dependen de cada objeto particular:

- El contador de un *Semaphore* se decrementó en uno y llegó a cero. En este caso, el *semphore* pasa a un estado *non-signaled*.

- Los estados de *Mutex*, *Event auto-reset* (un tipo especial de *Event*) y *Change notification* pasan al estado de *non-signaled*.

- El estado de un *Synchronization timer* (un tipo especial de *Waitable timer*) pasa a *non-signaled*.

- Otros objetos, como *Thread, Process* y *Console input* no son afectados por las *wait functions*.

También existen otros mecanismos de sincronización entre threads, como ser **Overlapped Input and Output**, **Asynchronous Procedure Calls**, **Critical Sections** y variables **Interlocked**.

Las *Critical Sections* son esencialmente lo mismo que los objetos *Mutex*, excepto que sólo pueden utilizarse entre threads de un mismo proceso. A su vez, proveen un mecanismo más rápido y eficiente para la sincronización mutuamente excluyente. A través de la función *EnterCriticalSection(..)* se accede en forma exclusiva al recurso compartido y luego se libera mediante un llamada a *LeaveCriticalSection(..)*.

Las variables *Interlocked*, proveen un simple mecanismo para sincronizar el acceso de múltiples threads a variables en memoria compartida. La Win32 API provee funciones atómicas para acceder, leer y modificar este tipo de variables.

Y por último, la plataforma Win32 provee diferentes funciones para controlar el flujo de ejecución de un mismo thread o de distintos threads. Un thread puede suspender o reanudar la ejecución de otro, a través de las funciones *SuspendThread(..)* y *ResumeThread(..)* respectivamente. A su vez, un thread puede suspender su ejecución por un intervalo específico de tiempo (especificado en milisegundos), mediante una llamada a *Sleep(..)*.

4 - PLANIFICACIÓN EN TIEMPO REAL Y SU EMULACIÓN

En este capítulo se estudiarán las características de los algoritmos de planificación de tareas para Tiempo Real y se describirán los utilizados tanto en Sistemas Operativos de Propósito General (GPOS, General Purpose Operating System) como en Sistemas Operativos específicos para Tiempo Real (RTOS, Real Time Operating System).

A su vez, se analizarán diferentes técnicas para emular los algoritmos de planificación en Tiempo Real en un GPOS y se describirá el modelo base y los algoritmos de emulación con los cuales se trabajará en esta Tesis.

4.1 Algoritmos de Planificación.

La complejidad de los algoritmos de planificación en Tiempo Real, reside en la diversidad de restricciones que deben cumplir. Las restricciones de tiempo pueden especificarse en términos de diferentes parámetros [Wai97]:

- Tiempo de llegada: es el momento en el cual una tarea ingresa al sistema.
- Tiempo de listo: es el tiempo en el cual una tarea puede comenzar a ejecutar.
- Tiempo de ejecución: es el peor caso de tiempo de ejecución de una tarea.
- *Meta*: es el tiempo en el cual la tarea deberá haber terminado.

Los tiempos de las tareas, en general, son estocásticos: el peor caso de tiempo de ejecución es mucho mayor que su tiempo promedio.

La gran mayoría de los modelos consideran dos tipos de tareas a planificar: las tareas **periódicas** y las tareas **aperiódicas** (esporádicas). Las tareas *periódicas* tienen una serie continua de invocaciones regulares (período de la tarea) y un tiempo máximo de cálculo. Este tipo de tareas suelen ser las más comunes, ya que generalmente los Sistemas de Tiempo Real tienen alguna forma de muestreo regular.

Las tareas *aperiódicas* tienen tiempo de llegada y *meta* arbitrarios y cuando se invocan, se ejecutan por única vez.

La teoría de planificación en Tiempo Real trata con dos clases de algoritmos de planificación: **estáticos** y **dinámicos**.

Un algoritmo *estático* produce un plan de ejecución basado en el conocimiento completo del conjunto de tareas y sus respectivas restricciones. En muchos casos, este plan de ejecución se determina fuera de línea (off-line).

Por todo lado, un algoritmo *dinámico*, produce un nuevo plan de ejecución cada vez que varía el conjunto de tareas o se modifican sus restricciones. Este plan de ejecución se determina en línea (on-line).

En algunos casos se pueden utilizar los mismos algoritmos *estáticos* en un ambiente en línea, lo cual no es siempre adecuado, debido a que los algoritmos *estáticos* pueden no ser óptimos para planes dinámicos y, además, su costo de ejecución puede ser muy elevado.

Como fue mencionado en la sección 3.1, un requerimiento fundamental para un RTOS es que sea **preemptive,** esto es, que acepte desalojo del procesador. Este requisito es muy importante para los algoritmos de planificación en Tiempo Real. El concepto de *preemptive* se puede ilustrar con el siguiente ejemplo: si la tarea A está ejecutando en el procesador y, en ese momento, llega una nueva tarea B con mayor prioridad que A, entonces el sistema desaloja (*preempts*) a la tarea A y despacha inmediatamente la tarea B para que comience su ejecución [Ade94].

En las próximas secciones se describirán varios algoritmos de planificación utilizados en GPOS y en RTOS, con sus principales ventajas y desventajas para las aplicaciones de Tiempo Real.

4.1.1 Planificación en Sistemas Operativos de Propósito General.

En esta sección, se realiza una introducción a los algoritmos de planificación de tareas más utilizados en GPOS y se estudian desde el punto de vista de satisfacer los requerimientos de los Sistemas de Tiempo Real.

La técnica más sencilla de planificar tareas es ejecutarlas en el orden de llegada (First Come First Served). Este algoritmo es utilizado por algunos RTOS, ya que si el conjunto de tareas consiste solamente en tareas *periódicas*, liberarán al procesador a intervalos regulares. Si las tareas cumplen sus *metas* en la primera invocación también lo harán en todas las siguientes. El plan resultante se denomina *ejecutivo cíclico* y puede ser muy eficiente y con bajo overhead por la reducción del intercambio de tareas.

Sin embargo, esta estrategia presenta varias desventajas. En primer lugar, las tareas de tiempo crítico pueden quedar demoradas por tareas de *metas* más blandas que llegan primero. Por otro lado, ante cualquier cambio en el conjunto de tareas, se debe armar otro plan para poder garantizar sus *metas*. Y por último, no es flexible para

manejar tareas *aperiódicas*, ni situaciones en las que las tareas ejecutan más de lo esperado.

Otro algoritmo es el conocido como Round-Robin, en donde el procesador se va asignando en forma cíclica a cada tarea durante un tiempo breve de ejecución (llamado quantum), de acuerdo al orden de llegada. Bajo este algoritmo, también se puede implementar un *ejecutivo cíclico*, aunque presenta algunos problemas, ya que una tarea con alto uso de procesador puede ser desalojada por otra menos urgente. En muchas ocasiones, el programador deberá construir un plan manualmente, asegurando que cada tarea pueda completar su tiempo de ejecución en un intervalo específico de tiempo y realizar un orden de las mismas de acuerdo a sus prioridades.

Una política tradicional con mayor utilidad se basa en el uso de prioridades. En este caso, el *planificador* seleccionará la tarea con mayor prioridad para ejecutar. Esta tarea ejecutará durante un intervalo de tiempo específico, llamado *time-slice*. Una vez que la tarea finaliza su *time-slice* o cuando libera el procesador por alguna solicitud de servicio al O/S, el *planificador* seleccionará la tarea con la siguiente prioridad.

Estos algoritmos pueden usar dos tipos de prioridades: fijas o variables. Los sistemas de prioridades fijas son menos flexibles, ya que las prioridades de las tareas no se pueden cambiar. Los sistemas de prioridades variables permiten que se modifique las prioridades de las tareas durante su ejecución.

Una política básica aplicable a todos los algoritmos conocidos, es la técnica de remoción o desalojo de tareas (*preemptive*). Como fue explicado en la sección anterior, esta característica es fundamental para la planificación en Tiempo Real.

Una última clase de algoritmos de planificación utilizados en GPOS, son los algoritmos multicolas (*multilevel*). La idea es manejar múltiples colas de listo, en donde cada una tiene una prioridad asignada y puede estar administrada por un política en particular. Primero se ejecutan las tareas que están en las colas de mayor prioridad (el nivel más alto) y sólo se podrán ejecutar las tareas que están en los niveles inferiores, cuando la cola del nivel superior esté vacía.

Para evitar que las tareas de menor prioridad entren en un estado de *inanición*, los algoritmos tradicionales suelen usar técnicas de envejecimiento (aging), las cuales aumentan la prioridad de dichas tareas. Esta política no es adecuada para los Sistemas de Tiempo Real, ya que en ellos, la prioridad de una tarea refleja su criticidad y aumentar las prioridades a tareas de menor urgencia, puede provocar fallas muy graves en la respuesta y estabilidad del sistema.

Como se puede observar, una planificación de tareas con prioridades variables, desalojo del procesador y múltiples niveles, permite una asignación dinámica y eficiente de los recursos. Las tareas duras de Tiempo Real se pueden configurar en un nivel

superior y las tareas blandas de Tiempo Real y otras tareas interactivas, se pueden asignar a niveles inferiores.

El principal problema de utilizar los algoritmos de planificación descriptos en esta sección, reside en que el implementador deberá realizar un ajuste manual de los planes de ejecución. Generalmente, este proceso se realiza en forma iterativa, con estudios exhaustivos y simulaciones. Inicialmente, el implementador construye un ejecutivo, en el cual las prioridades de las tareas reflejan su criticidad. Si alguna de las tareas críticas pierde su *meta* o si la utilización de los recursos es baja, las prioridades se sintonizan nuevamente y, en algunos casos, se optimiza el código de las tareas. Estos cambios y pruebas continúan, hasta que se logra un desempeño satisfactorio del sistema.

4.1.2 Planificación en Sistemas Operativos de Tiempo Real.

En esta sección se analizan algunos algoritmos usados para planificación en Tiempo Real, enfocando sus ventajas y dificultades. Varios de ellos son utilizados en RTOS y en Sistemas Operativos de Investigación.

Un algoritmo con gran difusión en la actualidad, es el llamado **Tasa Monotónica (RM - Rate Monotonic)**. Este algoritmo es *estático*, trabaja con prioridades fijas, con desalojo del procesador y se utiliza para planificar tareas *periódicas* independientes. Recibe este nombre porque ordena las tareas en forma monótonamente creciente de acuerdo a su tasa de ejecución: asigna mayor prioridad a las tareas que tienen períodos de ejecución más breves.

Este algoritmo tiene varias ventajas. Por un lado, es un algoritmo óptimo, en el sentido que, si no existe un plan para un conjunto de tareas *periódicas* independientes predecibles por *RM*, entonces no existe ningún otro plan que pueda hacerlo. Otra ventaja reside en su estabilidad, dado que las tareas de menor período, que se presumen como las más críticas, podrán cumplir las *metas*.

Una condición suficiente para que un conjunto de n tareas *periódicas* independientes sean planificables por *RM*, es que:

$$\sum_{i=1}^{n} (C_i/T_i) \leq (2^{1/n} - 1) \underset{n \to \infty}{\rightarrow} \ln 2 \text{ (equivale al 69 \% de uso del procesador).}$$

donde T_i es el período de la tarea y C_i su peor tiempo de ejecución.

Esta proposición significa que si se cumplen las condiciones de la inecuación, entonces el conjunto de tareas es planificable por *RM*, con un grado de utilización máximo de procesador del 69%. Esta condición impone una cota de uso del procesador relativamente baja [Wai97].

Por otro lado, el teorema de Zona Crítica provee una condición suficiente y necesaria para comprobar si un conjunto de tareas puede cumplir con todas sus *metas*.

Según este teorema, si cada tarea cumple su primera *meta* cuando todas las tareas comienzan su ejecución al mismo tiempo, entonces siempre se cumplirán todas las *metas* para toda combinación de tiempos de comienzo.

Para analizarlo, se deberá verificar que todas las tareas cumplan su primera *meta* cuando comienzan simultáneamente. Como el conjunto de tareas es periódico, luego de transcurrido el mínimo común múltiplo de los períodos de las tareas, la secuencia de ejecución se repite.

Uno de los problemas del algoritmo *RM*, está relacionado con la ejecución de tareas *aperiódicas*. Una posible solución es usar planificación en línea, ejecutando las tareas como si fueran *periódicas* de una única instancia. Para los cálculos de garantía se la considera como una tarea *periódica* más. Otras soluciones proponen ejecutar las tareas *aperiódicas* en los tiempos libres que dejan las tareas *periódicas*.

Otro problema del algoritmo *RM* es que, aunque es estable ante sobrecargas, se basa en la idea, no siempre cierta, que la criticidad de una tarea está relacionada directamente a su período.

Además, como el algoritmo *RM* está pensado para tareas independientes, puede provocar inversiones ilimitadas de prioridades en tareas que están relacionadas a través de algún recurso compartido. Si bien las inversiones no pueden eliminarse por completo, lo importante es minimizar su duración.

Otro algoritmo que ha alcanzado un alto grado de aceptación, es el llamado **Meta Más Temprana Primero (EDF - Earliest Deadline First)**. Este algoritmo es *dinámico*, trabaja con prioridades variables, con desalojo del procesador y es óptimo para planificar tareas con tiempos de llegada aleatorios.

En el algoritmo de *EDF*, las tareas con *metas* más próximas tienen mayor prioridad y pueden garantizarse si y sólo si:

$$\sum_{i=1}^{n} (C_i/T_i) \leq 1$$

donde T_i es el período de la tarea y C_i su peor tiempo de ejecución.

Diversos estudios permitieron mostrar que ante sobrecargas en el sistema, el algoritmo *RM* se comporta mejor que *EDF*, porque ejecuta las tareas más criticas logrando estabilidad (en caso de considerar que las tareas críticas son las que tienen menor período). En cambio, cuando se reducen las sobrecargas, el algoritmo *EDF* tiene mejor desempeño ya que no tiene una cota estricta de utilización del procesador (su

cota teórica es 100%). A su vez, *EDF* favorece a las tareas con *metas* más cercanas reduciendo sus tiempos de respuesta [Wai97].

Al introducir tareas *aperiódicas*, el comportamiento con respecto a *RM* es similar. En sistemas no sobrecargados, el desempeño de *EDF* es aún mayor si las tareas *aperiódicas* tienen *metas* más cortas que los períodos de las tareas *periódicas*.

El algoritmo *EDF* también es óptimo cuando no existen bloqueos, pero deja de serlo cuando las tareas son dependientes entre sí. Existen diferentes soluciones y extensiones a este algoritmo para poder minimizar los efectos de este problema.

También existen algunas variantes de *EDF*, como su versión dinámica con prioridades fijas y la llamada **Meta Monotónica (DM - Deadline Monotonic)**, la cual es un algoritmo *estático* y trabaja con prioridades fijas. El concepto es el mismo, ya que asigna la mayor prioridad a la tarea que tenga la *meta* más próxima. Este algoritmo también es óptimo dentro de la clase de algoritmos de prioridades fijas [Liu99].

Otro algoritmo de características similares es el conocido como **Menor Flexibilidad Primero (LSF – Least Slack First)**. Este algoritmo es *dinámico*, trabaja con prioridades fijas, con desalojo del procesador y se utiliza para detectar fallas temporales [Ade94].

La flexibilidad es la diferencia entre la *meta* y el tiempo remanente de ejecución. Una tarea con flexibilidad negativa no podrá cumplir su *meta*. Esto facilita la ejecución de las acciones preventivas correspondientes, antes que las fallas ocurran.

El principal mérito de *LSF* es que mejora la flexibilidad de las tareas para ser planificadas a tiempo y facilita la detección de fallas, pero al igual que *EDF*, tiene problemas ante situaciones de sobrecarga.

Aunque cada uno de los algoritmos estudiados en esta sección tienen algunas desventajas, han demostrado ser mucho más eficientes que los algoritmos de planificación de GPOS, sobre todo si el conjunto de tareas a planificar tiene las características especificadas por los algoritmos, lo cual ocurre en la mayoría de los casos.

4.2 Emulación de la planificación en Tiempo Real.

Si bien seguramente existirán diversas necesidades para instalar, implementar o utilizar un RTOS, diferentes factores justifican la utilización de un GPOS para aplicaciones de Tiempo Real con *metas* firmes o blandas, como son los sistemas SCADA:

- **Integración:** muchos sistemas SCADA están divididos en tareas de Tiempo Real, tareas interactivas y tareas de procesamiento batch. El desarrollo de una aplicación con todas estas tareas en un sólo Sistema Operativo, favorece a su integración, en comparación con una solución mixta y heterogénea.

- **Desarrollo:** debido al crecimiento de esta clase de sistemas, los programadores desean implementar sus aplicaciones en plataformas en las cuales están familiarizados a trabajar y en las que existen una gran cantidad de herramientas de desarrollo para la construcción de aplicaciones.

- **Portabilidad:** las aplicaciones escritas en un GPOS, con un lenguaje abierto y estándar, son más portables que los programas escritos en Sistemas Operativos con fines específicos y propietarios.

- **Costo:** implementar toda la aplicación en un sólo Sistema Operativo, reduce el costo total del sistema, los tiempos de desarrollo y mejora la calidad del código.

Por estas razones, se desea integrar las aplicaciones de Tiempo Real firmes y blandas dentro de un GPOS.

Un GPOS es un sistema que no fue diseñado originalmente para aplicaciones de Tiempo Real. Sin embargo, muchos GPOS de la actualidad, soportan muchas características de los RTOS. Una de las mayores diferencias entre ambas clases de sistemas, es la planificación de tareas y la predictibilidad. En un GPOS, se utilizan algoritmos de planificación para optimizar la administración de recursos, mientras que el objetivo de los algoritmos de los RTOS es predecir la ejecución de las tareas y cumplir con sus requerimientos de tiempo.

Para poder satisfacer con los requerimientos de las aplicaciones de Tiempo Real en un GPOS, se deberá construir un *planificador* de tareas de alto nivel que ejecute sobre el *planificador* nativo del GPOS. Este *planificador* deberá emular alguno de los algoritmos de planificación en Tiempo Real estudiados en la sección anterior. Para poder construir dicho emulador, es fundamental que el *planificador* del GPOS trabaje con prioridades y permita el desalojo del procesador.

El concepto principal es construir un *planificador* de tareas firmes y blandas, el cual invoca a los servicios del O/S para poder llevar a cabo su función. Las tareas de Tiempo Real deberán ser construidas para ejecutar sobre este *planificador* de alto nivel.

En la próxima sección se analizarán distintas técnicas de emulación, con sus principales ventajas y desventajas.

4.2.1 Diferentes técnicas.

El objetivo de esta sección es estudiar cómo emular la planificación en Tiempo Real sobre un *planificador* de un GPOS.

Las soluciones a este problema varían dependiendo de la exactitud del emulador deseado, la cantidad total de código complejo que se tolera y que tan flexible es la distribución del nuevo código entre las aplicaciones y el sistema.

Las diferentes técnicas que se aplican para la emulación de planificación en Tiempo Real son:

- Asignar manualmente las prioridades a las tareas y dejar que el *kernel* del GPOS las planifique de acuerdo a su algoritmo nativo. Estas prioridades pueden ser asignadas de acuerdo a un algoritmo de planificación en Tiempo Real o de acuerdo a pruebas exhaustivas y simulaciones que se realizan con el conjunto de tareas.

- Implementar el *planificador* y las tareas de acuerdo con la técnica *cliente/servidor*. En este diseño, el *planificador* actúa como *cliente* y las tareas como *servidoras*. Cada tarea queda bloqueada a la espera de una señal de sincronización por parte del *planificador* para que pueda ejecutar su instancia (ciclo de ejecución). El *planificador* señaliza sólo a una tarea por vez, de acuerdo a su algoritmo de planificación. De esta manera, obliga al *planificador* del GPOS a que elija a la única tarea que está lista para ejecutar. Una vez que la tarea finaliza su instancia, se bloquea a la espera de la próxima sincronización.

- Una variante de la técnica anterior, es un diseño *cliente/servidor*, pero donde el *planificador* señaliza a cada tarea al comienzo de cada instancia de ejecución. En este caso, el *planificador* GPOS deberá elegir la tarea a ejecutar de acuerdo a su prioridad. A su vez, esta técnica se puede combinar con la primera, para asignar a cada tarea una prioridad de acuerdo a un algoritmo de planificación en Tiempo Real.

- Implementar un paquete completo de rutinas para que ejecuten sobre el GPOS, con el algoritmo de planificación que se desea. En esta técnica, se llega a reescribir el *planificador* y parte del *kernel* del O/S.

Para implementar la primera técnica, se requiere que cada tarea invoque a una función específica para determinar la prioridad que le corresponde. El desafío en esta política, es poder determinar cuál de todos los niveles de prioridades disponibles, se le asignará a cada tarea que ingresa al sistema. La asignación de prioridades se realiza de acuerdo al algoritmo de planificación en Tiempo Real que se desea emular. El problema principal de este método es que, en muchos casos, la cantidad de prioridades

disponibles son muy pocas y, además, no se sabe realmente qué tareas llegarán en el futuro. Por lo tanto, las prioridades asignadas reflejarán en forma relativa la criticidad de las tareas.

Algunas de las ventajas de esta política son su relativa facilidad de implementación y la reusabilidad de las tareas, dado que su código no se deberá modificar. Por último, para implementar este método es fundamental que el GPOS no utilice la técnica de ajuste dinámico de prioridades de las tareas, ya que provocaría que una tarea que era considerada de alta prioridad deje de serlo.

Por otro lado, la política *cliente/servidor* provee un excelente control para la planificación de tareas. En esta técnica, se requiere que tanto el *planificador* como las tareas deban ser programadas teniendo en cuenta esta política. Esto significa que el *planificador* deberá tener conocimiento total de todas las tareas del sistema y enviar las señales de sincronización cuando corresponda, de acuerdo a sus restricciones de tiempo. Además cada tarea deberá ser programada para bloquearse a la espera de la señalización del *planificador*, ejecutar su instancia y luego volver a bloquearse.

La última opción es la posibilidad de escribir un conjunto de rutinas de muy bajo nivel sobre el *kernel* del GPOS. Esta técnica requiere que se escriba código que generalmente pertenece al O/S, como puede ser el *cambio de contexto* y el *planificador* en sí. La ventaja de esta política es que se puede implementar el algoritmo de planificación que se desee. Por otro lado, requiere mucho trabajo de programación de bajo nivel y el código resultante no es portable. En resumen, esta alternativa es muy efectiva para las tareas de Tiempo Real, pero anula las ventajas que ofrece la utilización de un GPOS, que era justamente lo que se quería preservar.

4.2.2 Modelo base.

Para implementar la emulación de la planificación en Tiempo Real, se trabajará con el siguiente modelo base y sus respectivos parámetros:

- La arquitectura es un sistema monoprocesador, en donde se ejecutarán todas las tareas.
- El O/S es un GPOS y no un RTOS.
- El GPOS trabajará con planificación por prioridades, con multicolas y con desalojo del procesador.
- En el O/S se ejecutarán las tareas de Tiempo Real y otras aplicaciones con diferentes propósitos. Estas aplicaciones competirán con las tareas de Tiempo Real para la asignación del procesador.

- El GPOS trabajará con una cantidad fija de niveles de prioridades: n. Cada nivel tendrá asociado su cola respectiva.

- Un nivel de prioridad se denota como p_{os}, el cual es un número entero entre 0 y n-1. La relación entre p_{os} y la prioridad de una tarea, varía dependiendo del O/S. En este trabajo se asumirá que, mayor p_{os}, mayor es la prioridad asociada.

- En lo posible, el GPOS no utilizará la técnica de ajuste dinámico de prioridades.

- Las tareas de Tiempo Real son *periódicas*.

- La *meta* de cada tarea de Tiempo Real es igual a su período de ejecución.

En este modelo, se utilizará la técnica de emulación *cliente/servidor*, en combinación con la asignación manual de prioridades, de acuerdo con un algoritmo de planificación en Tiempo Real.

Esta política mixta, permite emular un algoritmo de planificación en Tiempo Real de la siguiente manera:

- A cada tarea que ingresa al sistema, se le asigna una prioridad de acuerdo al algoritmo elegido.

- El *planificador* contiene una estructura de datos con todas las tareas y sus restricciones de tiempo.

- Inicialmente cada tarea se bloquea para esperar la señal de sincronización del *planificador*.

- El *planificador* señaliza a cada tarea de acuerdo a su período de ejecución. La frecuencia de ejecución del *planificador* puede ser en cada *tick* de reloj, o se puede optimizar para que se ejecute solamente cuando corresponda planificar por lo menos a una tarea.

- Dado que el O/S trabaja con planificación por prioridades, elegirá a la tarea de mayor prioridad, la cual es la que debería ejecutar si fuera planificada por un algoritmo de planificación en Tiempo Real.

- En caso que llegue una tarea de mayor prioridad para ejecutar, el *planificador* del O/S desalojará a la tarea en ejecución e inmediatamente despachará a la nueva tarea.

De esta manera, se emula el comportamiento del algoritmo deseado. Además, se asumirá que la asignación de prioridades se realiza una sola vez, antes de comenzar la ejecución de cada tarea y luego no se modificará.

