Universidad de las Ciencias Informáticas

TRABAJO DE DIPLOMA PARA OPTAR POR LA CATEGORIA DE INGENIERO INFORMATICO

Título: Construcción de un Ejecutivo de Tareas.

Autor: Yunior Peralta González.

Tutor: Ing. René López Baracaldo.

Curso 2008-2009

Síntesis

Existen aplicaciones que deben realizar un gran número de tareas que pueden acceder a recursos compartidos. Estas tareas compiten por la obtención de estos recursos y es necesario decidir un orden para la ejecución de las mismas. Para el control de las tareas se presenta la elaboración de un sistema ejecutivo que permite realizar una planificación determinada de todas las tareas que deben ser ejecutadas en el sistema. El ejecutivo que se presenta permite administrar las tareas de forma eficiente haciendo un buen uso de los recursos disponibles, y ampliar sus funcionalidades mediante la incorporación de extensiones. Además, es capaz de remplazar las políticas de planificación empleadas por otras que se ajusten mejor a las circunstancias y presenta la capacidad de procesar tareas de tiempo real.

Índice

Introducción....4
Capítulo 1: Fundamentación Teórica....7
1.1 Definición de ejecutivo....7
1.2 Sistemas de tiempo real....8
1.3 Objeto Activo....12
1.4 Asignación de memoria....12
1.5 Continuación....13
1.6 Bytecode....13
1.7 Opcode....14
1.8 Máquina Virtual....14
1.9 Desarrollar una Extensión....15
1.10 Cargar de forma dinámica una biblioteca compartida....16
Capítulo 2: Desarrollo de la Solución....18
2.1 