Cuando una nueva tarea ingresa al sistema, se deberán conocer los cuatro parámetros analizados en la sección 4.1: tiempo de llegada, a(T); tiempo de listo, s(T); tiempo de ejecución, x(T); y *meta*, d(T). Estos parámetros están relacionados por la siguiente ecuación:

$$d(T) = a(T) + x(T) + s(T).$$

El parámetro x(T) será estimado y sólo se usará en uno de los algoritmos. A su vez, no se incluirán parámetros como solicitudes de E/S, contención de recursos y otros efectos específicos provocados por las aplicaciones.

El objetivo principal del *planificador* es minimizar el número de tareas que pierden sus *metas* (**missed deadlines**). En caso que esto ocurra, se puede optar por las siguientes dos alternativas:

- Abortar la tarea que perdió su *meta*. En este caso, se asume que ya no es necesario que siga ejecutando, dado que la tarea no pudo cumplir con sus objetivos.

- Dejar que la tarea continúe su ejecución. En este caso, se asume que es mejor que continúe con su procesamiento, aunque sus resultados lleguen con retraso.

En este modelo se asumirá la segunda opción, dado que es necesario que las tareas sigan ejecutando, aun cuando pierdan algunas de sus *metas*.

4.2.3 Emulación de diferentes algoritmos.

Como fue mencionado en la sección anterior, para poder implementar la emulación de diferentes algoritmos de planificación en Tiempo Real, se deberán asignar prioridades a las tareas de acuerdo al algoritmo seleccionado.

La técnica de asignación de prioridades, conjuntamente con la política *cliente/servidor* estudiada en la sección 4.2.1, son fundamentales para poder representar la emulación de los algoritmos.

En esta sección, se estudia la técnica de asignación de prioridades para cada algoritmo en particular. Dado que la política de emulación *cliente/servidor* es la misma para todos los algoritmos, será analizada en el capítulo 6, en donde se estudiarán los detalles de implementación.

Para poder asignar una prioridad del O/S a una tarea dada con sus respectivos parámetros de tiempo, se utiliza una función de traducción lineal, la cual depende de las siguientes variables [Ade94]:

- R, un número entero positivo, el cual representa el argumento de entrada a la función de traducción.
- n, un número entero positivo, el cual representa la cantidad de niveles de prioridades provistas por el O/S.
- t_s, un número entero positivo, el cual representa un factor de escala para la función de traducción lineal.

Estas variables están relacionadas por la siguiente ecuación:

$R = n.t_s$.

con lo cual, el rango de R varía entre 0 y $n.t_s$.

A continuación se muestra la función de traducción lineal, escrita en pseudocódigo, la cual recibe como parámetro un valor dentro del rango de R y retorna el nivel de prioridad del O/S correspondiente. El objetivo de esta función es distribuir uniformemente el rango de r, entre los n niveles de prioridades:

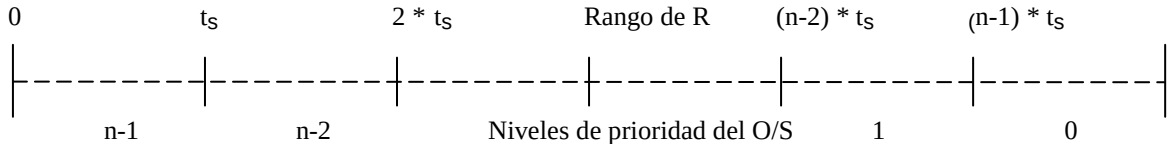
```
int map_prioridad(unsigned long int valor_algoritmo)
{
    unsigned long int    level;
    int                  prioridad;

    level = valor_algoritmo / ts.        // Se distribuye el parámetro de entrada
                                         // entre los n niveles de prioridades.
    if (level >= n)                      // En caso que el nivel sea mayor que n,
        level = n - 1;                   // se asigna el último intervalo de R.

    prioridad = n - 1 - level;           // Se asigna la prioridad de acuerdo a la
                                         // política del O/S, en donde mayor es el
                                         // nivel, mayor es la prioridad.
    return prioridad;
}
```

Esta función es llamada por cada algoritmo de emulación, para calcular la prioridad del O/S que se debe asignar a cada tarea que ingresa al sistema. El parámetro de entrada es la variable que cada algoritmo utiliza para ordenar a las tareas. Mientras más chico sea este valor, mayor será la prioridad que se le asigne cada tarea. Este valor puede ser la *meta* de una tarea, el tiempo de listo o cualquier

otro parámetro que utilice un algoritmo de planificación en Tiempo Real para ordenar a sus tareas. En la siguiente figura se muestra la distribución del rango de R sobre los niveles de prioridades:



A continuación se presentan los algoritmos con los cuales se trabajará en esta Tesis[Ade94]:

- *EDF*, en su versión dinámica con prioridades fijas y con tiempos absolutos (*EDABS*).

- *EDF*, en su versión dinámica con prioridades fijas y con tiempos relativos (*EDREL*).

- *LSF*, en su versión dinámica con prioridades fijas y con tiempos relativos (*LSREL*).

El objetivo de *EDF* es asignar la mayor prioridad a las tareas con *metas* más próximas. El problema de este algoritmo *dinámico*, se presenta cuando comienzan a llegar tareas con *metas* más estrictas que las que están ejecutando en ese momento y la cantidad de niveles de prioridades no es suficiente. Sucede lo mismo en el caso inverso, cuando comienzan a llegar tareas con *metas* muy distantes y tampoco alcanzan los niveles disponibles. El algoritmo *EDABS* plantea una solución a este problema, mientras que los otros dos algoritmos trabajan con tiempos relativos y, por lo tanto, tratan a cada tarea en forma independiente.

El algoritmo *EDABS* utiliza la llegada de la primera tarea para establecer un *busy period*. Este período es un intervalo de tiempo en el cual el sistema está realizando algún trabajo. Los intervalos entre *busy periods* se llaman *idle periods*. Claramente, un sistema con poca carga de CPU, se comportará como una sucesión de intervalos consecutivos de *idle periods*, interrumpidos ocasionalmente por algún *busy period*. En contraste, un sistema con mucha carga de procesamiento, se comportará de forma inversa.

En *EDABS*, el comienzo de un *busy period* se configura asignando el tiempo actual a una variable llamada t_{pinned}. Durante este intervalo, las *metas* de las tareas serán consideradas con respecto a t_{pinned}. En caso que este período sea muy largo, las nuevas tareas que llegan al sistemas tendrán *metas* muy grandes y, por lo tanto, se les asignará prioridades muy bajas. En estas situaciones, pueden quedar tareas relegadas

y posiblemente entren en un estado de *inanición*. Para solucionar este problema, se introduce una cota llamada Nr, que representa la cantidad máxima de tareas que se les asigna la prioridad más baja. En caso que se llegue a este límite, el valor de t_{pinned} se inicializa al tiempo actual. A partir de ese momento, las nuevas tareas comenzarán a ser asignadas a niveles de mayor prioridad. Cuando se llega a esta situación, se dice que ocurrió un *reshift* de prioridades. En estos casos, seguramente algunas de las tareas relegadas comenzarán a perder sus *metas*.

Con el objeto de distinguir realmente un *busy period* prolongado de la llegada inusual de tareas con *metas* distantes, se considerarán sólo las asignaciones a tareas consecutivas.

A continuación se muestra el pseudocódigo del algoritmo *EDABS*:

```
void ejecutar_edabs(tipo_tarea tarea)
{
    unsigned long int    tiempo_llegada, tiempo_meta;
    int                  prioridad;

    if (!pinned) {                      // pinned es una variable estática, que
       pinned = TRUE;                   // inicialmente está en FALSE.
       t_pinned = TiempoActual();       // t_pinned es una variable que se inicializa
       asignaciones = 0;                // con el tiempo actual como comienzo del
    }                                   // busy period.
                                        // asignaciones es una variable estática.
    tiempo_meta = leer_meta(tarea);
    tiempo_llegada = TiempoActual();    //Asigna los parámetros de tiempo de la tarea.

    // Se traduce la prioridad que corresponde a la tarea según EDABS, al nivel de prioridades
    // que maneja el O/S.
    prioridad = map_prioridad(tiempo_meta - t_pinned);

    if (prioridad == prioridad_mas_baja)
       asignaciones++;              // Incrementa el contador de asignaciones.
    else
       asignaciones = 0;            // Se asigna 0, porque las asignaciones deben ser sólo
                                    // consecutivas.

    if (asignaciones == Nr) {       // Si se llega al límite de asignaciones, se
       t_pinned = TiempoActual();   //  inicializa a t_pinned.
       prioridad = map_prioridad(tiempo_meta - t_pinned);
       asignaciones = 0;            // También se recalcula la prioridad de la tarea
    }                               // y se inicializa la variable asignaciones.

    AsignarPrioridad(tarea, prioridad);  // Y por último, se invoca a una función del O/S
}                                        // para asignar la prioridad calculada.
```

La última función de este algoritmo, simplemente llama a una rutina del O/S para asignar la prioridad calculada a la tarea que se quiere planificar. Una vez asignada la prioridad que corresponde, se podrá comenzar la ejecución de la tarea.

El segundo algoritmo, *EDREL*, en lugar de tener en cuenta las prioridades que ya fueron asignadas, trata a cada tarea del sistema en forma independiente. Para ello, calcula la prioridad de cada tarea basándose en su *meta* relativa al tiempo de su llegada.

A continuación se muestra un pseudocódigo del algoritmo *EDREL*:

```
void ejecutar_edrel(tipo_tarea tarea)
{
    unsigned long int    tiempo_llegada, tiempo_meta;
    int                  prioridad;

    tiempo_meta = leer_meta(tarea);
    tiempo_llegada = TiempoActual();    //Asigna los parámetros de tiempo de la tarea.

    // Se traduce la prioridad que corresponde a la tarea según EDREL, al nivel de prioridades
    // que maneja el O/S.
    prioridad = map_prioridad(tiempo_meta - tiempo_llegada);

    AsignarPrioridad(tarea, prioridad);    // Y por último, se invoca a una función del O/S
                                           // para asignar la prioridad calculada.
}
```

Este algoritmo sólo depende de la variable R y no utiliza otra información de las tareas activas. Esta independencia de tareas, libera al *planificador* de crear una escala absoluta de tiempo y prioridades. De esta manera, se permite una planificación descentralizada.

La desventaja de *EDREL* es que puede discriminar a las tareas con *metas* muy distantes, ya que éstas quedarán relegadas ante la llegada de tareas con *metas* más próximas, las cuales obtendrán siempre mayor prioridad.

El último de los algoritmos presentados, *LSREL*, trabaja de la misma manera que *EDREL*. Esto significa que trata a cada tarea en forma independiente y no tiene en cuenta las prioridades asignadas a otras tareas. Para calcular la prioridad de una tarea dada, se basa en el tiempo de listo relativo al tiempo de su llegada.

Para implementar este algoritmo se necesita una estimación de x(T), ya que el tiempo de listo se define como:

$s(T) = (d(T) - a(T) - x(T))$.

A continuación se muestra un pseudocódigo del algoritmo *LSREL*:

```
void ejecutar_lsrel(tipo_tarea tarea)
{
    unsigned long int    tiempo_llegada, tiempo_ejecucion, tiempo_meta;
    int                  prioridad;

    tiempo_ejecucion = leer_ejecucion(tarea);
    tiempo_meta = leer_meta(tarea);
    tiempo_llegada = TiempoActual();    //Asigna los parámetros de tiempo de la tarea.

    // Se traduce la prioridad que corresponde a la tarea según LSREL, al nivel de prioridades
    // que maneja el O/S.
    prioridad = map_prioridad(tiempo_meta - tiempo_llegada - tiempo_ejecucion)

    AsignarPrioridad(tarea, prioridad);    // Y por último, se invoca a una función del O/S
                                           // para asignar la prioridad calculada.
}
```

5 - ClashRT: DISEÑO ORIENTADO A OBJETOS

En este capítulo, se describirá la especificación y el diseño de la herramienta para construir sistemas SCADA, conjuntamente con sus clases y relaciones entre sí, y también se analizará su interface con los diferentes niveles de servicio.

5.1 Especificación de la herramienta.

En esta Tesis, se desarrolla una herramienta para construir sistemas SCADA, basada en el nivel de servicio BSL, el cual es común para todos estos sistemas.

Esta herramienta, llamada ClashRT, se puede ubicar dentro de la categoría Interfaz con un lenguaje, dado que es una biblioteca de módulos comunes a todos los Supervisores y se accede a su interface a través de un lenguaje de programación.

ClashRT provee las facilidades necesarias para construir sistemas SCADA, conjuntamente con los servicios de planificación de alto nivel y la emulación de los algoritmos de planificación en Tiempo Real. A través de estos servicios, el programador puede desarrollar sus propias tareas dentro del proyecto y planificarlas de acuerdo a sus requerimientos de tiempo y al algoritmo seleccionado.

ClashRT está diseñada bajo el paradigma orientado a objetos. La herramienta está construida como una colección de clases relacionadas entre sí y provee el acceso a sus servicios e información a través de los mensajes de acceso público.

La herramienta está desarrollada con la arquitectura *cliente/servidor*, en la cual ClashRT es un *servidor* y los niveles de servicios ISL y PSL actúan como *clientes*. En este modelo, se pueden tener varias instancias de los niveles de servicios ISL y PSL, de acuerdo a cada aplicación en particular, pero siempre se comunican con el mismo servidor BSL. Incluso se podría desarrollar los servicios ISL en otra computadora y acceder a ClashRT a través de rutinas de acceso especiales para manejar la comunicación vía red.

La funcionalidad e información provista por ClashRT se divide en 2 servicios bien diferenciados:

- **Motor de Ejecución.**
- **Tablas de representación.**

El *Motor de Ejecución* está compuesto por el Planificador con sus algoritmos de emulación y por las Tareas de Tiempo Real. Estas tareas son las que proveen los servicios básicos y comunes a todos los sistemas SCADA. Son tareas *periódicas* y son

activadas por el Planificador en cada ciclo de ejecución. A continuación se describen los componentes del *Motor de Ejecución*:

- **Planificador:** administra y controla la ejecución de las Tareas de Tiempo Real. Es el encargado de asignar las prioridades que correspondan a cada tarea, comenzar la ejecución de cada una de ellas y de sincronizar su procesamiento de acuerdo a sus requerimientos de tiempo.

- **Tarea de Alarmas:** analiza las condiciones de alarmas y ejecuta las rutinas de atención implementadas por el programador.

- **Tarea de Comandos:** interpreta y ejecuta en forma planificada los comandos implementados por el programador.

- **Tarea de Histórico:** actualiza el archivo de registro de las operaciones, de acuerdo a parámetros especificados por el usuario. Esta tarea también es la encargada de leer todos los datos que provienen desde el exterior y almacenarlos en las *Tablas de Representación.*

- **Tarea de Mímicos:** despliega los *puntos de supervisión* en la pantalla.

- **Tarea de Ports:** maneja la comunicación entre el sistema SCADA y el mundo exterior.

Para que el Planificador pueda realizar sus funciones de asignación de prioridades, administración y control de tareas, deberá utilizar los mecanismos de sincronización y control de tareas provistos por el O/S. A su vez, cada tarea deberá ser construida para ejecutar sobre este Planificador.

Por otro lado, las *Tablas de Representación* almacenan toda la información necesaria para la operación de los sistemas SCADA:

- **Tabla de Alarmas:** almacena todas las condiciones de alarmas con sus parámetros respectivos.

- **Colas de Entrada/Salida:** almacenan todos los *puntos de supervisión* que se reciben y transmiten desde el SCADA.

- **Tabla de Comandos:** almacena todos los comandos implementados por el programador.

- **Archivo Histórico:** es un archivo de disco, el cual almacena todos los *puntos de supervisión* que fueron salvados de acuerdo a parámetros especificados por el usuario.

- **Tabla de Mímicos:** almacena todos los *puntos de supervisión* que serán desplegados por pantalla, con sus atributos de letra y color. A cada uno de estos puntos se los llama mímico.

- **Tabla de Ports:** almacena los ports de comunicaciones, a través de los cuales se accede al mundo exterior.

- **Tabla de Procesos:** almacena todos los procesos del mundo exterior que se desean modelar. Un proceso es utilizado por el nivel de servicio ISL, para monitorear por pantalla los mímicos que correspondan.

- **Tableaux:** almacena todos los *puntos de supervisión* de la Imagen. Esta tabla está dividida en 3 segmentos de acuerdo al tipo de puntos que almacena: analógicos, digitales y enteros.

- **Tabla de Tareas:** almacena todas las tareas del sistema SCADA, con sus requerimientos de tiempo.

Todas estas *Tablas de Representación* son accedidas por las Tareas de Tiempo Real durante su ejecución. Para coordinar y administrar el acceso concurrente a esta información, se deberá sincronizar el acceso a través de algún mecanismo de comunicación provisto por el O/S.

5.2 Descripción funcional de las Clases.

El *Motor de Ejecución,* las *Tablas de Representación* y las estructuras de datos auxiliares, están encapsuladas en clases. Cada una de ellas, contiene la información y funcionalidad necesaria para su operación. El acceso a los servicios provistos por cada clase, se realiza a través de los mensajes de acceso público.

A continuación se describen todas las clases que componen ClashRT. Las *Tablas de Representación* están diseñadas como colecciones de registros y se encuentran encapsuladas dentro de sus respectivas clases. Con lo cual, cada clase tendrá una sola instancia, con excepción de la clase Cola, la cual tendrá dos instancias, Cola de Entrada y Cola de Salida:

- **Clase Alarma:** tiene la responsabilidad de administrar la Tabla de Alarmas y las relaciones entre Alarma-Tableaux. Dentro de esta clase, está encapsulada la Tarea de Alarmas

- **Clase Algoritmo:** tiene la responsabilidad de administrar los diferentes algoritmos de emulación. Es una clase abstracta y tiene 3 subclases, una por cada algoritmo: Edabs, Edrel, Lsrel.

- **Clase Cola:** tiene la responsabilidad de administrar la estructura Cola. Esta clase tiene dos instancias: Cola de Entrada y Cola de Salida, las cuales son utilizadas por diferentes clases para almacenar los datos que serán recibidos y transmitidos desde el SCADA.

- **Clase Comando:** tiene la responsabilidad de administrar la Tabla de Comandos. Dentro de esta clase, se encuentran las funciones para la ejecución inmediata de comandos, como así también está encapsulada la Tarea de Comandos, la cual ejecuta los comandos en forma planificada.

- **Clase Emulador:** tiene la responsabilidad de administrar la Tabla de Emulador y la relación Emulador-Tableaux. Dentro de esta clase, está encapsulada la Tarea Emulador. Utiliza la Cola de Entrada para almacenar los datos que serán recibidos por el sistema SCADA. Esta clase es utilizada para realizar pruebas de la herramienta, simulando la entrada de datos desde el exterior.

- **Clase Histórico:** tiene la responsabilidad de administrar el acceso al archivo Histórico. Dentro de esta clase, está encapsulada la Tarea de Histórico. Utiliza la Cola de Entrada para leer los datos que fueron recibidos desde el exterior.

- **Clase Mensaje:** tiene la responsabilidad de administrar la estructura de los mensajes.

- **Clase Mímico:** tiene la responsabilidad de administrar la Tabla de Mímicos y la relación Mímico-Tableaux-Proceso. Dentro de esta clase, está encapsulada la Tarea de Mímicos. Utiliza la clase Pipe y la clase Mensaje para realizar la comunicación con el nivel de servicio ISL.

- **Clase Pipe:** tiene la responsabilidad de administrar el pipe de comunicaciones que se utiliza entre el BSL y el ISL.

- **Clase Planificador:** tiene la responsabilidad de administrar las funciones del Planificador. Esta clase utiliza la clase Algoritmo y la clase Tarea, las cuales son necesarias para poder realizar sus funciones de administración y control de tareas.

- **Clase Port:** tiene la responsabilidad de administrar la Tabla de Ports y la relación Port-Tableaux. Dentro de esta clase, está encapsulada la Tarea de Ports Utiliza las 2 instancias de la clase Cola: la Cola de Entrada para almacenar los datos que serán recibidos por el sistema SCADA y la Cola de Salida para almacenar los datos que serán transmitidos hacia el exterior.

- **Clase Proceso:** tiene la responsabilidad de administrar la Tabla de Procesos.

- **Clase Tableaux:** tiene la responsabilidad de administrar los 3 tipos de Tableaux. Es una clase abstracta y tiene 3 subclases, una por cada tipo de *puntos de supervisión* que almacena: analógicos, digitales y enteros.

- **Clase Tarea:** tiene la responsabilidad de administrar la Tabla de Tareas.

5.3 Diagrama de Clases.

A continuación se muestra la estructura de las clases principales y sus relaciones entre sí, diseñada con el lenguaje *UML (Unified Modeling Language)*, el cual es específico para el diseño orientado a objetos:

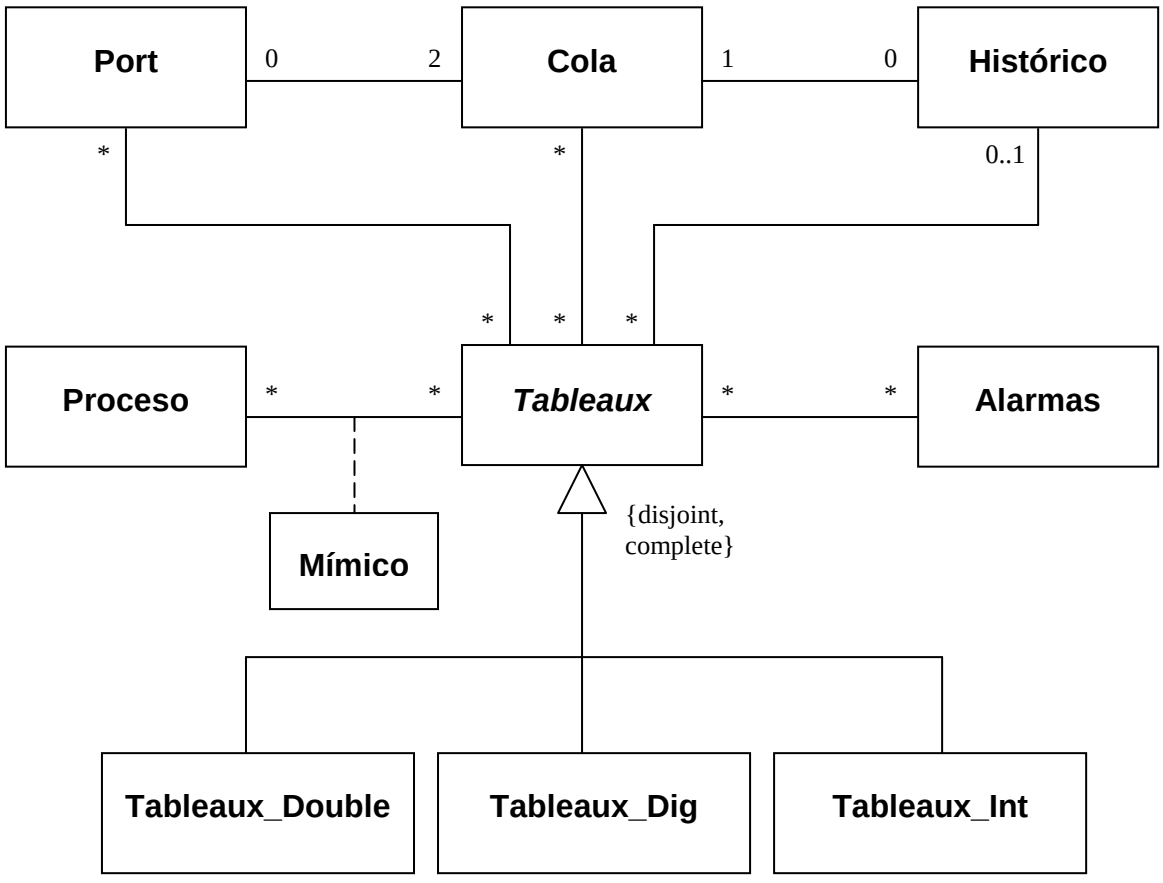


Figura 7 - Diagrama de Estructuras I de ClashRT.

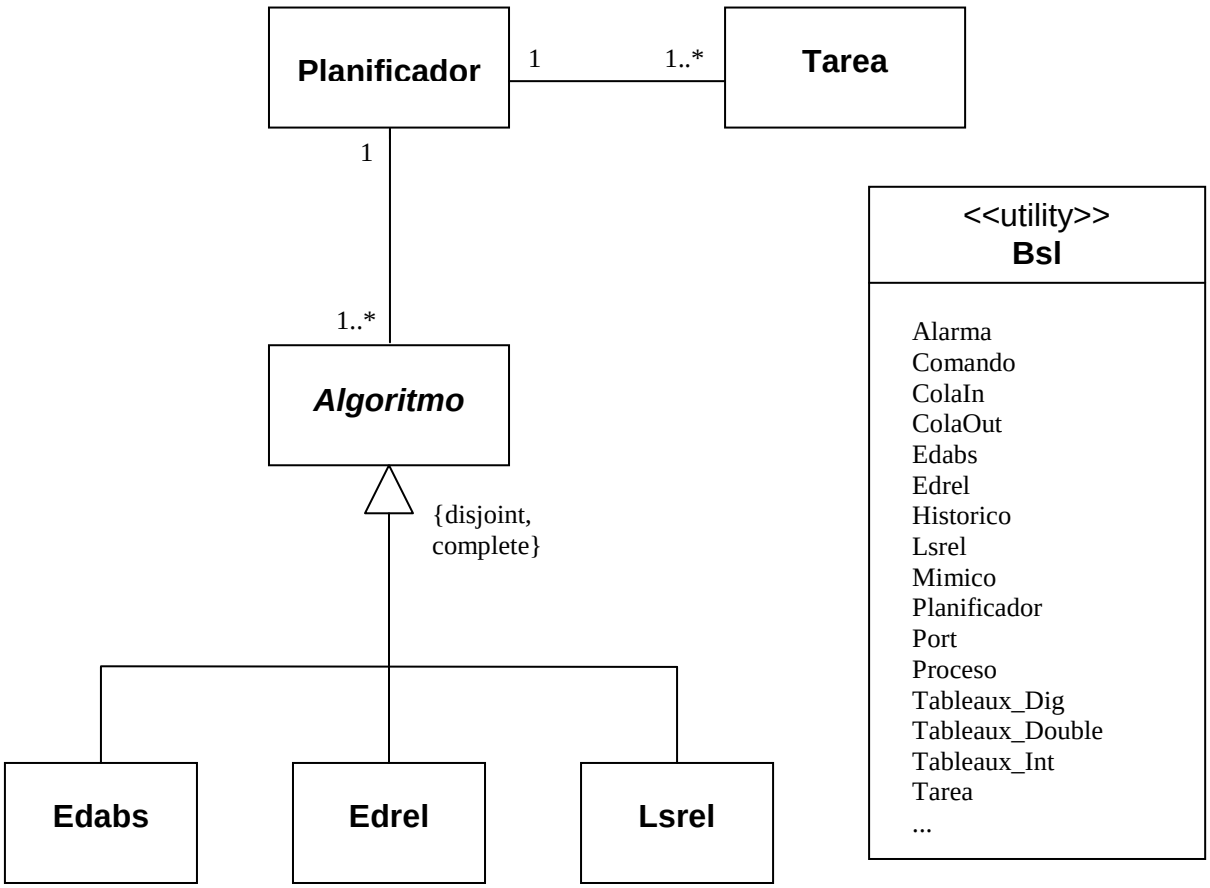


Figura 8 - Diagrama de Estructuras II de ClashRT.

5.4 Interface con los diferentes niveles.

Todas las clases de ClashRT están encapsuladas en la **Clase Bsl**. A través de esta clase, se acceden a todos los servicios provistos por cada una de las clases que la componen.

Como se mencionó anteriormente, los niveles de servicio ISL y PSL, son diferentes de acuerdo a cada sistema SCADA en particular. Estos servicios se desarrollan de acuerdo a los requerimientos funcionales de cada aplicación específica.

El nivel de servicio ISL, es el encargado de proveer todos los servicios relacionados con la interface gráfica y la interacción con el usuario. Este nivel se puede dividir en los siguientes módulos de acuerdo a su funcionalidad:

- **Módulos para la administración del sistema SCADA:** estas rutinas incluyen las tareas de administración y configuración del sistema, como arrancar y parar el *Motor de Ejecución*, establecer los parámetros para la emulación en Tiempo Real y especificar los requerimientos de tiempo de las tareas.

- **Módulos para la manipulación de los datos:** estas rutinas incluyen las ventanas para la actualización de las *Tablas de Representación* y sus relaciones entre sí.

- **Módulos para la visualización de los datos:** estas rutinas incluyen las funciones para la actualización de ventanas y despliegue de datos y mímicos en pantalla.

La comunicación entre ClashRT y el nivel de servicio ISL se realiza a través de las siguientes interfaces:

- Mensajes de acceso público de la *clase Bsl*.

- Mensajes de acceso público de cada clase que integran la *clase Bsl*.

- La clase Pipe conjuntamente con la clase Mensaje y/o a través de algún mecanismo de comunicación provisto por el O/S.

A su vez, el nivel de servicio PSL es el encargado de proveer todos los servicios específicos de cada sistema SCADA. En general, estas tareas se pueden dividir en los siguientes módulos:

- **Módulos de atención de alarmas:** estas funciones son implementadas por el programador de acuerdo con los requerimientos específicos de cada aplicación y de la severidad de cada alarma. Los ejemplos más comunes son: solicitar al operador el reconocimiento de la alarma, generar un sonido de alarma, informar al operador mediante una ventana de advertencia, etc.

- **Módulos de atención de comandos:** estas funciones son codificadas por el programador, de acuerdo a los requerimientos de la aplicación. Algunos ejemplos pueden ser: mostrar totales de alarmas, incrementar la frecuencia de display de los mímicos, etc.

La comunicación entre ClashRT y el nivel de servicio PSL se realiza a través de las clases involucradas:

- **Clase Alarma:** al agregar una alarma en la Tabla de Alarmas, se puede especificar la rutina de atención que corresponda. Esta función se ejecutará

automáticamente cuando se cumpla la condición de alarma y recibirá una estructura de parámetros con toda la información necesaria para procesar la alarma.

- **Clase Comando:** al agregar un comando en la Tabla de Comandos, se debe especificar la rutina de atención que corresponda. Esta función se ejecutará inmediatamente o en forma planificada, según como se haya ingresado, y recibirá una estructura de parámetros con toda la información necesaria para procesar el comando

5.5 Selección del lenguaje de programación.

La herramienta ClashRT está desarrollada en el lenguaje de programación C++. Este lenguaje es de propósito general y fue desarrollado sobre su antecesor, el lenguaje C. Salvo algunas excepciones, se puede decir que C++ incluye a C.

Por un lado, C++ tiene toda la funcionalidad para realizar tareas de bajo nivel, ya que incluye al lenguaje C y, a su vez, es un lenguaje de alto nivel, ya que está orientado a objetos y provee todas características de abstracción necesarias para el desarrollo de aplicaciones.

El concepto clave en C++ es la **clase**. Básicamente, una *clase* es un tipo de datos definido por el usuario. A través de las *clases* de C++, se proveen los siguientes mecanismos y facilidades de programación:

- Ocultamiento de la información.
- Encapsulamiento.
- Inicialización garantizada de los datos.
- Jerarquías de clases, lo cual facilita la abstracción de comportamiento e información.
- Administración de memoria controlada por el usuario.
- Conversiones de tipos implícitos.
- Sobrecarga de operadores.
- Polimorfismo.

Un tema muy importante para el desarrollo de ClashRT, es la interface con el O/S. Como se mencionó en el capítulo 3, la plataforma elegida para la implementación de la herramienta es Win32 y su familia de Sistemas Operativos.

Para acceder a los servicios de cada O/S, se debe invocar alguna función de la Win32 API. Esta capa de funciones y estructuras de datos, está escrita en el lenguaje C. Esto facilita mucho el acceso a sus servicios, ya que desde C++ resulta natural invocar a funciones y acceder a datos con sintaxis C.

A su vez, para permitir portabilidad con otros Sistemas Operativos, ClashRT está implementada en forma abierta y compatible con el ANSI C++ [Str91].

En el próximo capítulo se describirán los detalles de implementación en Win32 y, en algunos casos, se ilustrará con ejemplos escritos en C++.

6 - ClashRT: IMPLEMENTACIÓN EN WIN32

En este capítulo, se describirá la implementación de ClashRT en la plataforma Win32 y en su familia de Sistemas Operativos. A su vez, se estudiarán las ventajas de cada O/S para la emulación de la planificación en Tiempo Real y se analizará la portabilidad de la herramienta a otros Sistemas Operativos.

ClashRT está implementada para ejecutar en toda la familia de Sistemas Operativos de la plataforma Win32. Si bien cada uno de ellos provee diferentes capacidades y servicios, desde el punto de vista del programador, se pueden desarrollar aplicaciones para que ejecuten en todos los Sistemas Operativos y, a su vez, obtener los beneficios propios de cada uno de ellos.

En las próximas secciones se describirán los detalles de implementación en forma genérica para toda la familia de los Sistemas Operativos de Win32 y, cuando corresponda, se especificarán las funciones que son únicas para un sistema en particular.

6.1 Estructura del Planificador.

Como fue descripto en el capítulo anterior, todas las clases de ClashRT están encapsuladas en la **Clase Bsl**. A través de esta clase, se acceden a todos los servicios provistos por cada una de las clases que la componen. Esta clase tiene una sola instancia, el objeto ClashRT, a partir del cual se accede al nivel de servicio BSL.

La funcionalidad e información de la herramienta se divide en 2 servicios bien diferenciados: el *Motor de Ejecución* y las *Tablas de Representación*. A su vez, el *Motor de Ejecución* está compuesto por el Planificador con sus algoritmos de emulación y por las Tareas de Tiempo Real.

En ClashRT, tanto el Planificador como las Tareas de Tiempo Real, están implementadas como threads de Win32, dentro de un proceso del O/S.

Para ejecutar y detener al *Motor de Ejecución*, se debe enviar al objeto ClashRT, los mensajes *EjecutarMotorDeEjecucion()* y *DetenerMotorDeEjecucion()* respectivamente.

Cuando se ejecuta el *Motor de Ejecución*, se realizan los siguientes pasos:

- Se realiza la configuración de los parámetros de Tiempo Real.
- Se crea el thread del Planificador con su algoritmo de emulación y el conjunto de tareas asociados.

- El Planificador comienza su ejecución.
- El Planificador empieza a crear y planificar a todos los threads de Tiempo Real que están definidos en el conjunto de tareas.
- Los Tareas de Tiempo Real comienzan su ejecución.
- A partir de este momento, tanto el Planificador como las Tareas de Tiempo Real, se encuentran ejecutando sus respectivos ciclos de ejecución de tiempo infinito.

Para crear un thread en Win32, se puede utilizar la función de la Win32 API *CreateThread(..)* o una función más eficiente de la biblioteca CRT (C Run Time Libraries), llamada *_beginthreadex(..)*, la cual encapsula a la primera. Ambas funciones retornan un valor de tipo HANDLE, el cual será utilizado posteriormente para controlar al thread creado. En forma análoga, para terminar un thread, se puede llamar a *ExitThread(..)* o a *_endthreadex(..)*.

Para coordinar y sincronizar la ejecución de los threads de ClashRT, el Planificador y las tareas utilizan los objetos *Event*, y sus funciones asociadas: *SetEvent(..)* y *ResetEvent(..)*.

Como fue estudiado en el capítulo 4, en esta Tesis se implementa la técnica de emulación *cliente/servidor*, en combinación con la asignación manual de prioridades, de acuerdo con un algoritmo de planificación en Tiempo Real.

La estructura del Planificador es como la de un *cliente*, el cual realiza sus pedidos en forma cíclica, de acuerdo a los períodos de los tareas definidas. Su procesamiento consiste, básicamente, en solicitar a cada tarea que ejecute su ciclo de ejecución. En caso que no tenga que señalizar a ninguna tarea, simplemente quedará inactivo a través de una llamada a *Sleep(..)*. El Planificador está implementado de forma óptima, en el sentido que quedará inactivo el máximo tiempo posible hasta que tenga que planificar por lo menos a una tarea.

Todo este comportamiento está encapsulado en la clase **Planificador**. A través de los siguientes 2 métodos, el Planificador realiza sus funciones principales:

- **planificador::planificar_tarea**.
- **planificador::planificar**.

El primero de ellos, *planificar_tarea(..)*, es el que se invoca para planificar cada tarea que ingresa al sistema. Su función principal es crear el thread para ejecutar la tarea y asignar el nivel de prioridad del O/S que corresponda:

```
HANDLE planificador::planificar_tarea(char* nombre_tarea,
                                       PFNTHREAD funcion_tarea,
                                       void* parametros_tarea,
                                       unsigned long int tiempo_periodo,
                                       unsigned long int tiempo_ejecucion)
{
    HANDLE        evento_handle, thread;
    char          nombre_evento[20] = "EVENT_";
    unsigned int  thread_id;
    int           indice ;

    // Se crea el objeto Event para controlar a la tarea que se va a planificar.
    evento_handle = CreateEvent(NULL, TRUE, FALSE, strcat(nombre_evento,
                                nombre_tarea));

    // Se crea la tarea en un thread del proceso, la cual queda bloqueada a la espera de la señal
    // del Planificador para comenzar su ejecución.
    thread = (HANDLE)_beginthreadex(NULL, 0, funcion_tarea,
                                    parametros_tarea, 0, &thread_id);

    // Se agrega la tarea a la lista de Tareas asociadas al Planificador.
    indice = tareas->agregar_tarea(nombre_tarea, (tareas->cant_tareas() + 1), 0,
                                   tiempo_periodo, tiempo_ejecucion, 0, 0,
                                   evento_handle);

    // Se configura el thread para el algoritmo de emulación asociado y se despierta al thread.
    algoritmo_emulacion->ejecutar(tareas, indice, thread, evento_handle);

    // Retorna el HANDLE del thread creado.
    return thread;
}
```

El segundo método, *planificar()*, es el que se ejecuta en el thread del Planificador y es el que controla y sincroniza la ejecución de todos los threads de la herramienta. Este método es donde se implementa la estructura de *cliente* del Planificador y es el verdadero motor de ClashRT:

```
void planificador::planificar()
{
    HANDLE              evento_handle;
    unsigned long int   antes, ahora, tiempo_max_sleep, tiempo_periodo,
                        tiempo_transcurrido, tiempo_esperado;
    char                nombre_tarea[12];

    // Se asigna la cantidad de milisegundos transcurridos desde el inicio del sistema, para
    // poder contar el transcurso del tiempo.
```

```
antes = GetTickCount();

// La variable continuar es un atributo de Planificador y está inicializado en TRUE.
while (continuar) {

    // Se calcula la cantidad máxima de milisegundos que el Planificador deberá quedar
    // inactivo hasta que tenga que planificar al menos un thread.
    tiempo_max_sleep = calcular_max_sleep();
    ahora = GetTickCount();
    tiempo_transcurrido = DeltaTime(ahora, antes);

    if (tiempo_max_sleep > tiempo_transcurrido) {
        tiempo_max_sleep -= tiempo_transcurrido;
        Sleep(tiempo_max_sleep);
    }

    // Se calcula nuevamente el transcurso del tiempo para analizar que threads deberán ser
    // planificados.
    ahora = GetTickCount();
    tiempo_transcurrido = DeltaTime(ahora, antes);
    antes = ahora;

    for (int indice=0; indice<tareas->cant_tareas(); indice++) {

        tareas->leer_datos(indice, nombre_tarea, tiempo_periodo,
                                    tiempo_esperado, evento_handle);

        // Para cada uno de los threads, se verifica si se debe comenzar su ciclo de
        // ejecución.

        if ((tiempo_transcurrido + tiempo_esperado) >= tiempo_periodo) {

            // Se inicializa el tiempo esperado del thread en 0.
            tareas->asignar_tiempo_esperado(indice, 0);

            // Se despierta al thread bloqueado.
            SetEvent(evento_handle);

        } else {

            // Se incrementa el tiempo esperado del thread.
            tiempo_esperado += tiempo_transcurrido;
            tareas->asignar_tiempo_esperado(indice, tiempo_esperado);
        }
    }
}
```

```
        // Antes de finalizar, se realiza un Set Event a todos los threads para asegurar que no quede
        // ninguno bloqueado.
        ……
}
```

Estos dos métodos están implementados con algunas sentencias para generar estadísticas de *missed deadlines* y para escribir en el Log File del sistema, con el objeto de tener un control exacto de la operación del Planificador. En esta sección, se omitieron estas sentencias con el sólo propósito de hacer más legible el código.

6.2 Estructura de las Tareas.

Las Tareas de Tiempo Real proveen los servicios básicos y comunes a todos los sistemas SCADA. Son tareas *periódicas* y son activadas por el Planificador en cada ciclo de ejecución.

Todas estas tareas están estructuradas de la misma manera. Esto es así, ya que deben ser sincronizadas y controladas por el Planificador.

La estructura de cada tarea es como la de un *servidor*. Esto significa, que inicialmente se encuentra bloqueada a la espera de una solicitud de un *cliente*, en este caso, el Planificador. Una vez que recibe la señal correspondiente, procesa su ciclo de ejecución y se vuelve a bloquear a la espera de la próxima señalización. Por este comportamiento, se dice que la tarea ejecuta indefinidamente o que es de tiempo infinito.

Si bien las Tareas de Tiempo Real del BSL están predefinidas y encapsuladas dentro de cada clase respectiva, el programador puede agregar todas las tareas que considere necesarias para su aplicación, respetando la estructura común a todas ellas.

Para crear y planificar una tarea, se deberá proveer la siguiente información:

- Un nombre de la tarea.
- Una función de tipo PFNTHREAD, de acuerdo a los requerimientos de la Win32 API. Esta función es la que se pasa como parámetro en la llamada a *_beginthreadex*(..).
- Los parámetros que serán pasados a la función anterior.
- Una **clase** en donde se encapsulen los métodos que se describen a continuación:
    - **nombreclase::ejecutar**, el cual se ejecuta en el thread de la tarea.

- **nombreclase::detener**, el cual se utiliza para detener el thread.

- El período o *meta* de la tarea.

- El tiempo de ejecución de la tarea. Este parámetro es usado sólo cuando se utiliza el algorimto *LSREL*.

Estos dos últimos parámetros de tiempo, conjuntamente con el nombre de la tarea, se almacenan en la Tabla de Tareas.

La función de tipo PFNTHREAD, requerida por la Win32 API para crear un thread, deberá estar estructurada de la siguiente manera:

```
unsigned int __stdcall NombreTarea(void* parametros_tarea)
{
    nombreclase*    objeto_nombreclase;

    // Se procesan los parametros_tarea, de los cuales uno de ellos es objeto_nombreclase .
    .....

    // Se invoca al método ejecutar(), el cual realiza todo el procesamiento de la tarea.
    objeto_nombreclase->ejecutar();

    // Cuando se retorna de ejecutar(), se termina el thread en ejecución.
    _endthreadex(0);

    return(0);
}
```

Una vez creado el thread de la tarea, se ejecuta el método *ejecutar()*, en el cual se implementa la estructura de *servidor* de la tarea y en donde se realiza todo el procesamiento correspondiente:

```
void nombreclase::ejecutar()
{
    HANDLE              evento = 0;
    unsigned long int   antes = 0, ahora = 0, tiempo_transcurrido;

    // La variable continuar es un atributo de nombreclase y se inicializa en TRUE.
    continuar = TRUE;

    // Se abre el evento, el cual fue creado por el Planificador.
    evento = OpenEvent(EVENT_ALL_ACCESS, FALSE,
                       "EVENT_NOMBRETAREA");
```

```
    while (continuar) {

        // Espera por la señalización del Planificador.
        WaitForSingleObject(evento, INFINITE);

        // Modifica el evento al estado non-signaled.
        ResetEvent(evento);

        // Se asigna la cantidad de milisegundos transcurridos desde el inicio del sistema,
        // para poder contar el transcurso del tiempo.
        antes = GetTickCount();

        // En este lugar se realiza todo el procesamiento correspondiente al thread.
        ......

        // Se incrementa el atributo periodo_tarea, el cual es utilizado con fines estadísticos.
        periodo_tarea++;

        // En caso que se necesite, se calcula el tiempo transcurrido de cada ciclo de
        // ejecución.
        ahora = GetTickCount();
        tiempo_transcurrido = DeltaTime(ahora, antes);
    }

    // Se cierra el objeto evento, el cual es borrado del O/S.
    CloseHandle(evento);
}
```

Este método está implementado con algunas sentencias para generar estadísticas de *missed deadlines* y para escribir en el Log File del sistema, con el objeto de tener un control exacto de la ejecución de la tarea. En esta sección, se omitieron estas sentencias con el sólo propósito de hacer más legible el código.

Y por último, se muestra el método *detener()*, el cual solamente asigna FALSE al atributo continuar, lo que provoca que el thread ejecute su último ciclo:

```
void nombreclase::detener()
{
    continuar = FALSE;
}
```

6.3 Tablas de Representación.

ClashRT almacena cada *Tabla de Representación* en arreglos de n elementos. El tamaño de cada tabla se asigna en el momento de inicialización de cada objeto. Las únicas excepciones son las Colas de Entrada y Salida, las cuales están implementadas

como listas encadenadas y el Archivo Histórico que se almacena en disco. El límite de estas tablas está dado sólo por los recursos disponibles del O/S.

A su vez, cada *Tabla de Representación* está encapsulada dentro de la clase correspondiente. Es por este motivo, que cada clase de C++ tiene una sola instancia, con la excepción de la clase Cola, la cual tiene dos instancias: Cola de Entrada y Cola de Salida. Por otro lado, las relaciones entre clases también se almacenan como arreglos de m elementos. El tamaño de estas relaciones es asignado al momento de la creación de los objetos respectivos y su límite está dado sólo por los recursos disponibles del O/S.

Las razones principales para utilizar estructuras de datos estáticas son las siguientes:

- En la mayoría de los casos, se conoce en tiempo de programación, la cantidad de elementos que se van a representar. El usuario de la herramienta, el programador, sabe que cantidad de alarmas se van a definir, cuáles son los *puntos de supervisión* que se necesitarán, etc. En el caso de las Colas, la situación es diferente, ya que no se conoce de forma anticipada cuántos elementos se recibirán o transmitirán en un momento dado. Lo mismo ocurre con el Archivo Histórico, cuya frecuencia de actualización es configurada por el usuario.

- Optimizar los tiempos de ejecución de la herramienta, ya que agregar o eliminar un elemento de una tabla, se realiza de forma más rápida y eficiente que utilizando estructuras dinámicas.

- El costo de desarrollo, debugging y mantenimiento de la herramienta es bastante menor que si se implementan estructuras dinámicas.

Dado que las *Tablas de Representación* van a ser accedidas concurrentemente por el *Motor de Ejecución* y por los niveles de servicio ISL y PSL, se necesita utilizar mecanismos de comunicación y sincronización adecuados para garantizar la consistencia de los datos.

En general, la aplicación que utilice la herramienta será, desde el punto de vista del O/S, un sólo proceso con múltiples threads. Para realizar la comunicación entre todos los threads, se utiliza el mecanismo de *Variables Globales*. En realidad, el único objeto global, es ClashRT, la única instancia de la *clase Bsl*, la cual contiene a cada instancia de cada clase. A su vez, como todos los threads comparten el mismo espacio de direccionamiento, la *heap memory* también es compartida por todos ellos. Por lo tanto, cualquier solicitud y asignación de memoria dinámica también será válida para todos los threads del proceso.

Para sincronizar el acceso a las *Tablas de Representación*, se utiliza el mecanismo de *Critical Sections*, ya que es el más eficiente cuando se implementa un proceso con múltiples threads. Cada clase que tiene la responsabilidad de administrar una *Tabla de Representación*, deberá implementar los siguientes pasos:

1. Definir una variable global de tipo CRITICAL_SECTION, requerida por la Win32 API para utilizar este mecanismo.

   ```
   CRITICAL_SECTION        NombreClase;
   ```

2. Agregar la sentencia de la Win32 API para inicializar la Sección Crítica, dentro del método Bsl::EjecutarBsl():

   ```
   InitializeCriticalSection(&NombreClaseCriticalSection);
   ```

3. Definir los siguientes 2 métodos en nombreclase:

   ```
   void nombreclase::entrar_critical_section()
   {
       // Llamada a la Win32 API para entrar en la Sección Crítica.
       EnterCriticalSection(&NombreClaseCriticalSection);
   }

   void nombreclase::salir_critical_section()
   {
       // Llamada a la Win32 API para salir de la Sección Crítica.
       LeaveCriticalSection(&NombreClaseCriticalSection);
   }
   ```

4. Agregar la sentencia de la Win32 API para borrar la Critical Section, dentro del método Bsl::DetenerBsl():

   ```
   DeleteCriticalSection(&NombreClaseCriticalSection);
   ```

5. Y por último, acceder a la *Tabla de Representación* dentro de cada método de nombreclase, de la siguiente manera:

   ```
   .. nombreclase::metodo(..)
   {
       ......
       entrar_critical_section();
       accede a la TDF;
       salir_critical_section();
       ......
   }
   ```

6.4 Asignación de prioridades.

En esta sección se describen los detalles de implementación de la técnica de asignación de prioridades de acuerdo a cada algoritmo de emulación y a cada O/S de la plataforma Win32.

Para implementar estos algoritmos de asignación de prioridades, se utilizan dos clases fundamentales: Algoritmo y Tarea.

La clase **Tarea** esencialmente almacena y administra la Tabla de Tareas, la cual contiene los siguientes parámetros, imprescindibles para implementar esta técnica y para tener un control de las estadísticas de las tareas:

- **Tiempo_llegada:** representa el tiempo absoluto de la llegada de la tarea al sistema, expresado en milisegundos.

- **Tiempo_periodo:** representa la *meta* de la tarea, expresado en milisegundos.

- **Tiempo_ejecución:** representa el tiempo de ejecución estimado de la tarea, expresado en milisegundos. Este parámetro es usado sólo cuando se utiliza el algorimto *LSREL*.

- **Tiempo_esperado:** representa el tiempo que la tarea se encuentra esperando para su próximo ciclo de ejecución, expresado en milisegundos.

- **Ciclos_señalizados:** representa un contador de la cantidad de veces que la tarea fue señalizada por el Planificador.

- **Ciclos_ejecutados:** representa un contador de la cantidad de períodos que la tarea ha ejecutado.

- **Evento_handle:** representa el HANDLE del O/S al thread de la tarea.

En ClashRT se implementa una función llamada *CurrentTime()*, la cual retorna la cantidad de milisegundos transcurridos desde que se inició el *Motor de Ejecución*. Esta función es muy importante para establecer un referente absoluto de tiempo.

Por otro lado, la clase **Algoritmo** encapsula el comportamiento de cualquier algoritmo de emulación. Es una clase abstracta y es responsable de la funcionalidad común a todos ellos. A su vez, tiene 3 subclases, una por cada algoritmo: Edabs, Edrel, Lsrel.

En la clase Algoritmo, se podría realizar una generalización más estricta y clasificarla en otras dos subclases abstractas: **Estático** y **Dinámico**, las cuales

representan los dos tipos de algoritmos. En este diseño, cada algoritmo en particular sería una subclase concreta de la clase que corresponda.

6.4.1 Implementación de los algoritmos.

Como fue estudiado en el capítulo 4, los algoritmos de estudio se basan en los parámetros R, n y t_s. Estos parámetros son atributos de la clase Algoritmo y son inicializados cuando se arranca el *Motor de Ejecución*.

La implementación de la función de traducción y de los algoritmos de emulación es muy similar al pseudocódigo presentado en el capítulo 4. Pero hay algunas diferencias de implementación, las cuales se describen a continuación:

- Los niveles de prioridades generalmente no son consecutivos, con lo cual, se utiliza el atributo priority_levels, el cual es un arreglo de n elementos y contiene en cada elemento el verdadero valor de las prioridades del O/S.

- El tiempo de llegada de una tarea, es un valor absoluto del tiempo, en referencia al inicio del *Motor de Ejecución*. En cambio, la *meta* de la tarea está expresado como período y es un valor relativo. Para poder aplicar la técnica de los algoritmos, se debe normalizar el período a un valor absoluto.

- La asignación de la prioridad calculada, se realiza a través de la llamada a *SetThreadPriority(..)*.

- Una vez asignada la prioridad, se envía una señal de sincronización para despertar al thread de la tarea, para que comience su ejecución.

Cada algoritmo esta implementado en el método **algoritmo::ejecutar**. El Planificador invoca a este método para que se asigne la prioridad correspondiente y se despierte al thread.

A modo de ejemplo, a continuación se muestra el código de *ejecutar(..)* de la clase Edrel:

```
void edrel::ejecutar(tarea* tareas, int indice, HANDLE thread, HANDLE
                     evento_handle)
{
    unsigned long int       tiempo_llegada , tiempo_periodo;
    int                     prioridad;

    //Asigna los parámetros de tiempo de la tarea.
    tiempo_periodo = tareas->leer_tpo_periodo(indice);
    tiempo_llegada = CurrentTime();
```

```
        tareas->asignar_tpo_llegada(indice, tiempo_llegada);

        //Normaliza el período de la tarea.
        tiempo_periodo += CurrentTime();

        // Se traduce la prioridad que corresponde a la tarea según EDREL, al nivel de prioridades
        // que maneja el O/S.
        prioridad = map_prioridad(tiempo_periodo - tiempo_llegada);

        // Se invoca a una función del O/S para asignar la prioridad calculada.
        SetThreadPriority(thread, prioridad);

        // Se señaliza al thread, para que comience su ejecución.
        SetEvent(evento_handle);
    }
```

Este método está implementado con algunas sentencias para generar estadísticas de *missed deadlines* y para escribir en el Log File del sistema, con el objeto de tener un control exacto de la ejecución de la tarea. En esta sección, se omitieron estas sentencias con el sólo propósito de hacer más legible el código.

6.4.2 Windows2000/NT vs. Windows98/95.

Para implementar la técnica de asignación de prioridades en un GPOS, como lo es cada O/S de la plataforma Win32, se deberá elegir un rango de prioridades que comúnmente no es utilizado por las aplicaciones de propósito general. Es por ello, que ClashRT se ejecuta solamente en las siguientes dos *priority classes*:

- REALTIME_PRIORITY_CLASS.
- HIGH_PRIORITY_CLASS.

La *priority class* REALTIME_PRIORITY_CLASS, como su nombre lo indica, es específica para Sistemas de Tiempo Real y también es compartida con los *kernel threads* del O/S. En general, para obtener el mejor rendimiento posible, se sugiere la siguiente configuración para cada O/S:

- En **Windows2000/NT** se recomienda **REALTIME_PRIORITY_CLASS**. Si bien este rango de prioridades es compartido con los threads del O/S, la robustez de estos Sistemas Operativos hace posible la utilización de esta *priority class* sin que impacte en la performance del sistema.
- En **Windows98/95** se recomienda **HIGH_PRIORITY_CLASS**. En este rango de prioridades, ClashRT funciona correctamente y, en general, las aplicaciones de propósito general no utilizan esta *priority class*, sino que

están configuradas para ejecutar en el rango de prioridades de NORMAL_PRIORITY_CLASS. En el caso que se requiera configurar para REALTIME_PRIORITY_CLASS, se evidenciará un deterioro notable en la respuesta del O/S.

Como fue estudiado en el capítulo 3, existen algunas diferencias importantes con respecto a los niveles de prioridades disponibles en cada *priority class*:

- REALTIME_PRIORITY_CLASS: esta *priority class* es la misma para WindowsNT/98/95. La diferencia está en Windows2000, el cual dispone de los 16 niveles de prioridades.

- HIGH_PRIORITY_CLASS: esta *priority class* es igual para todos los sistemas.

A su vez, Windows2000/NT proveen la posibilidad de deshabilitar el mecanismo de ajuste dinámico de prioridades que se realiza en HIGH_PRIORITY_CLASS, por medio de una llamada a *SetProcessPriorityBoost(..)*.

Para realizar toda esta configuración, la clase Algoritmo provee el método **algoritmo::config_real_time**, el cual implementa los siguientes pasos:

- De acuerdo a la *priority class* elegida, configura al proceso y a todos sus threads para que ejecuten dentro de esa clase.

- Detecta automáticamente en que O/S está ejecutando ClashRT y asigna los niveles de prioridades que corresponden al atributo priority_levels.

- De acuerdo al O/S y a la *priority class* elegida, inicializa los valores de n y t_s.

- En caso que el O/S sea Windows2000/NT, se deshabilita el ajuste automático de prioridades.

Esta configuración se realiza al inicio del *Motor de Ejecución*, de acuerdo a los parámetros seleccionados.

Con el objetivo de obtener un control exacto y preciso de la ejecución de todas las tareas, se le asigna al thread del Planificador el nivel más alto de prioridad respecto a la *priority class* elegida: THREAD_PRIORITY_TIME_CRITICAL.

A su vez, a los threads que corresponden al ISL y PSL, se les asigna la mínima prioridad dentro de la *priority class*, para que no interfieran con las actividades de Tiempo Real.

6.5 Portabilidad a otros Sistemas Operativos.

Para analizar la portabilidad de la herramienta desarrollada en esta Tesis, se debe estudiar su dependencia con la plataforma de ejecución subyacente.

ClashRT encapsula el nivel de servicio BSL de los sistemas SCADA, en donde muchas de sus funciones dependen fuertemente del Sistema Operativo elegido para su desarrollo. Es por ello, que en cada sección en particular, se especifican las funciones dependientes del O/S.

Esta fuerte interrelación entre el nivel de servicio BSL y el O/S se extiende a todos los Sistemas de Tiempo Real en general. Esto es así, ya que estos sistemas utilizan al máximo las capacidades que brinda el *kernel* del O/S:

- Desarrollo de drivers para la atención de interrupciones de dispositivos.
- Creación y destrucción de procesos y/o threads en tiempo de ejecución.
- Asignación de prioridades de acuerdo a la arquitectura de prioridades disponible.
- Utilización de los mecanismos de comunicación, control y sincronización de tareas.

Para desarrollar aplicaciones que utilizan estos servicios, el programador deberá estudiar en detalle las capacidades provistas por el O/S y las técnicas recomendadas para su implementación. Si bien el concepto de estas facilidades y servicios son comunes a todos los Sistemas Operativos, su implementación varía en cada uno de ellos.

Para simplificar el proceso de migración de la herramienta de una plataforma a otra, se deberá considerar los siguientes aspectos:

- La herramienta está desarrollada en ANSI C++ [Str91], lo cual favorece notablemente su portabilidad.
- Muchas de las funciones del O/S fueron encapsuladas en métodos propios de cada clase.
- Todos los métodos que llamen por lo menos a una función del O/S están definidos con la palabra reservada **virtual**.

En caso que se desee portar la herramienta, simplemente se deberán revisar todos los métodos definidos como *virtual*, los cuales seguramente habrá que

reescribirlos, de acuerdo a los servicios y funciones que provea el O/S al cual se quiere migrar.

El *Motor de Ejecución* es un componente que está muy relacionado con el O/S en el cual ejecuta. Tanto el Planificador, sus algoritmos y las Tareas de Tiempo Real deberán ser revisadas en detalle, específicamente los métodos donde se implementan:

- La creación y destrucción de threads.
- Asignación y administración de prioridades.
- Sincronización, comunicación y control entre threads.
- Administración y control del tiempo.

Por otro lado, tanto la estructura de las *Tablas de Representación* como su administración, son transparentes al O/S subyacente. Esto no incluye, los mecanismos de comunicación y sincronización utilizados en la herramienta para compartir la información, dado que son específicos de cada O/S.

En caso de implementarse un sistema SCADA, en el cual intervienen los niveles de servicio ISL y PSL, la portabilidad es aún más complicada, ya que estos servicios están muy acoplados a los servicios de Interfaz provistos por el O/S. En este caso, los niveles de servicio ISL y PSL, deberán ser programados nuevamente.

7 - UNA APLICACIÓN SCADA, USANDO ClashRT

En este capítulo se describirá e ilustrará el desarrollo de una aplicación SCADA, la cual utiliza los servicios provistos por la herramienta construida e implementa los niveles de servicio ISL y PSL.

7.1 Especificación de la aplicación SCADA.

El objetivo de implementar una aplicación SCADA, es demostrar el funcionamiento de la herramienta y de los servicios que brinda.

Con ese propósito, se construyen los niveles de servicio ISL y PSL de acuerdo a las necesidades típicas de una aplicación SCADA.

Este ejemplo de aplicación SCADA es un sistema flexible y dinámico que se configura de acuerdo a los requerimientos del operador. El sistema provee las siguientes facilidades para configurar el *Motor de Ejecución:*

- **Operación del Motor de Ejecución:** se lo puede iniciar y detener la cantidad de veces que se considere necesario, de acuerdo a los requerimientos del usuario.

- **Parámetros de Tiempo Real:** se puede configurar la *priority class* y el algorimto de emulación que se desee, con sus parámetros respectivos.

- **Parámetros de las Tareas:** se puede configurar la *meta* y tiempo de ejecución de cada tarea.

- **Log File:** se puede visualizar con un editor de texto predefinido, el archivo generado automáticamente con todo el registro de las operaciones.

Por otro lado, la aplicación SCADA permite realizar las siguientes operaciones en las *Tablas de Representación*, incluso cuando el *Motor de Ejecución* se encuentra activo:

- **Almacenar y cargar las Bases de Datos:** se pueden almacenar las tablas en sus respectivos archivos en disco o cargarlas con los últimos datos salvados, de acuerdo a las necesidades del usuario.

- **Actualización de las Imágenes:** se pueden configurar los *puntos de supervisión* para cada tipo de tableaux, con sus respectivos parámetros.

- **Actualización de las Alarmas:** se pueden configurar los registros de las alarmas, con sus respectivos parámetros.

- **Actualización de las relación Alarma-Imagen:** se pueden relacionar los registros de las alarmas con cada *punto de supervisión*, de acuerdo a los requerimientos del operador.

- **Actualización de los Mímicos:** se pueden relacionar los *puntos de supervisión* con cada proceso y con sus respectivos atributos de mímicos.

- **Actualización de los Procesos:** se pueden configurar los registros de los procesos, con sus respectivos parámetros.

- **Procesamiento de Comandos:** se pueden seleccionar los comandos predefinidos y ejecutarlos.

- **Visualización del Archivo Histórico:** se puede visualizar el contenido completo del archivo.

7.2 Implementación de la aplicación SCADA.

La aplicación SCADA está desarrollada en el lenguaje de programación **Visual C++ 6.0** de Microsoft®. Este producto está incluido en el paquete de desarrollo Microsoft® Visual Studio Development 6.0, el cual provee un conjunto de herramientas para la construcción de aplicaciones en toda la familia de Sistemas Operativos de la plataforma Win32. Esta implementación del lenguaje C++ es compatible con las normas del ANSI C++ [Str91].

El GUI (Graphical User Interface) está implementado utilizando el entorno de desarrollo **MFC (Microsoft® Foundation Class Library)**. Esta herramienta es un framework orientado a objetos, el cual se encuentra incluido dentro del producto Visual C++ y el código fuente está disponible para el programador.

Este ambiente orientado a objetos está desarrollado en C++ y su objetivo es facilitar la construcción de aplicaciones que ejecutan en un entorno gráfico. Para ello, provee una jerarquía de clases de C++, las cuales encapsulan los servicios de interfaz provistos por la Win32 API.

Si bien esta capa de abstracción facilita el desarrollo de aplicaciones en un entorno gráfico, en comparación con realizar las llamadas directamente a la Win32 API, su programación sigue siendo de muy bajo nivel y se requiere bastante conocimiento previo de los servicios de interfaz provistos por el O/S. Sin embargo, la aplicación construida con este framework, posee un muy bajo overhead adicional.

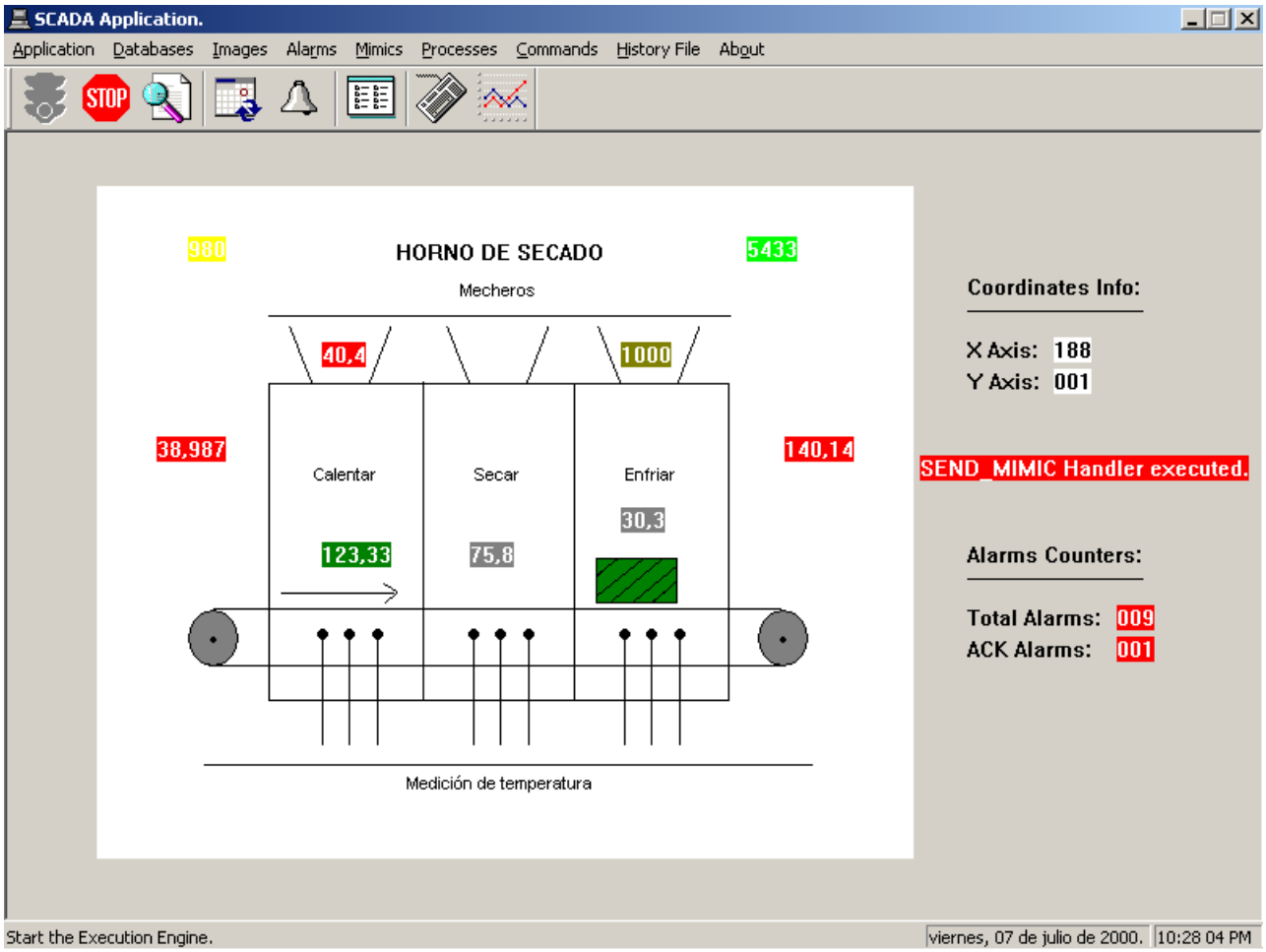
La aplicación SCADA implementa los niveles de servicio ISL y PSL. Cada uno de ellos, está construido como un conjunto de diferentes módulos. Estos módulos son archivos fuente escritos en C++, los cuales encapsulan una funcionalidad específica.

En las próximas secciones, se describirán los módulos que componen cada nivel de servicio y se ilustrarán con los objetos gráficos correspondientes.

7.2.1 Servicios de Interfaz.

A continuación se describen los módulos que componen el nivel de servicio ISL y se ilustran con algunos de sus objetos gráficos:

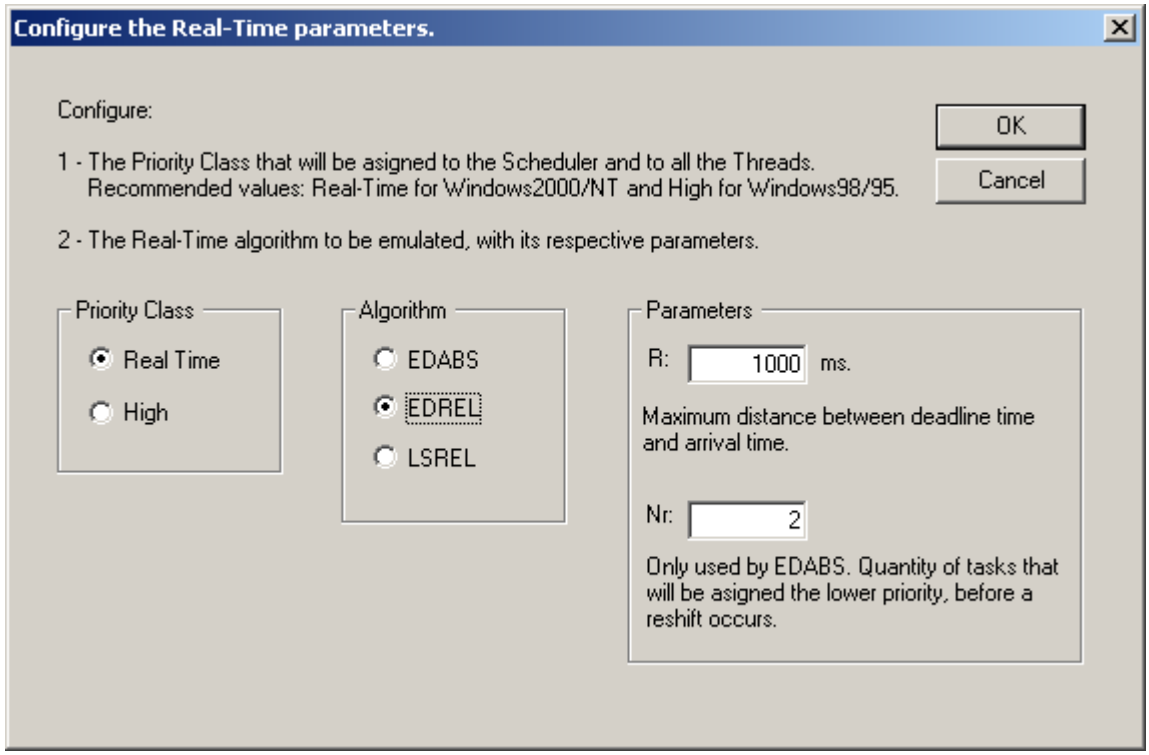
- **Módulos para la visualización de los datos:** estos componentes proveen la ventana principal y sus funciones para la actualización de ventanas y despliegue de datos y mímicos en pantalla:



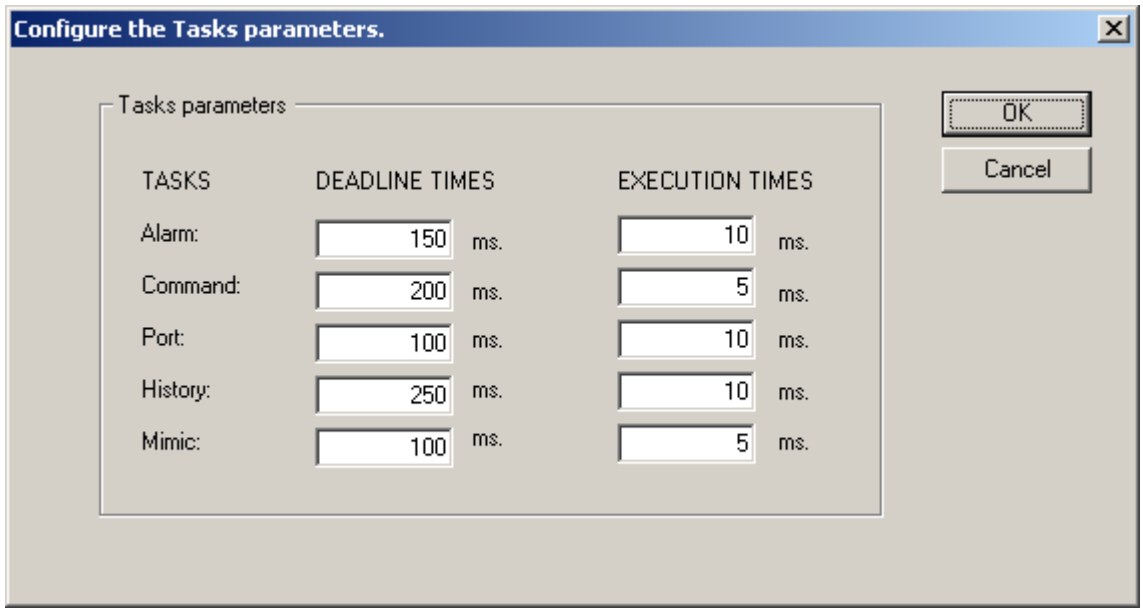
Esta ventana principal posee la barra de menús, en la cual se definen todos los items para desplegar las ventanas de operación del sistema, los front-end para la manipulación de datos y los componentes gráficos implementados por el usuario.

- **Módulos para la administración del sistema SCADA:** estas funciones incluyen las ventanas de administración y configuración del *Motor de Ejecución,* como se describe a continuación*:*

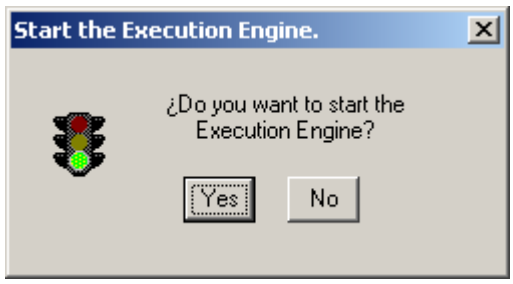
  - **Parámetros de Tiempo Real:** en esta ventana se selecciona la *priority class* del proceso, el algoritmo de emulación y sus respectivos parámetros:

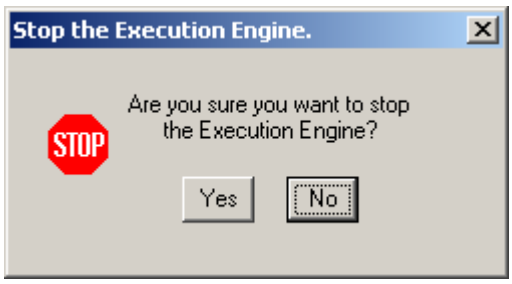


  - **Parámetros de las Tareas:** en la siguiente ventana se configura los tiempos de *meta* y ejecución para cada tarea, de acuerdo a los requerimientos del usuario:

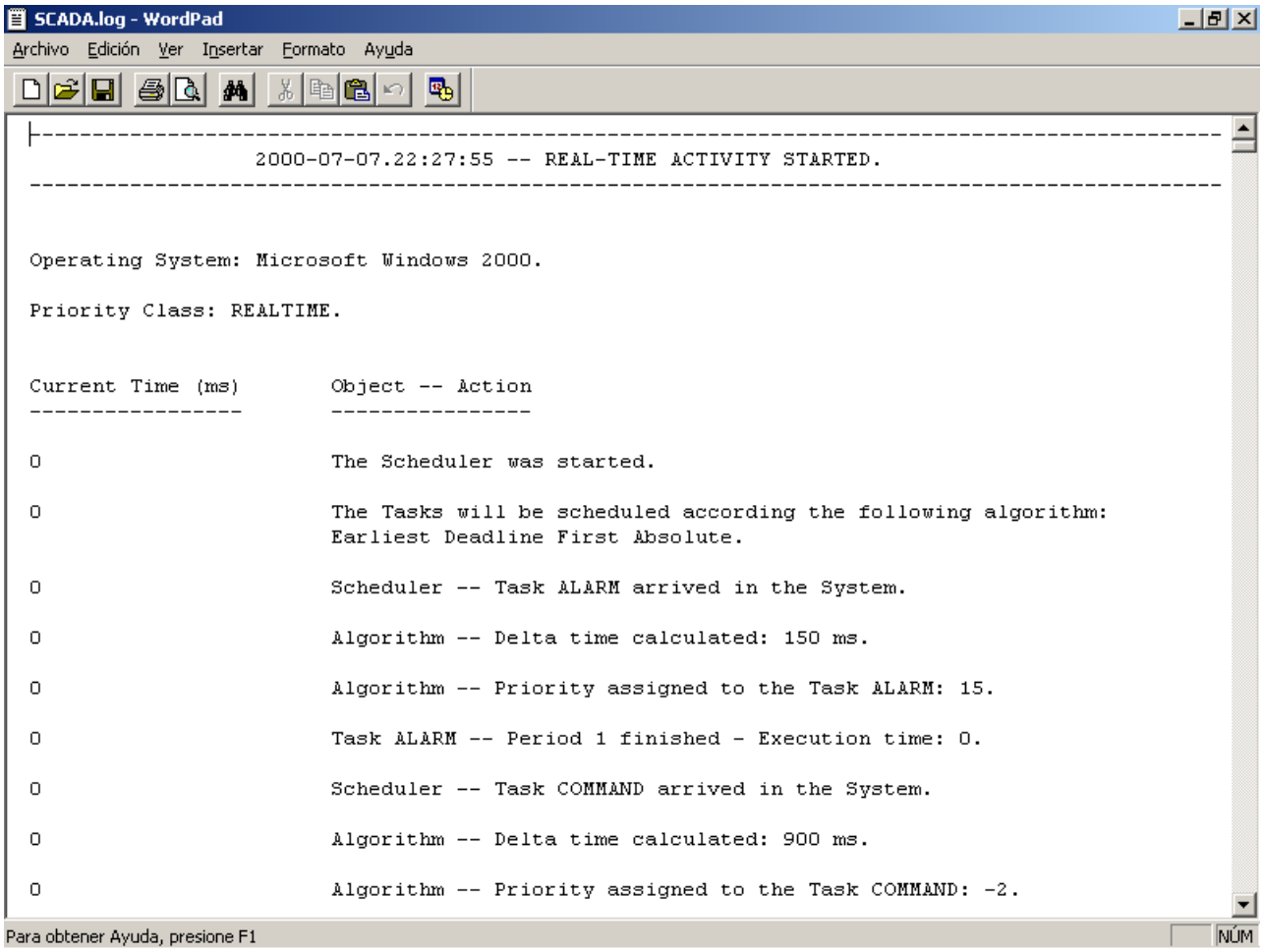


- **Ejecutar y Detener el Motor de Ejecución:** en estas ventanas se ejecuta y detiene al *Motor de Ejecución*. Los parámetros seleccionados en las ventanas anteriores son los que se aplicarán durante su ejecución.



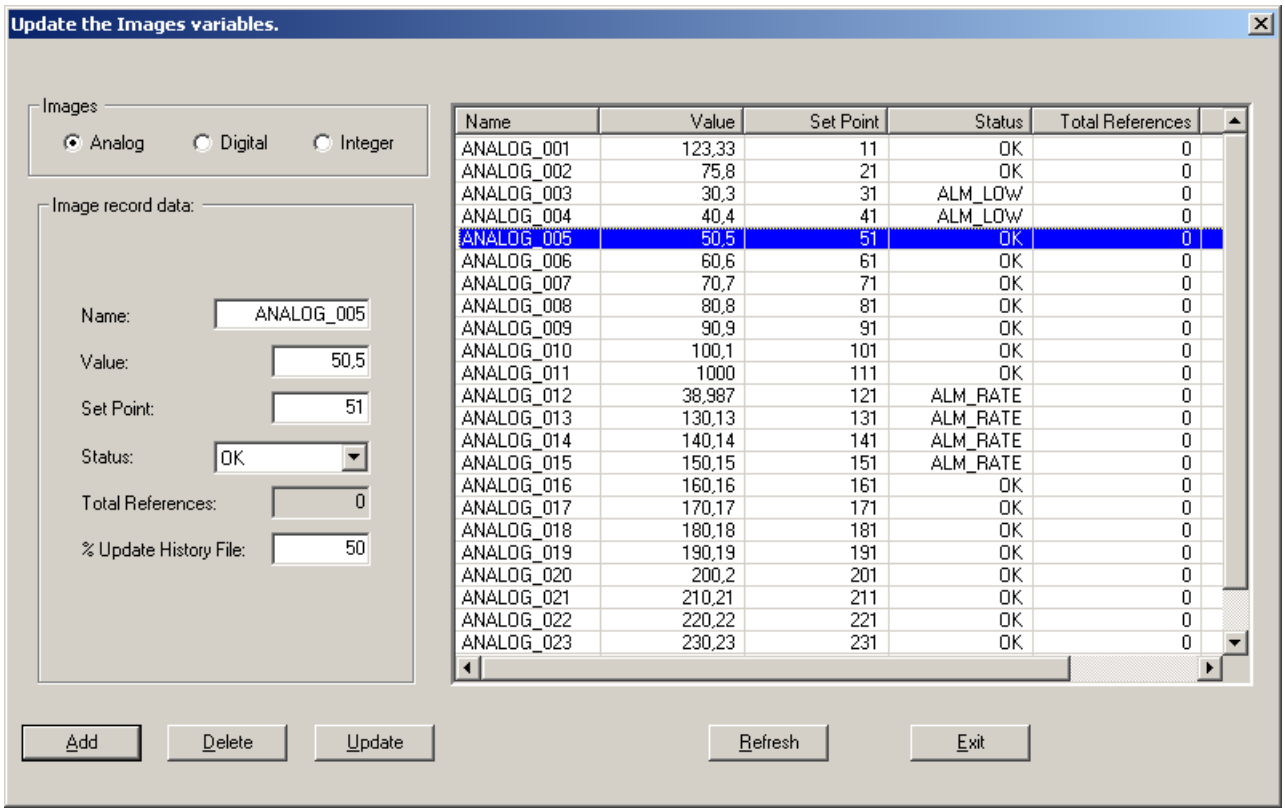


- **Visualización del Log File:** el operador puede abrir el Log File del sistema para analizar el comportamiento general del Planificador y las Tareas de Tiempo Real. Este archivo es desplegado en un editor de texto predefinido y contiene paso a paso todos los eventos ocurridos desde que se inició el *Motor de Ejecución*. Posee dos columnas: la izquierda, contiene el momento en el cual transcurrió el evento y en la derecha, se describe el tipo de evento. El tiempo está especificado en milisegundos y es relativo al momento en que se inició el *Motor de Ejecución*:



- **Módulos para la manipulación de los datos:** estas funciones incluyen los front-end para la actualización de las *Tablas de Representación* y sus relaciones entre sí, como se describe a continuación:

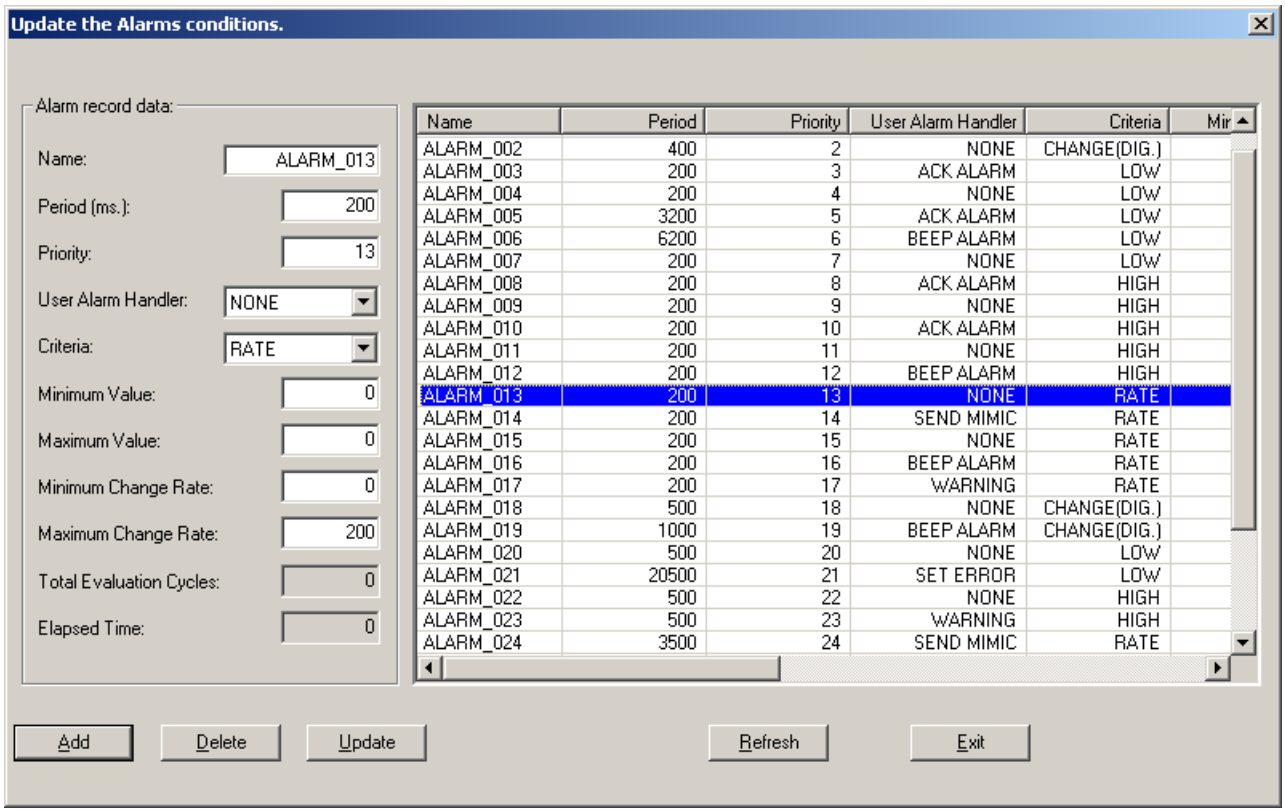
  - **Front-end para la actualización de las Imágenes:** en la siguiente ventana se realiza la configuración de los *puntos de supervisión* para cada tipo de tableaux, con sus respectivos campos: Nombre, Valor, Valor del Set Point, Total de Referencias y el Porcentaje de Actualización. Este último campo indica que un punto será almacenado en el Archivo Histórico si varía en una proporción mayor a ese porcentaje:

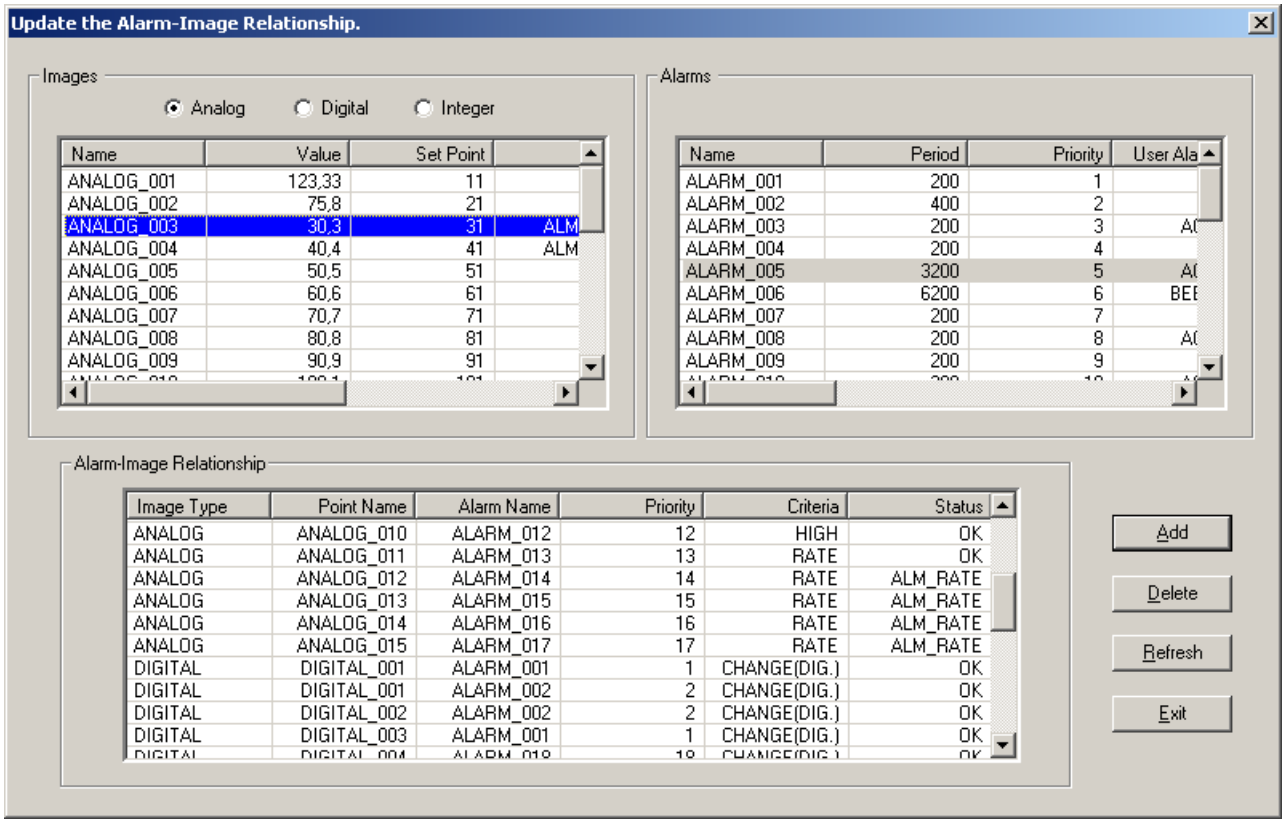


Las operaciones que se realizan en la mayoría de los front-end son: agregar, borrar y modificar los registros. A su vez, se puede refrescar la información de toda la tabla en cualquier momento. Estas acciones pueden realizarse incluso cuando el *Motor de Ejecución* esté activo.

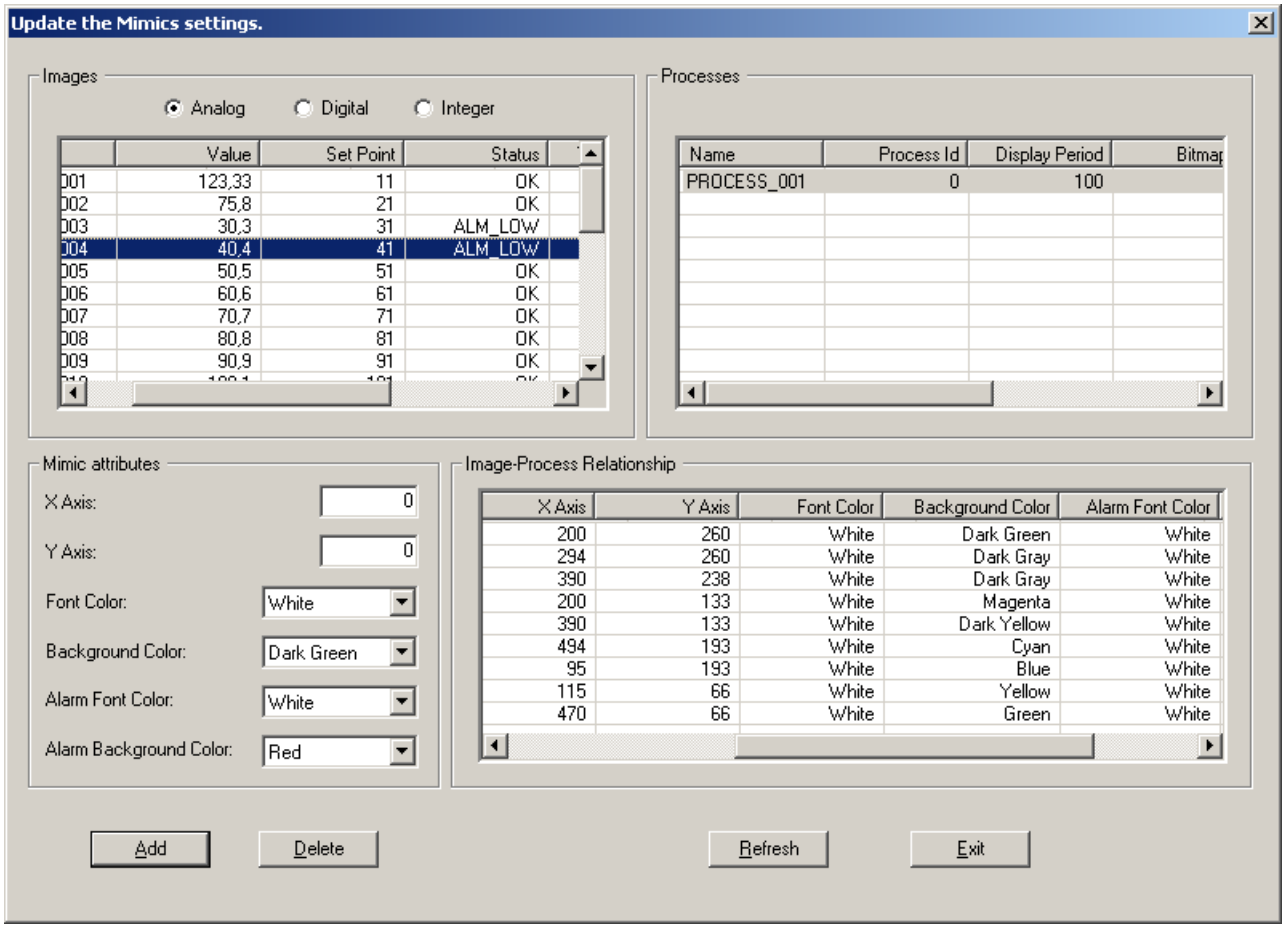
- **Front-end para la actualización de las Alarmas:** en la siguiente ventana se realiza la configuración de las condiciones de alarma, con sus respectivos campos: Nombre, Período de evaluación, Prioridad, Procedimiento de Usuario que se ejecutará en caso de se cumpla la condición, criterio de evaluación, los valores mínimos y máximos y las tasas de cambio mínimas y máximas.

- **Front-end para la actualización de la relación Alarma-Imagen:** en la siguiente ventana se realiza la asociación entre los *puntos de supervisión* y los registros de alarmas. La única restricción que existe es que los puntos digitales deberán estar asociados a condiciones de alarma que tengan criterio de evaluación CHANGE y viceversa, ya que las otras combinaciones no son aplicables:

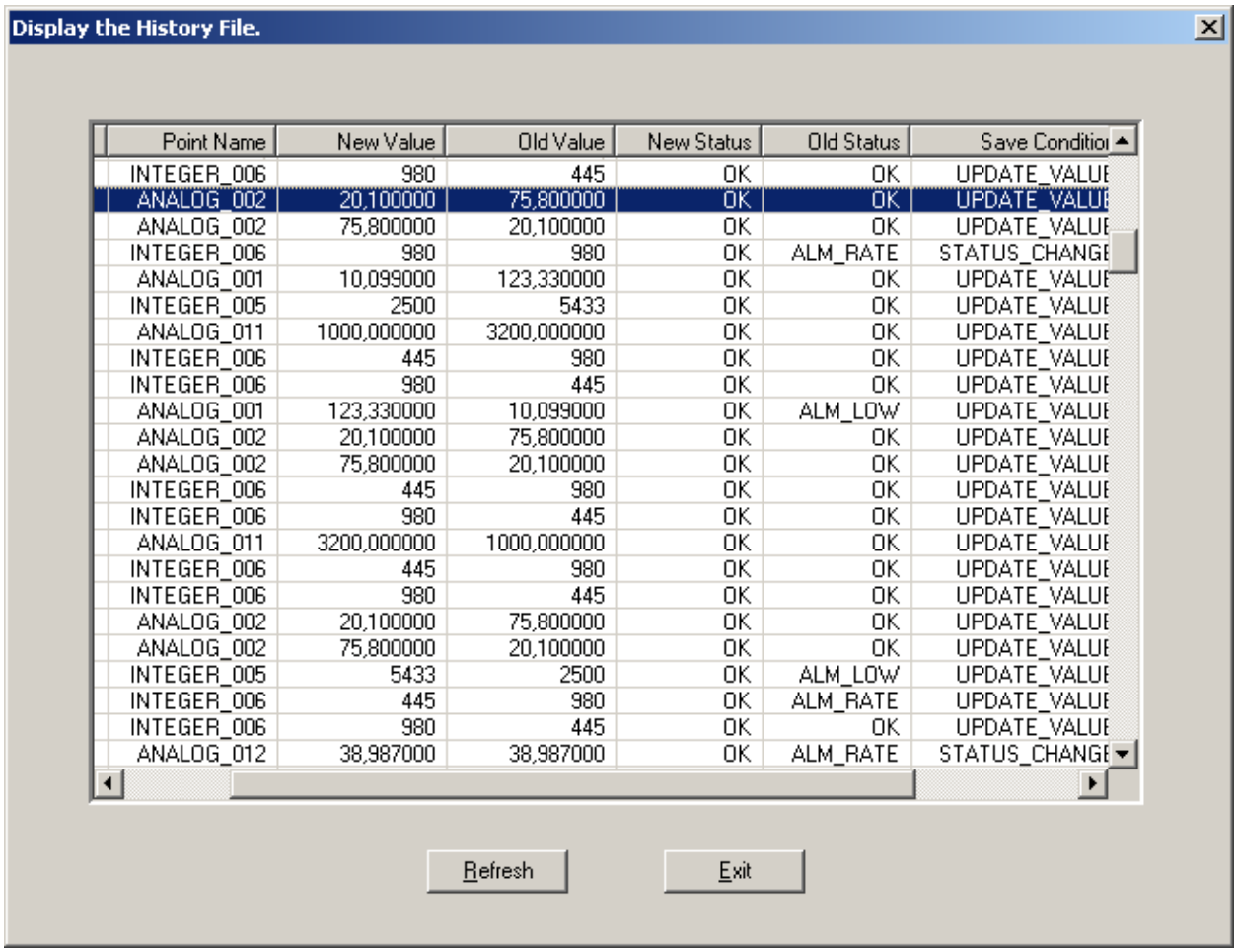

Update the Alarms conditions.
Alarm record data:
Name: ALARM_013
Period (ms.): 200
Priority: 13
User Alarm Handler: NONE
Criteria: RATE
Minimum Value: 0
Maximum Value: 0
Minimum Change Rate: 0
Maximum Change Rate: 200
Total Evaluation Cycles: 0
Elapsed Time: 0
Name | Period | Priority | User Alarm Handler | Criteria | Mir
ALARM_002 | 400 | 2 | NONE | CHANGE(DIG.)
ALARM_003 | 200 | 3 | ACK ALARM | LOW
ALARM_004 | 200 | 4 | NONE | LOW
ALARM_005 | 3200 | 5 | ACK ALARM | LOW
ALARM_006 | 6200 | 6 | BEEP ALARM | LOW
ALARM_007 | 200 | 7 | NONE | LOW
ALARM_008 | 200 | 8 | ACK ALARM | HIGH
ALARM_009 | 200 | 9 | NONE | HIGH
ALARM_010 | 200 | 10 | ACK ALARM | HIGH
ALARM_011 | 200 | 11 | NONE | HIGH
ALARM_012 | 200 | 12 | BEEP ALARM | HIGH
ALARM_013 | 200 | 13 | NONE | RATE
ALARM_014 | 200 | 14 | SEND MIMIC | RATE
ALARM_015 | 200 | 15 | NONE | RATE
ALARM_016 | 200 | 16 | BEEP ALARM | RATE
ALARM_017 | 200 | 17 | WARNING | RATE
ALARM_018 | 500 | 18 | NONE | CHANGE(DIG.)
ALARM_019 | 1000 | 19 | BEEP ALARM | CHANGE(DIG.)
ALARM_020 | 500 | 20 | NONE | LOW
ALARM_021 | 20500 | 21 | SET ERROR | LOW
ALARM_022 | 500 | 22 | NONE | HIGH
ALARM_023 | 500 | 23 | WARNING | HIGH
ALARM_024 | 3500 | 24 | SEND MIMIC | RATE
Add
Delete
Update
Refresh
Exit


Update the Alarm-Image Relationship.
Images
Analog
Digital
Integer
Name | Value | Set Point
ANALOG_001 | 123,33 | 11
ANALOG_002 | 75,8 | 21
ANALOG_003 | 30,3 | 31 | ALM
ANALOG_004 | 40,4 | 41 | ALM
ANALOG_005 | 50,5 | 51
ANALOG_006 | 60,6 | 61
ANALOG_007 | 70,7 | 71
ANALOG_008 | 80,8 | 81
ANALOG_009 | 90,9 | 91
Alarms
Name | Period | Priority | User Ala
ALARM_001 | 200 | 1
ALARM_002 | 400 | 2
ALARM_003 | 200 | 3
ALARM_004 | 200 | 4
ALARM_005 | 3200 | 5
ALARM_006 | 6200 | 6
ALARM_007 | 200 | 7
ALARM_008 | 200 | 8
ALARM_009 | 200 | 9
Alarm-Image Relationship
Image Type | Point Name | Alarm Name | Priority | Criteria | Status
ANALOG | ANALOG_010 | ALARM_012 | 12 | HIGH | OK
ANALOG | ANALOG_011 | ALARM_013 | 13 | RATE | OK
ANALOG | ANALOG_012 | ALARM_014 | 14 | RATE | ALM_RATE
ANALOG | ANALOG_013 | ALARM_015 | 15 | RATE | ALM_RATE
ANALOG | ANALOG_014 | ALARM_016 | 16 | RATE | ALM_RATE
ANALOG | ANALOG_015 | ALARM_017 | 17 | RATE | ALM_RATE
DIGITAL | DIGITAL_001 | ALARM_001 | 1 | CHANGE(DIG.) | OK
DIGITAL | DIGITAL_001 | ALARM_002 | 2 | CHANGE(DIG.) | OK
DIGITAL | DIGITAL_002 | ALARM_002 | 2 | CHANGE(DIG.) | OK
DIGITAL | DIGITAL_003 | ALARM_001 | 1 | CHANGE(DIG.) | OK
Add
Delete
Refresh
Exit

- **Front-end para la actualización de los Mímicos:** en esta ventana se realiza la asociación entre los *puntos de supervisión* y los registros de procesos, con los siguientes atributos: coordenadas x e y en donde se desplegará el mímico, su color de letra y de fondo, y su color de letra y de fondo cuando el punto está en alarma:



- **Front-end para la visualización del Archivo Histórico:** en la siguiente ventana se visualiza el contenido completo del archivo al momento de seleccionar esta opción. Este archivo se encuentra siempre abierto y se va actualizando periódicamente por la Tarea Histórico. Cuando se solicita este front-end, el sistema muestra una foto del archivo en ese momento:



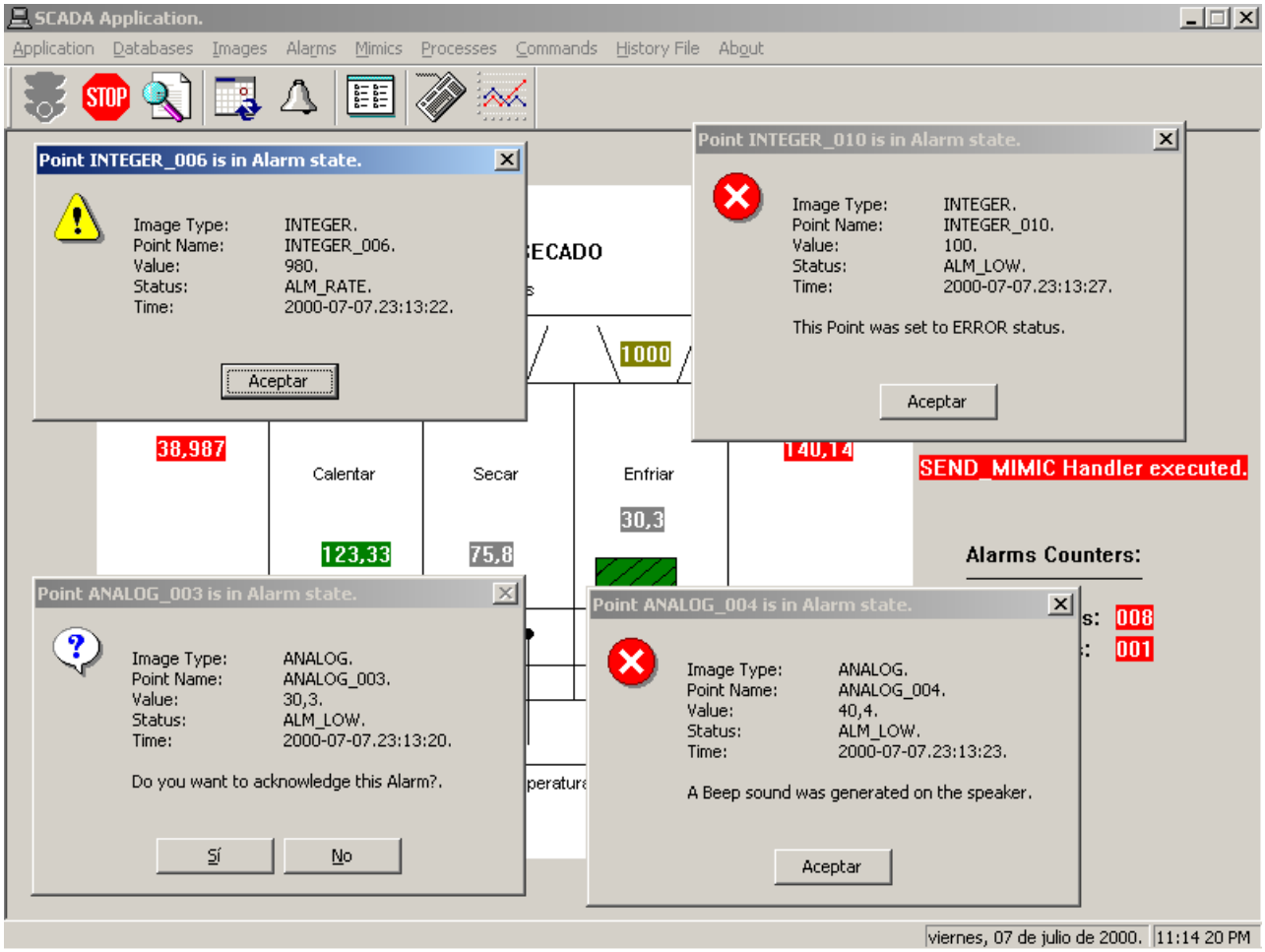
Todas las *Tablas de Representación* y sus relaciones son consistentes. En general, el campo Nombre de cada tabla es la *primary key* y cada una de ellas se encuentra normalizada. Cada vez que se realiza una actualización en cualquiera de las tablas, se verifica la consistencia completa de la Base de Datos.

7.2.2 Servicios Particulares.

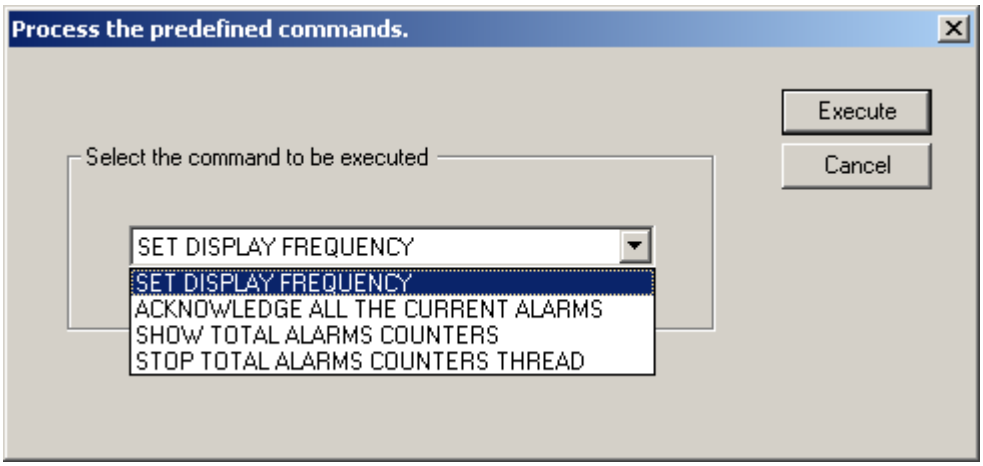
A continuación se describen los módulos que componen el nivel de servicio PSL y se ilustran con algunos de sus objetos gráficos:

- **Módulos de atención de alarmas:** estas funciones incluyen rutinas implementadas específicamente para esta aplicación SCADA, las cuales son activadas cuando un *punto de supervisión* cumple con la condición de alarma relacionada. Los procedimientos desarrollados son: solicitar al operador el reconocimiento de la alarma, desplegar una ventana de advertencia, generar

un sonido de alarma, asignar ERROR al punto que se encuentra en alarma y enviar un mensaje de alarma. A continuación se muestra la ventana principal con diferentes puntos en alarma y la activación de los procedimientos de usuario respectivos:



- **Módulos de atención de comandos:** estas funciones incluyen rutinas implementadas específicamente para esta aplicación SCADA, las cuales son ejecutadas bajo demanda del operador. Los procedimientos desarrollados son: mostrar totales de alarmas, incrementar la frecuencia de display de los mímicos, reconocer todas las alarmas actuales y parar el thread que muestra los totales de alarmas. A continuación se muestra la ventana en la cual se seleccionan los comandos a ejecutar:



Todas las funciones de atención de alarmas y comandos están implementadas como threads de Win32 independientes de los threads que ejecutan las Tareas de Tiempo Real.

Por otro lado, el *primary thread* de la aplicación, contiene la ventana principal y todas las ventanas para la actualización de tablas.

Tanto el *primary thread* del ISL, como los threads del PSL, se les asigna en tiempo de ejecución la mínima prioridad dentro de la *priority class* elegida para toda la aplicación.

8 - RESULTADOS OBTENIDOS

En esta sección se describirán los resultados obtenidos luego de realizar diferentes pruebas de integración con la aplicación SCADA y de desempeño con la herramienta ClashRT.

Con estas pruebas, también se estudia la utilidad de la herramienta construida en esta Tesis, como así también su flexibilidad para incluir diferentes configuraciones y requerimientos.

8.1 Pruebas de integración con la aplicación SCADA.

La aplicación SCADA está desarrollada para ejecutar en cualquier O/S de la plataforma Win32. En particular, la etapa de programación y testing se realizó inicialmente en WindowsNT y luego en Windows2000, dado que son los sistemas más robustos y confiables de toda la plataforma de Win32.

El hardware utilizado para el desarrollo y testing de la aplicación fue esencialmente dos tipos de sistemas:

- Una Laptop, con procesador Intel® Pentium 75 MHz, 40 MB de memoria RAM y con WindowsNT, versión 4.0 y SP5 instalado.

- Una Desktop, con procesador Intel® Celeron 500 MHz, 128 MB de memoria RAM y con Windows2000 instalado.

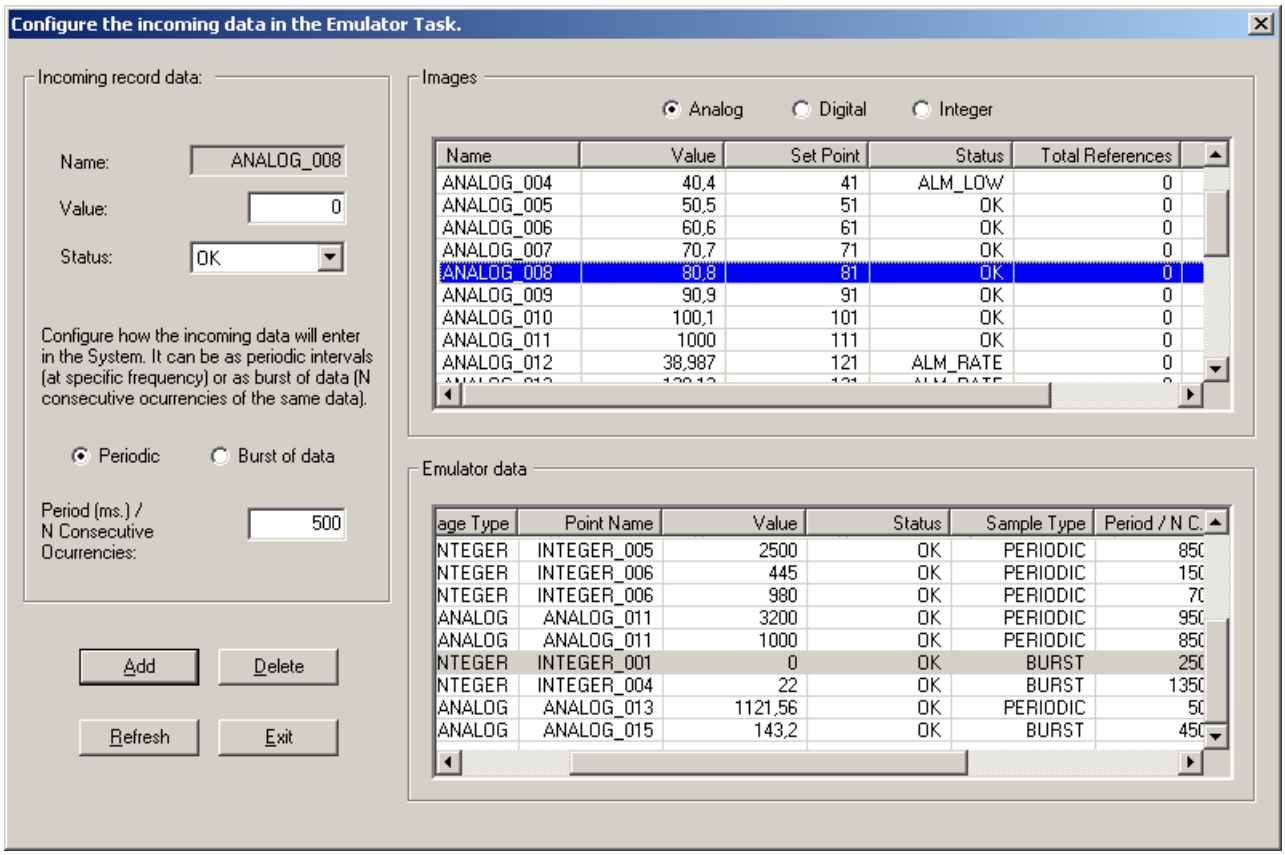
Si bien esta última configuración es mucho más poderosa que la primera, esto no es un requerimiento para ejecutar la aplicación, sino que se utilizó para realizar la etapa de compilación y testing de forma más rápida.

La aplicación fue sometida a pruebas exhaustivas con el mundo exterior, a través de dos técnicas de simulación:

- Por medio de un simulador que se ejecuta en otra máquina y está comunicado vía RS-232 con la computadora en donde se ejecuta la aplicación. Este simulador es utilizado para realizar la transmisión y recepción de datos desde el exterior hacia la aplicación.

- Por medio de un emulador de ports, el cual ejecuta en la misma máquina en donde está la aplicación. Este emulador es un thread independiente, el cual escribe en la Cola de Entrada para inyectar datos a la aplicación.

En la siguiente ventana se muestra el front-end del emulador de ports:



El emulador de ports es sincronizado por el Planificador de ClashRT. De esta manera, se ejecuta en forma cíclica y escribe los datos en la Cola de Entrada de acuerdo a una frecuencia definida.

Todo este comportamiento está implementado en la Clase Emulador, en donde se encapsula la Tarea de Tiempo Real y la *Tabla de Representación*.

En la Tabla de Emulador se definen todos los *puntos de supervisión* que se desea que la tarea escriba en la Cola de Entrada. Cada punto está definido en un registro, el cual contiene los siguientes campos: Tipo de Tableaux, Nombre, Valor, Estado, Tipo de Muestreo y el Valor del Tipo de Muestreo.

A su vez, cada registro está definido en alguno de los 2 modos de operación que soporta el emulador:

- **Modo Periódico:** en este modo, el emulador escribe periódicamente en la Cola de Entrada, el Valor del punto con su Estado respectivo. El período es el Valor del Tipo de Muestreo.

- **Modo Ráfaga:** en este modo, el emulador escribe en la Cola de Entrada, el Valor del punto con su Estado respectivo, N veces en forma consecutiva, una vez por cada ciclo de ejecución de la tarea. N es el Valor del Tipo de Muestreo.

Se definieron varias instancias de la Clase Emulador, con el objetivo de emular diferentes conexiones con el mundo exterior, cada una con su secuencia de eventos y conjunto de variables que se desea monitorear.

Esta implementación del emulador de ports, permitió demostrar la facilidad de ClashRT para planificar otras Tareas de Tiempo Real, adicionales al conjunto predefinido, y para realizar diferentes pruebas de funcionamiento del sistema SCADA.

La aplicación SCADA desarrollada en esta Tesis, provee la flexibilidad necesaria para configurar dinámicamente el sistema, de acuerdo a los requerimientos del usuario.

Los módulos para la administración del sistema SCADA, permiten modificar los parámetros del *Motor de Ejecución*, iniciar su ejecución, analizar el comportamiento del sistema, detener su ejecución y luego volver a modificar estos parámetros para realizar diferentes pruebas.

A su vez, por medio de los módulos de manipulación de datos, se pueden agregar, modificar o borrar los registros de las *Tablas de Representación*, cada vez que sea necesario. Todas estas operaciones se realizan en memoria principal e incluso cuando el sistema está on-line. Para almacenar todos los cambios realizados, el sistema provee la posibilidad de grabar las Bases de Datos a disco y, cuando el operador lo requiera, de cargarlas nuevamente para inicializar todas las tablas.

Este ejemplo de aplicación SCADA complejo, permitió medir la utilidad de la herramienta construida en esta Tesis. La flexibilidad dada por la configuración dinámica, permite aplicar esta aplicación a cualquier uso particular.

8.2 Pruebas de desempeño.

El objetivo de las pruebas de desempeño es estudiar y analizar el comportamiento de los siguientes componentes:

- Los algoritmos de emulación de ClashRT.

- La aplicación SCADA ante sobrecargas del sistema.

Para poder medir el desempeño de ClashRT y de la aplicación SCADA, se analizan los tiempos de carga de CPU versus el parámetro *missed deadlines*, el cual mide el porcentaje de *metas* que no se cumplieron para cada algoritmo de emulación.

Si bien, el sistema estará ocasionalmente sobrecargado, es precisamente en esos momentos donde se requiere de una política adecuada de planificación para poder asegurar que se perderán la mínima cantidad de *metas* posibles. Es por esta razón que se compara la carga de CPU versus el parámetro *missed deadlines*.

Con el propósito de medir el cumplimiento de las *metas* de las tareas, el *Motor de Ejecución* de ClashRT realiza un control de flujo muy preciso de la ejecución del Planificador y de las Tareas de Tiempo Real:

- Por medio de la función *CurrentTime()*, mantiene un contador general de tiempo, relativo al inicio del *Motor de Ejecución*.

- Cada vez que el Planificador envía una señal a una tarea, incrementa el contador *ciclos_señalizados* de esa tarea.

- Cada vez que una tarea finaliza un ciclo de ejecución, incrementa el contador de *ciclos_ejecutados* de esa tarea.

- Cuando el *Motor de Ejecución* finaliza su ejecución, se puede medir cuántos ciclos debería haber ejecutado cada tarea, dado que se conoce el tiempo transcurrido desde su inicio, los parámetros de las tareas y los contadores descriptos precedentemente.

Todas estas operaciones son registradas en el Log File del sistema. Cuando se detiene el *Motor de Ejecución*, se escriben al final del archivo las estadísticas generales por cada tarea y el promedio total de *missed deadlines*.

Es importante señalar, que la aplicación SCADA en situaciones normales de ejecución, cumple todas las *metas* de sus tareas. Incluso, cuando existen sobrecargas esporádicas, el porcentaje de *missed deadlines* es prácticamente 0, independientemente del hardware donde se ejecute.

Como fue estudiado en el capítulo 7, la arquitectura de la aplicación SCADA está compuesta por los 3 niveles de servicios definidos para este tipo de sistemas: BSL, ISL y PSL. El nivel de servicio BSL, es la herramienta desarrollada en esta Tesis, ClashRT, y los niveles ISL y PSL, son el conjunto de módulos descriptos en el capítulo anterior.

Desde el punto de vista del O/S, la aplicación SCADA es un proceso de Win32, el cual contiene sus respectivos threads para cada nivel de servicio:

- El *primary thread*, en donde se ejecuta el nivel de servicio ISL.

- El thread del Planificador y un thread para cada Tarea de Tiempo Real. Todos estos threads componen el *Motor de Ejecución* del BSL.

- Los threads de usuario del PSL, como pueden ser las rutinas de atención de alarmas o de comandos.

Debido a que la aplicación SCADA contiene relativamente pocas tareas, no es adecuado estudiar el comportamiento de los algoritmos de emulación basándose en una pequeña cantidad de tareas. Es por ello, que se necesita de alguna aplicación adicional, que pueda crear N tareas independientes y, en consecuencia, poder medir el desempeño de los algoritmos con una muestra de tareas considerable.

Con ese objetivo, se utiliza una aplicación llamada **TestExecutionEngine**. Esta aplicación es una versión reducida y modificada del *Motor de Ejecución* de ClashRT. Simplemente consta de 4 clases de ClashRT: Planificador, Algoritmo, Tarea y Bsl. Las dos primeras clases son las mismas que en ClashRT, ya que precisamente lo que se quiere analizar es el desempeño de los algoritmos. Las otras dos clases, Tarea y Bsl, contienen algunas pequeñas modificaciones, que es justamente lo que se quiere cambiar. En particular, la Clase Tarea, posee algunos métodos adicionales a la original, como por ejemplo *ejecutar()* y *detener()*, los cuales se utilizan para crear y destruir N threads en tiempo de ejecución. A su vez, como el *Motor de Ejecución* tiene la misma arquitectura, también genera un Log File con las estadísticas de *missed deadlines* para todas las tareas que supervisa.

Además de utilizar la aplicación *TestExecutionEngine* para analizar el desempeño de los algoritmos de emulación, también se utiliza para sobrecargar al sistema y, de esta manera, estudiar la estabilidad de la aplicación SCADA.

Se realizaron diferentes pruebas de desempeño, cada una con sus parámetros específicos, pero siempre respetando el siguiente **modelo base de testing:**

- Las pruebas se realizan en la plataforma Windows2000/NT.

- La aplicación SCADA se configura de la siguiente manera: la *priority class* para toda la aplicación es REALTIME_PRIORITY_CLASS y los requerimientos de tiempo de las tareas, se asignan con valores predeterminados. El algoritmo de emulación elegido depende de la prueba a realizar.

- Se ejecuta la aplicación SCADA y se inicia su *Motor de Ejecución*.

- A partir de este momento, el Planificador y las Tareas de Tiempo Real de la aplicación SCADA, se encuentran ejecutando sus respectivos ciclos de ejecución de tiempo infinito.

- Se ejecuta la aplicación *TestExecutionEngine* con sus respectivos parámetros. Esta aplicación se ejecuta en otro proceso del O/S, independiente de la aplicación SCADA.

- Dentro del proceso *TestExecutionEngine*, se crean N threads, con distintos tiempos de llegada y de período, y se varían los diferentes algoritmos de emulación. Este programa es una *Console Application de Win32*, la cual opera en una consola del sistema, no utiliza el entorno gráfico y se ejecuta en modo background. El objetivo de ejecutar esta aplicación es **doble**: por un lado, analizar el comportamiento de los algoritmos de emulación, y por otro, poder sobrecargar el sistema para estudiar la estabilidad de la aplicación SCADA.

- En este momento, el O/S está ejecutando dos procesos independientes: la aplicación SCADA y *TestExecutionEngine*.

- El sistema es monitoreado con herramientas como Performance Monitor y Task Manager, propias del O/S.

- Luego de transcurrido un tiempo predefinido de sobrecarga, se detiene la ejecución de *TestExecutionEngine*.

- A continuación, se detiene la ejecución de la aplicación SCADA.

- Se analizan los Log Files de cada aplicación y se obtienen los resultados correspondientes.

Luego de cada prueba, se genera el gráfico respectivo con los resultados obtenidos. Cada uno de ellos, muestra la carga promedio del sistema versus el porcentaje de *missed deadlines*.

Para medir la carga promedio del sistema, ρ, se utiliza la siguiente fórmula:

$$\rho = \sum_{i=1}^{n} (C_i/T_i)$$

donde n es la cantidad de tareas, T_i es el período de cada tarea y C_i su peor tiempo de ejecución.

Cada una de las tareas que se ejecutan en *TestExecutionEngine*, son *compound bound tasks*, esto significa, tareas que requieren mucho procesamiento y, por lo tanto, compiten constantemente por la asignación de la CPU.

Esencialmente, cada tarea de *TestExecutionEngine*, ejecuta un *loop* muy grande, en el cual se realizan operaciones matemáticas. El objetivo de estas tareas es consumir tiempo de procesador y, de esta manera, provocar una sobrecarga constante en el sistema. A continuación se describe el pseudocódigo de una tarea de *TestExecutionEngine*:

```
void tarea::ejecutar(int indice)
{
    HANDLE              evento = 0;
    char                nombre_evento[20] = "EVENT_";
    unsigned long int   antes = 0, ahora = 0, tiempo_transcurrido;

    // Se abre el evento, el cual fue creado por el Planificador.
    evento = OpenEvent(EVENT_ALL_ACCESS, FALSE,
                       strcat(nombre_evento, leer_nombre(indice));

    while (leer_continuar(indice)) {

        // Espera por la señalización del Planificador.
        WaitForSingleObject(evento, INFINITE);

        // Modifica el evento al estado non-signaled.
        ResetEvent(evento);

        // Se asigna la cantidad de milisegundos transcurridos desde el inicio del sistema,
        // para poder contar el transcurso del tiempo.
        antes = GetTickCount();

        // En este lugar se realiza todo el procesamiento correspondiente al thread.
        ……

        for (i=0,i< LIMITE_LOOP;i++) {

            cálculo con operaciones matemáticas.
        }

        ……
        // En caso que se necesite, se calcula el tiempo transcurrido de cada ciclo de
        // ejecución.
        ahora = GetTickCount();
        tiempo_transcurrido = DeltaTime(ahora, antes);
    }

    // Se cierra el objeto evento, el cual es borrado del O/S.
    CloseHandle(evento);
}
```

Para estimar cada C_i, se realizaron pruebas individuales con cada tarea y se tomó como referencia el promedio de todos los ciclos de ejecución. Estas pruebas se realizaron cuando el sistema está sin carga de CPU, para poder realizar una estimación lo más exacta posible.

Para simplificar la cantidad de combinaciones de los parámetros involucrados, los valores de C_i y T_i son fijos por cada prueba que se realiza. Esto significa que, en una prueba en particular, todas las tareas de *TestExecutionEngine* tendrán el mismo tiempo de ejecución y el mismo período.

Las tareas que se ejecutan en *TestExecutionEngine*, son ingresadas al sistema con una frecuencia de aproximadamente 5 tareas por segundo.

Cada ejecución de *TestExecutionEngine*, tiene asociada los siguientes parámetros:

- **A**, algoritmo de emulación.
- **N**, cantidad de tareas;
- **T_i**, período de las tareas;
- **C_i**, tiempo de ejecución de las tareas;
- **R**, valor máximo de la distancia entre el período de las tarea y el tiempo de llegada de la misma. Este valor es utilizado por cualquiera de los algoritmos de emulación.
- **Nr**, valor utilizado sólo para *EDABS*, para especificar la cantidad máxima permitida de tareas que se les asigna la prioridad del menor nivel en forma consecutiva. Cuando se llega a esta cota, el algoritmo realiza un reshift de prioridades.
- **LIMITE_LOOP**, valor que especifica la longitud del *loop* en donde se realizan las operaciones matemáticas, dentro del ciclo de ejecución de la tarea. Este valor es utilizado para incrementar o disminuir el tiempo de ejecución de las tareas.

Por otro lado, el parámetro de carga promedio del sistema, ρ, varía entre 0.45 y 0.90. Esto es así, ya que con valores menores al 45% de utilización del procesador, no se experimentan diferencias entre los algoritmos, y con valores superiores al 90% teórico de carga de CPU, el sistema llega a tener una utilización del procesador del 100%, debido a los *cambios de contexto* de los threads y al propio *overhead* del O/S.

En estos casos de máxima sobrecarga, el sistema queda sin respuesta, ya que está constantemente ejecutando y sirviendo a los threads de tiempo crítico.

En los siguientes gráficos, se analizan los 3 algoritmos de emulación estudiados en esta Tesis: *EDABS*, *EDREL* y *LSREL*. Para cada uno de ellos, se muestra el desempeño de las siguientes 4 curvas:

- La aplicación *TestExecutionEngine* con 50 tareas y sus respectivos parámetros.
- La aplicación *TestExecutionEngine* con 100 tareas y sus respectivos parámetros.
- La aplicación SCADA cuando se ejecuta *TestExecutionEngine* con 50 tareas y sus respectivos parámetros.
- La aplicación SCADA cuando se ejecuta *TestExecutionEngine* con 100 tareas y sus respectivos parámetros.

Cada punto del gráfico se corresponde con la secuencia de pasos del *modelo base de testing:*

- Se ejecuta la aplicación SCADA con sus parámetros respectivos.
- Se ejecuta la aplicación *TestExecutionEngine* con N tareas y sus parámetros respectivos, de forma tal de generar una carga promedio ρ.
- Luego de transcurrido un tiempo predefinido de sobrecarga, se detiene la ejecución de *TestExecutionEngine*.
- A continuación, se detiene la ejecución de la aplicación SCADA.
- Se obtienen los resultados de ambas aplicaciones y se generan los puntos del gráfico correspondientes.

Esta secuencia se repite para las 4 combinaciones de ρ igual a 45% e igual a 90%; y de N igual a 50 y N igual a 100.

Los parámetros utilizados en la aplicación *TestExecutionEngine* para cada prueba, son los siguientes.

- **TEE, N = 50**, Ti = 1100 ms., Ci = 10 ms., R = 12000, Nr = 3, LIMITE_LOOP = 1000000. Este conjunto de parámetros genera una sobrecarga ρ igual a 45%.

- **TEE, N = 50**, Ti = 550 ms., Ci = 10 ms., R = 12000, Nr = 3, LIMITE_LOOP = 1000000. Este conjunto de parámetros genera una sobrecarga ρ igual a 90%.
- **TEE, N = 100**, Ti = 2200 ms., Ci = 10 ms., R = 22000, Nr = 5, LIMITE_LOOP = 1000000. Este conjunto de parámetros genera una sobrecarga ρ igual a 45%.
- **TEE, N = 100**, Ti = 1100 ms., Ci = 10 ms., R = 22000, Nr = 5, LIMITE_LOOP = 1000000. Este conjunto de parámetros genera una sobrecarga ρ igual a 90%.

  A su vez, los parámetros utilizados en la aplicación SCADA son:

  - El algoritmo de emulación correspondiente se selecciona para cada prueba en particular y es el mismo que en la aplicación *TestExecutionEngine*.
  - **R**, es igual a 400 ms.
  - **Nr**, es igual a 2 (sólo usado en *EDABS*).
  - Los períodos de las Tareas de Tiempo Real son: **Tarea Alarma**, 100 ms.; **Tarea Comando**, 500 ms.; **Tarea Histórico**, 100 ms.; **Tarea Port**:, 250 ms. y **Tarea Mímico**: 200 ms.

A continuación se muestran los gráficos correspondientes a las pruebas con el algoritmo *EDABS*. Se generan dos figuras por cada algoritmo, debido al formato de la escalas de *missed deadlines*:

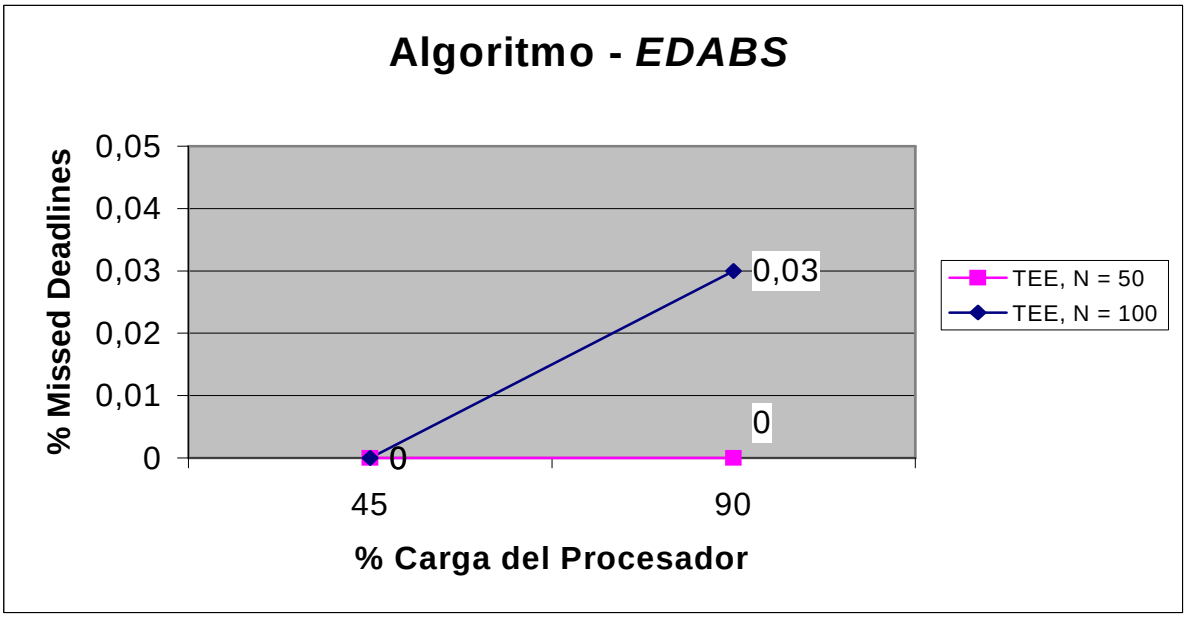


Figura 9 - Resultados con el Algoritmo EDABS para TestExecutionEngine.

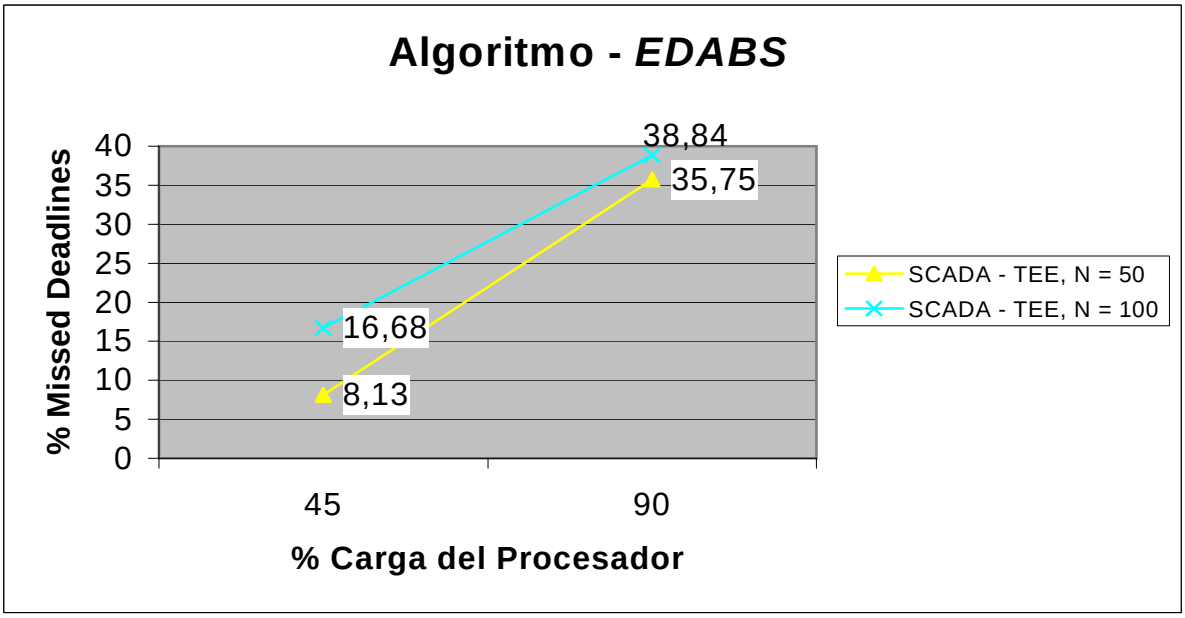


Figura 10 - Resultados con el Algoritmo EDABS para la aplicación SCADA.

A continuación se muestran los gráficos correspondientes a las pruebas con el algoritmo *EDREL*. Se generan dos figuras por cada algoritmo, debido al formato de la escalas de *missed deadlines*:

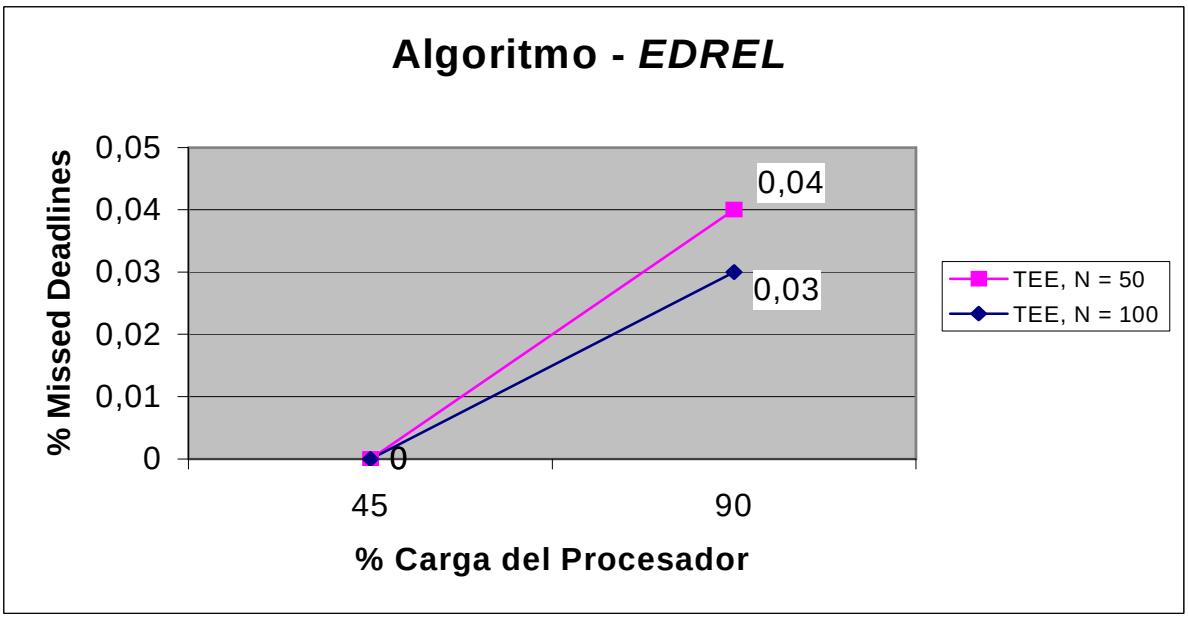


Figura 11 - Resultados con el Algoritmo EDREL para TestExecutionEngine.

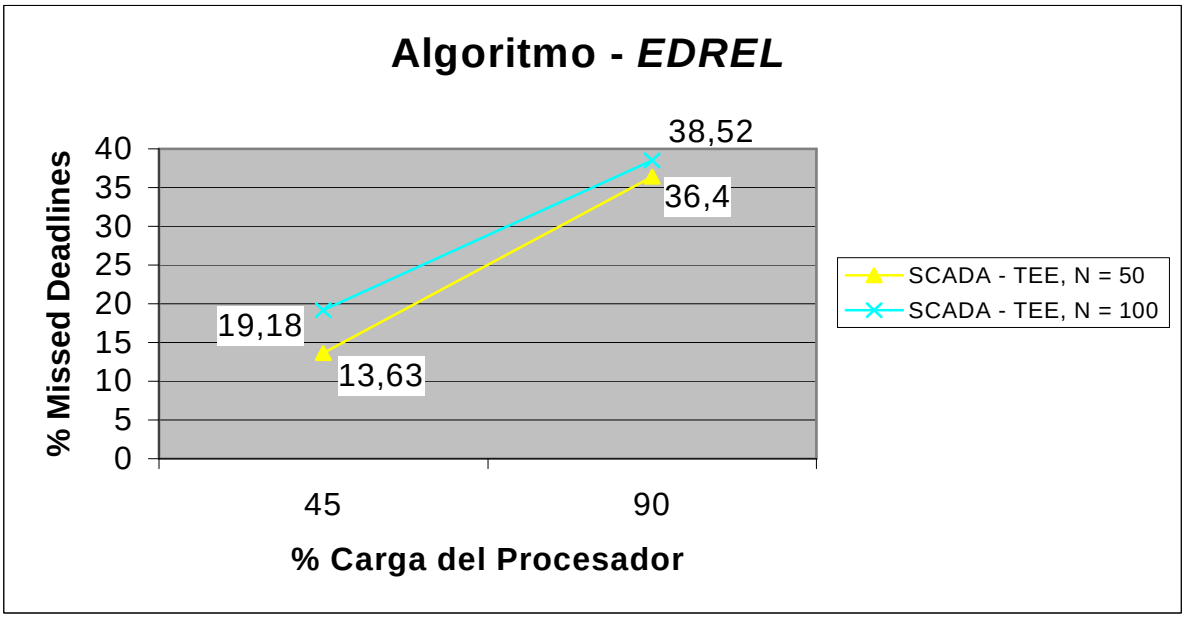


Figura 12 - Resultados con el Algoritmo EDREL para la aplicación SCADA.

A continuación se muestran los gráficos correspondientes a las pruebas con el algoritmo *LSREL*. Se generan dos figuras por cada algoritmo, debido al formato de la escalas de *missed deadlines*:

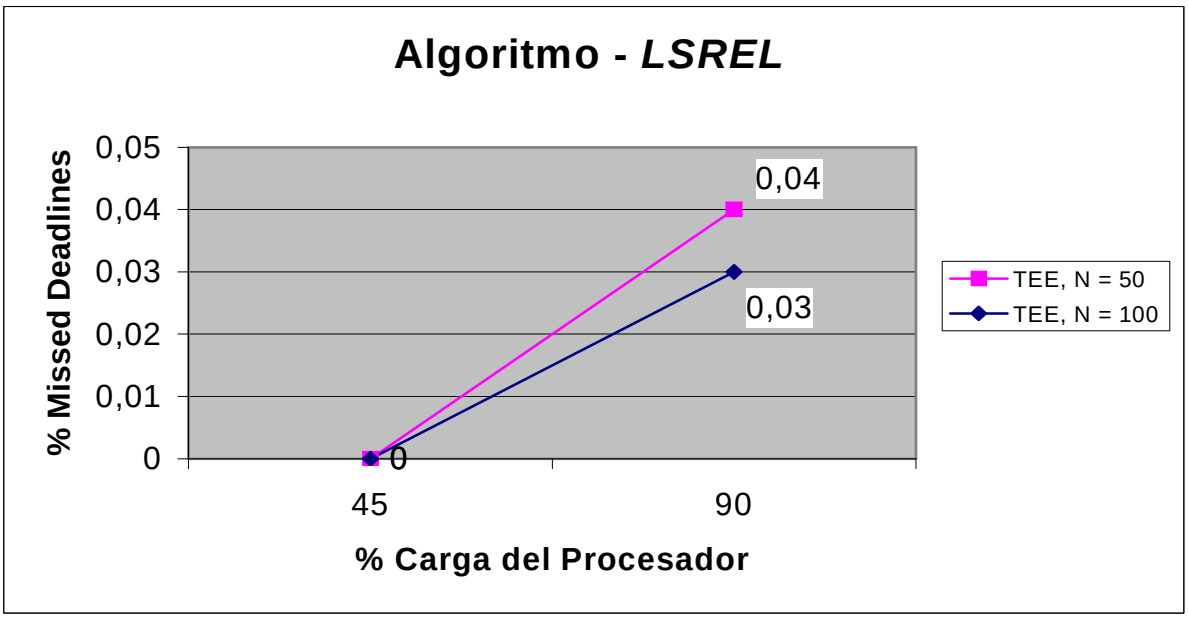


Figura 13 - Resultados con el Algoritmo LSREL para TestExecutionEngine.

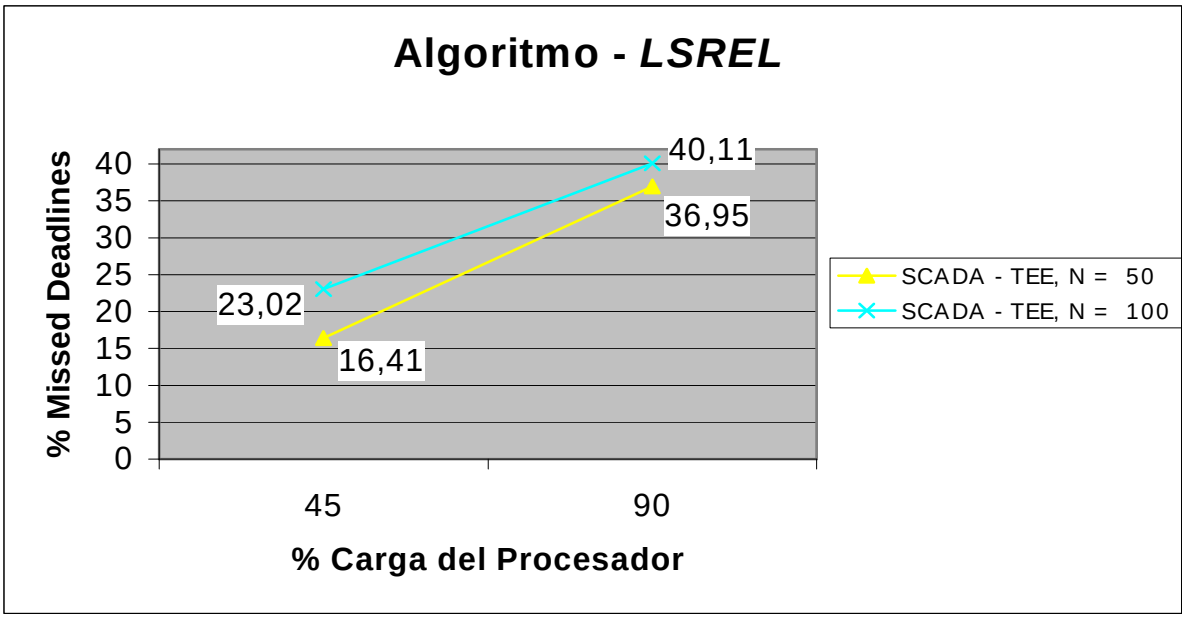


Figura 14 - Resultados con el Algoritmo LSREL para la aplicación SCADA.

A continuación se muestran los gráficos comparativos de los algoritmos para ambas aplicaciones, de acuerdo al promedio de los resultados obtenidos:

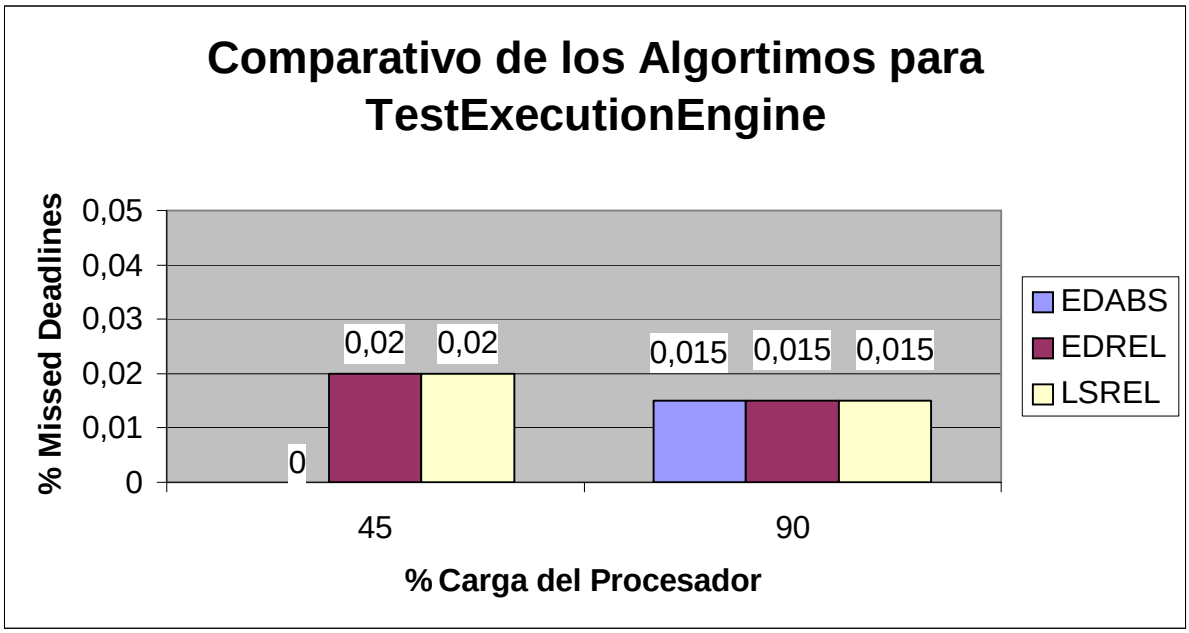


Figura 15 - Gráfico comparativo con los resultados promedio para TestExecutionEngine con N = 50 y N = 100.

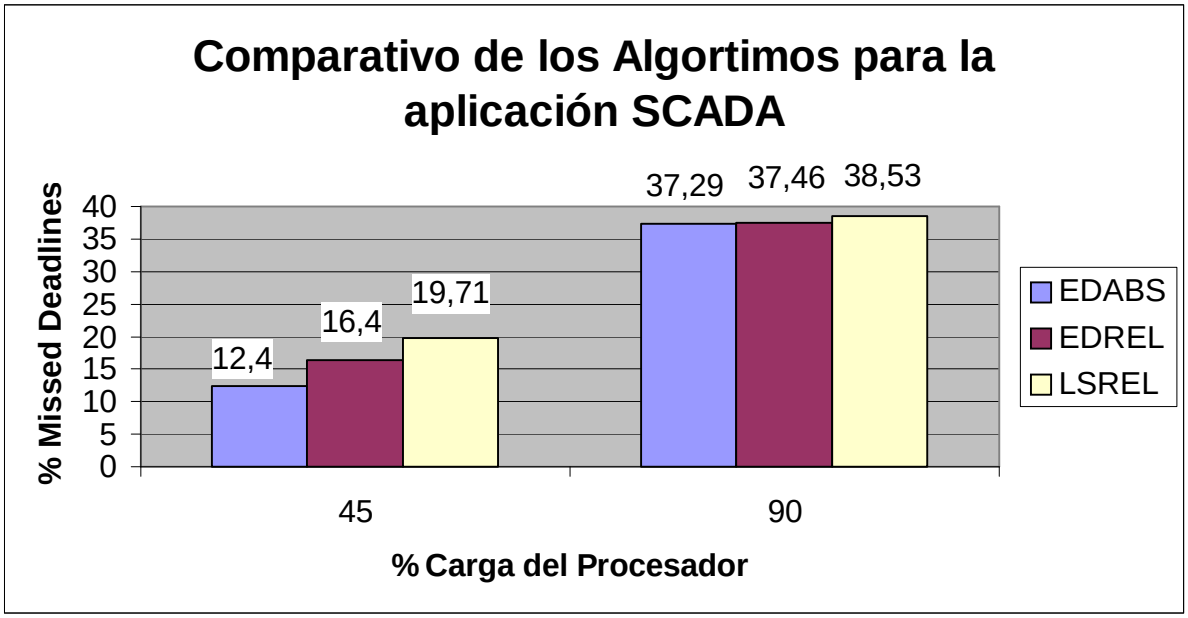


Figura 16 - Gráfico comparativo con los resultados promedio para la aplicación SCADA con N = 50 y N = 100.

El intervalo de mayor pérdida de *metas* ocurre en los primeros segundos o minutos en los cuales se va sobrecargando el sistema. Luego, cuando el conjunto de tareas está completo y continúa ejecutando por un largo período, cada aplicación tiende a estabilizarse en la proporción de *missed deadlines*. Es por ello, que las estadísticas se obtienen unos minutos después haber sobrecargado el sistema.

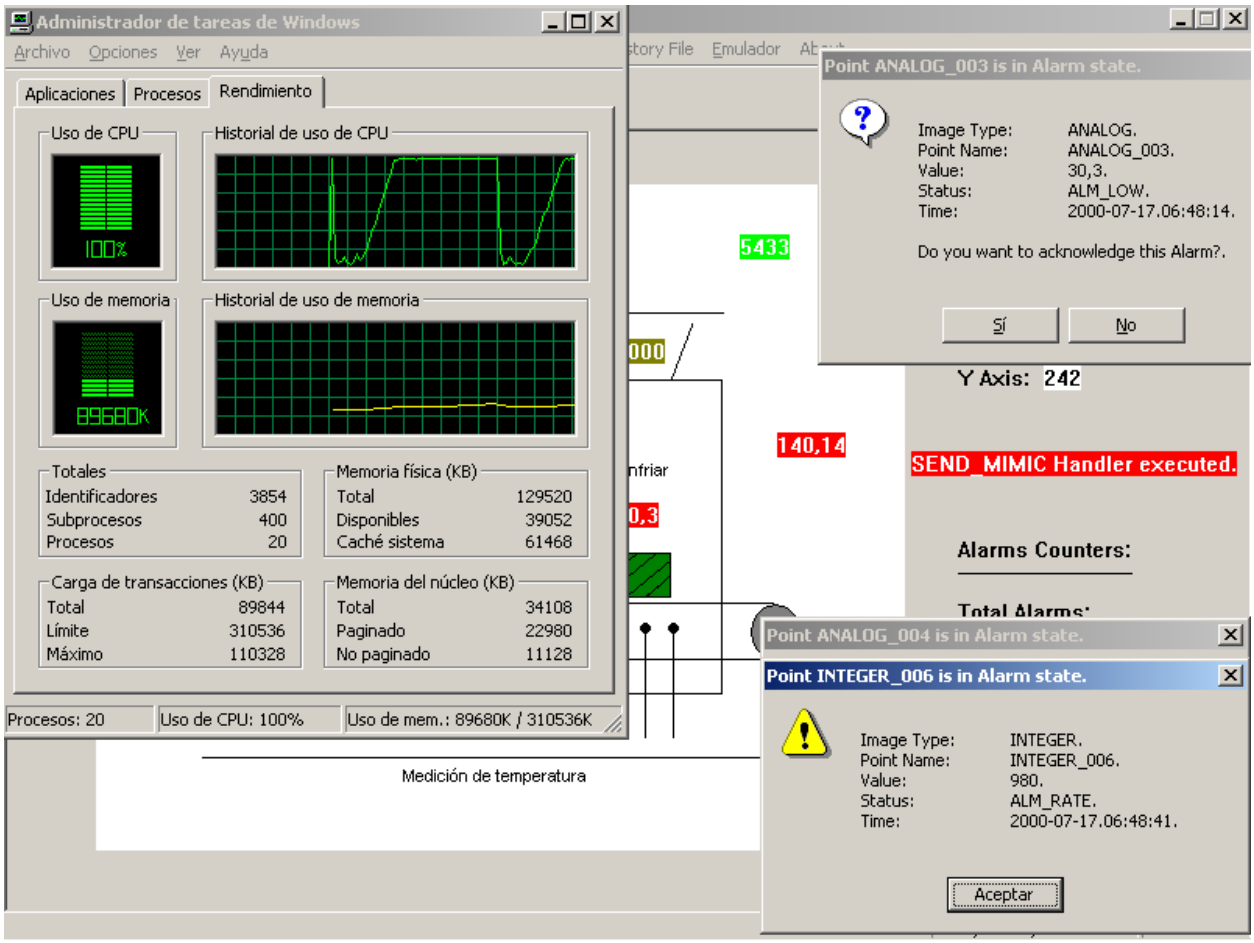
Como se puede observar, *EDABS* tiene el mejor desempeño de todos. Si bien los algoritmos se comportan en forma similar, este algoritmo demostró tener una mejor distribución de prioridades en situaciones donde el sistema está cargado un 50 % e incluso cuando el sistema está completamente saturado.

Es importante destacar nuevamente, que en situaciones normales de ejecución e incluso con sobrecargas esporádicas, la aplicación SCADA no evidencia prácticamente pérdidas en sus *metas*.

Estas pruebas realizadas son prácticamente los peores casos de exigencia sobre la aplicación SCADA, debido a que los estrictos requerimientos de tiempo de las Tareas de Tiempo Real, mientras que las *metas* de las tareas de *TestExecutionEngine* son más relajadas.

En la siguiente ventana, se muestra un foto de una de las pruebas, con diferentes intervalos de sobrecarga del 90% de utilización de la CPU, provocada por la

aplicación **TEE, N = 100**. En la práctica, este número asciende al 100% de carga, debido a que también se ejecuta la aplicación SCADA y al *overhead* introducido por el O/S:



9 - CONCLUSIONES Y TRABAJOS FUTUROS

El objetivo final de esta Tesis es el de proveer una plataforma común para la construcción de los Sistemas de Supervisión de Procesos, implementada con los servicios de planificación en Tiempo Real, emulados en un ambiente de un Sistema Operativo de Propósito General (GPOS).

Con ese objetivo como meta principal, en los primeros capítulos de este trabajo, se estudió la base teórica de los Sistemas de Tiempo Real y sus características. Luego, se enfocó el estudio en particular de los sistemas SCADA y su arquitectura. Una vez presentados esos capítulos introductorios, en el capítulo 3, se analizó el Software de Base que se requiere para los Sistemas de Tiempo Real y se empezó a describir la importancia de la planificación de tareas en esos sistemas.

Debido que la planificación en Tiempo Real es generalmente implementada en Sistemas Operativos específicos para Tiempo Real (RTOS), en el capitulo 4 se comenzó a estudiar como poder emular dicho comportamiento en un GPOS. Para ello, se eligió la plataforma Win32 de Microsoft®, debido a que cumple con las principales características que se requieren para los RTOS y, a su vez, provee un conjunto muy amplio de herramientas de desarrollo, debido a su presencia en el mercado y en la industria. También, en ese mismo capítulo, se estudiaron diferentes algoritmos de planificación en Tiempo Real y las diversas técnicas para implementar su emulación.

En los últimos capítulos, se describió el diseño e implementación de la **herramienta para construir sistemas SCADA, emulando la planificación en Tiempo Real**. Esta herramienta encapsula el funcionamiento común a todos los sistemas de Supervisión de Procesos, implementa los servicios de planificación de alto nivel, emula los algoritmos de planificación en Tiempo Real y provee una interface orientada a objetos, simple y consistente para el programador.

Una vez estudiado el diseño de la herramienta, en el capítulo 6 se describieron los detalles técnicos de implementación en la plataforma elegida y las ventajas para cada O/S en particular. En el siguiente capítulo se demostró la utilización de la herramienta y se implemento un sistema SCADA flexible y dinámico, el cual se puede configurar de acuerdo a los requerimientos del usuario.

Por último, en el capítulo 8, se realizaron diferentes pruebas con la aplicación SCADA y con la herramienta desarrollada.

Las pruebas de integración permitieron demostrar cómo se puede implementar una aplicación SCADA con todos sus niveles de servicio e interactuar con el mundo exterior. A su vez, las pruebas de desempeño mostraron la robustez de la emulación de los algoritmos y permitieron realizar un estudio exhaustivo de *missed deadlines* en situaciones de sobrecarga.

Luego de haber estudiado las características y requerimientos de los Sistemas de Tiempo Real y, posteriormente, haber implementado una herramienta para la construcción de sistemas SCADA en la plataforma Win32, se destacaron las siguientes conclusiones y recomendaciones más relevantes:

- **La herramienta desarrollada provee una solución genérica para la construcción de sistemas SCADA:**

  En la actualidad, no existen aún entornos de desarrollo que solucionen todos los problemas relacionados con los sistemas SCADA, lo que ha provocado que cada aplicación se construya utilizando técnicas específicas.

  Con ese panorama en vista, el objetivo de esta herramienta es el de proveer una capa de abstracción común en el diseño de los sistemas SCADA.

  La herramienta construida encapsula el nivel de servicio BSL y, de esta manera, provee un framework de desarrollo muy importante para todos los sistemas SCADA. Está diseñada como un conjunto de clases independientes, para permitir la construcción de diferentes aplicaciones de Supervisión de Procesos con distintas finalidades y con muy poco esfuerzo, facilitando de esta manera el trabajo a los diseñadores y programadores.

- **La arquitectura de un sistema SCADA facilita notablemente su construcción:**

  El desarrollo de un sistema SCADA diseñado con una arquitectura de 3 niveles de servicio bien diferenciados, facilita el análisis de sus diversas funciones y, de esta manera, simplifica notablemente su implementación.

  Esta arquitectura brinda la posibilidad de dividir el desarrollo del sistema SCADA, en donde cada grupo de trabajo se concentra en los detalles específicos de cada nivel de servicio.

- **El lenguaje C++ brinda muchas facilidades para Tiempo Real.**

  Si bien el lenguaje C++ es de propósito general, brinda toda la potencia de bajo nivel de su antecesor, el lenguaje C y, además, ofrece la posibilidad de encapsular el comportamiento en clases, conjuntamente con la mayoría de las ventajas que brinda la programación orientada a objetos.

  Otro punto muy importante, es que el código generado es ejecutable, altamente eficiente, muy liviano y con muy poco *overhead*, las cuales son características muy importantes para el desarrollo de Sistemas de Tiempo

Real. Estas facilidades no siempre están disponibles en otros lenguajes orientados a objetos, los cuales necesitan de una *máquina run-time* o tienen algún mecanismo de intérprete embebido.

Y por último, su tradicional característica de fuertemente tipado y de *static binding*, proveen un mecanismo muy importante para la prevención, detección y corrección de errores, mientras que en otros lenguajes, existen diversos tipos de errores que son detectados en tiempo de ejecución, los cuales pueden provocar fallas irrecuperables.

- **La herramienta y cualquier aplicación que utilice sus servicios, debería ejecutarse en los Sistemas Operativos Windows2000/NT.**

Esta recomendación se basa en el hecho que esta tecnología de Sistemas Operativos está basada en una arquitectura de 32-bit nativa, en contraposición a la línea Windows98/95, la cual provee compatibilidad total con el código existente de 16-bit y, por lo tanto, su *kernel* está escrito parte en 32-bit y otra parte en 16-bit.

Este beneficio de compatibilidad con las aplicaciones existentes de sus sistemas predecesores, tiene asociado un costo bastante alto: la inestabilidad de sus sistemas.

Las características de robustez y estabilidad de un O/S son fundamentales para los Sistemas de Tiempo Real y son provistas únicamente por la tecnología Windows2000/NT.

Inicialmente el sistema WindowsNT fue concebido para el ámbito empresarial, pero la evolución permanente de esta tecnología, conjuntamente con su nueva versión, Windows2000, han provocado su aceptación e implementación en prácticamente cualquier tipo de industria.

- **Selección de la *priority class* para ejecutar la herramienta.**

Luego de haber realizado diferentes pruebas en todos los Sistemas Operativos de Win32, en las cuales se configuraba el rango de prioridades y las Tareas de Tiempo Real con *metas* muy estrictas, se recomienda la siguiente configuración para la ejecución de aplicaciones de Tiempo Real:

  - En **Windows2000/NT** se recomienda **REALTIME_PRIORITY _CLASS**. Si bien este rango de prioridades es compartido con los threads del O/S, la robustez de estos Sistemas Operativos hace posible la utilización de esta *priority class* sin que impacte en la performance del sistema.

  - En **Windows98/95** se recomienda **HIGH_PRIORITY_CLASS**. En este rango de prioridades, ClashRT funciona correctamente y, en general, las aplicaciones de propósito general no utilizan esta *priority class*, sino que están configuradas para ejecutar en el rango de prioridades de NORMAL_PRIORITY_CLASS. En el caso que se requiera configurar para REALTIME_PRIORITY_CLASS, se evidenciará un deterioro notable en la respuesta del O/S.

- **La cantidad de niveles de prioridades para Tiempo Real.**

  La cantidad de niveles de prioridades disponibles para las aplicaciones de Tiempo Real es un factor muy importante, sobre todo si la aplicación a desarrollar se subdivide en muchas tareas independientes de tiempo crítico.

  Windows2000 es el único de todos los O/S de Win32 que provee 16 niveles para REALTIME_PRIORITY_CLASS. Es por ello, que se recomienda fuertemente este O/S, especialmente en aplicaciones que necesiten muchas tareas que ejecuten concurrentemente y con estrictos requerimientos de tiempo.

- **Se debe deshabilitar el mecanismo de ajuste dinámico de prioridades.**

  Como fue estudiado en esta Tesis, es muy importante que el O/S no altere la asignación de prioridades realizadas por los algoritmos de emulación. Esto provocaría que una tarea que era considerada de baja prioridad se convierta en una tarea con muy alta prioridad y, lo que es peor aún, que una tarea que sí era de alta prioridad, deje de serlo.

  Esta característica se aplica sólo en HIGH_PRIORITY_CLASS y solamente se puede deshabilitar en Windows2000/NT.

- **La exactitud del Planificador depende de la granularidad del *System Timer*.**

  La resolución con la que puede trabajar el Planificador depende exclusivamente del *System Timer*. Esto significa que una tarea podrá ser planificada a intervalos iguales o superiores a la granularidad del *System Timer*.

  Aunque a las Tareas de Tiempo Real se les puede asignar cualquier período de activación, el Planificador sólo se ejecutará en intervalos múltiplos de la resolución del *System Timer*.

  Como fue estudiado en el capítulo 3, el *System Timer* en Windows2000/NT tiene una resolución de 10 ms. y en Windows98/95, 55 ms.

Claramente, mayor es este intervalo, peor será la exactitud con la que el Planificador podrá señalizar a las tareas.

- **El *intralevel scheduling* es un factor a considerar en los Sistemas de Tiempo duros.**

  En situaciones en donde se ejecutan aplicaciones de Tiempo Real con requerimientos de tiempo duros, se deberá analizar la posibilidad de utilizar la política FIFO en *intralevel scheduling*. Esta política permite que un thread pueda ejecutar en forma *non-preemptable*, hasta que termine su ciclo de ejecución, independientemente de la longitud del *time-slice* del O/S. En estos casos, el thread podrá ser desalojado sólo si existe un thread con prioridad más alta. En caso que el thread esté ejecutando con la prioridad más alta, sólo podrá ser desalojado por una interrupción de un nivel superior.

  Esta técnica tiene sus riesgos asociados, ya que un thread puede monopolizar el recurso del procesador, provocando una inestabilidad total en el sistema hasta dejarlo sin respuesta.

  Como fue estudiado en el capítulo 3, Windows2000 provee la posibilidad de configurar el *intralevel scheduling* con la política FIFO.

- **Es importante la relación entre los niveles de prioridades y el parámetro R.**

  Para sintonizar los algoritmos de emulación, se realizaron diferentes pruebas de desempeño con distintos parámetros. Es importante destacar, que la relación entre los parámetros R y n, afectan la asignación de prioridades a las tareas. El parámetro n, es fijo y está dado por el O/S, con lo cual, el único que se puede variar es R.

  El valor de R deberá ser, como mínimo, la distancia máxima entre el tiempo de *meta* y el tiempo de llegada de todas las tareas. Cuanto mayor es este número, mas grande serán los intervalos en donde se asignarán las prioridades. Por lo tanto, las tareas que tengan *metas* relativamente distantes, se les asignará la misma prioridad. En situaciones donde se requiere tener una gran resolución para la asignación de prioridades, se debe elegir el mínimo valor posible para R.

- **El futuro de la emulación de planificación en Tiempo Real.**

  En la actualidad, las aplicaciones y tareas que requieren respuestas en Tiempo Real está en permanente crecimiento. Esta gran variedad de aplicaciones, no sólo se restringe a los sistemas tradicionales de Tiempo Real, sino que además, se han agregado un nuevo tipo de aplicaciones y

servicios. Esto se debe principalmente a la evolución de Internet, a la nueva clase de aplicaciones *on-line* y a la necesidad constante de asegurar un tiempo de respuesta dentro de límites predefinidos.

A su vez, muchas de estas aplicaciones, especialmente las que tienen tareas con *metas* blandas o firmes, se desarrollan en Sistemas Operativos con múltiples propósitos. Tanto los sistemas SCADA para controlar y supervisar una planta, como aplicaciones financieras de Internet, en las cuales se brinda la evolución de los mercados en Tiempo Real, requieren de la emulación de planificación en Tiempo Real.

En general, estas nuevas aplicaciones operan en un entorno que no es de Tiempo Real y será esencial desarrollar nuevos modelos y técnicas para poder emular los algoritmos originales, los cuales garantizan el cumplimiento de los requerimientos de Tiempo Real.

La necesidad de emular la planificación en Tiempo Real, será un requerimiento muy importante en el diseño y construcción de las aplicaciones del futuro.

Durante el desarrollo de la herramienta, se han experimentado diferentes problemas, los cuales son importantes mencionar.

En primer lugar, trabajar en un ambiente *multithreading* genera bastantes complicaciones. Esto ocurre simplemente porque muchas veces es difícil detectar los errores de programación, debido a la ejecución concurrente de diferentes threads.

Los problemas de concurrencia son inherentes a los Sistemas de Tiempo Real y abarcan desde los errores de sincronización y comunicación de tareas, hasta los errores de programación propios de un thread, los cuales provocan resultados erróneos en todo el sistema.

Para este clase de programas, es fundamental utilizar un *debugger multithreading*, ya que de otra manera sería muy difícil realizar el seguimiento de la ejecución de una aplicación con múltiples tareas.

Otro de los errores típicos de programación son los problemas de *memory leaks*. Esto ocurre cuando un programa solicita memoria dinámicamente al O/S y cuando termina de utilizarla, nunca la libera. Este problema puede ser muy grave en un Sistema de Tiempo Real debido a que en su ejecución de tiempo infinito, puede llegar a consumir todos los recursos disponibles y, en consecuencia, provocar su irremediable falla.

Esta clase de errores son muy difíciles de detectar de dónde provienen debido a la ejecución concurrente de todos los threads. Una de las técnicas para minimizar la

cantidad de errores en un ambiente *multithreading*, es dividir el desarrollo de la aplicación en los módulos que se ejecutan en forma concurrente de los que proveen funcionalidad común a todas las tareas.

Utilizando esta metodología facilitó notablemente la solución de todos los problemas descriptos precedentemente. Para ello, primero se realizó el desarrollo, debugging y testing del *Motor de Ejecución*, con el Planificador, sus Tareas de Tiempo Real y los algoritmos de emulación. Una vez que ese núcleo de servicios había sido depurado totalmente, se comenzó a realizar el mismo trabajo con las *Tablas de Representación* y sus respectivos métodos de administración. Y, por último, se realizó la integración completa de la herramienta con el *Motor de Ejecución* y todas sus Clases.

Como una ampliación a esta Tesis, se podrían plantear los siguientes trabajos futuros:

- Portar la herramienta construida a otro Sistema Operativo, como podría ser Linux.

- Desarrollar los niveles de servicio ISL y PSL para que ejecuten en otro nodo de una LAN, los cuales se comunican con el nivel de servicio BSL, vía un handler de TCP/IP. Este desarrollo *cliente/servidor* permitiría utilizar la herramienta en un ambiente distribuido. En este caso, se pueden desarrollar N *clientes*, los cuales cada uno ejecuta en su respectiva plataforma, pero todos ellos acceden siempre al mismo *servidor* BSL.

- Desarrollar nuevas funciones de supervisión, combinadas con técnicas de Inteligencia Artificial, para ejecutar acciones correctivas en forma automática. Además, se pueden brindar herramientas que provean al operador diferentes niveles del estado de las operaciones: generación de reportes, estadísticas e informes con gráficos.

- Incluir diferentes interfaces de hardware con sus respectivos drivers, para facilitar y ampliar el uso de la herramienta.

- Incluir la emulación de otros algoritmos de planificación en Tiempo Real y comparar sus respectivos desempeños.

- Modificar los algoritmos de emulación para incluir diferentes variantes. Una posibilidad sería detectar sobrecargas del sistema y poder modificar las prioridades de las tareas para asegurar que las aquellas de tiempo crítico siempre cumplen sus *metas*. Otra modificación podría ser implementar algún mecanismo de reasignación de prioridades por falta de niveles disponibles. Esta variante sería útil cuando, por ejemplo, se ejecutan muchas tareas concurrentemente y sus tiempos de llegada son aleatorios. En estos casos,

hay que redistribuir la asignación de prioridades ya que algunas tareas que eran consideradas de relativamente alta prioridad ya no lo sean, debido a que ingresaron al sistema nuevas tareas con requerimientos más estrictos.

- Modificar el Planificador para permitir una estimulación doblemente cruzada. Esto significa que no sólo el Planificador envía una señal a las tareas para poder comenzar su ciclo de ejecución, sino que también las tareas deberían señalizar al Planificador cuando terminaron su ciclo o tuvieron algún problema. Esta información podrá ser utilizada por el Planificador para decidir si continúa la ejecución de la tarea o si decide abortar finalmente su ejecución.

REFERENCIAS Y BIBLIOGRAFÍA

[Ade94] Adelberg, Brad; García-Molina, Héctor y Kao, Ben. *Emulating Soft Real-Time Scheduling Using Traditional Operating System Schedulers*. Stanford University, 1994.

[Ben97] Benítez, Silvia V.; Seoane, Juan J.; Wainer, Gabriel; Bevilacqua, Roberto J. G. *IGNATIUS, Una herramienta para Construir Sistemas de Supervisión en Tiempo Real.* Universidad de Buenos Aires, 1997.

[Bev91] Bevilacqua, R., *Arquitecturas del Procesador y Sistemas Operativos*. Departamento de Computación de la Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, 1991.

[Liu99] Liu, J. W. S.; Rajkumar, R.; Deng, Z.; Seri M.; Frei A.; Zhang L. and Shih C. S. *How to get most predictability out of Windows NT*. Illinois University, 1999.

[Msb00] MSDN (Microsoft Software Development Network*). Backgrounders & Technical Articles*. 2000.

[Msk00] MSDN (Microsoft Software Development Network). *Knowledge Base*. 2000.

[Msr00] MSDN (Microsoft Software Development Network). *Windows95/98/NT/2000, Resource Kits*. 1995-2000.

[Msv00] MSDN (Microsoft Software Development Network). *Visual Development Studio 6.0, Specification & Documentation*. 2000.

[Msw00] MSDN (Microsoft Software Development Network). *Win32 Platform Software Development Kit, Specification & Documentation*. 2000.

[Str91] Stroustrup, Bjarne. *The C++ Programming Language,* Second Edition. New Jersey, Watcom, 1991.

[Ker84] Kernigan, Brian W.; Pike, Rob. *The Unix Programming Environment*. Prentice-Hall, 1984.

[Tan87] Tanenbaum, A. *Operating Systems – Design and Implementation.* Prentice-Hall, 1987.

[Tan90] Tanenbaum, A. *Computer Organization*. New Jersey, Prentice-Hall, 1990.

[Tan93] Tanenbaum, A. *Modern Operating Systems.* Prentice-Hall, 1993.

[Tan96] Tanenbaum, A. *Computer Networks, Third edition.* Prentice-Hall, 1993.

[Uml99] OMG (Object Managmente Group*). UML, The Unified Modeling Language Specification & Documentation, Version 1.3*. 1999.

[Wai93] Wainer, Gabriel. *SSDT: Una herramienta para Desarrollar Sistemas Supervisores en Tiempo Real*. La Serena, Encuentro Chileno de Computación, 1993.

[Wai95] Wainer, Gabriel. *Sistemas de Tiempo Real*. Departamento de Computación de la Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, 1995.

[Wai97] Wainer, Gabriel. *Sistemas de Tiempo Real, Conceptos y Aplicaciones*. Bs. As., Nueva Librería, 1997.

[Wai99] Wainer, Gabriel. *Concurrencia en Sistemas Operativos*. Publicación de trabajos de investigación en el Web Site del Departamento de Computación de la Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, 1999.

[War86] Ward, J.; Mellor, P. *Structured Development for Real Time Systems*. Yourdon Press, 1986.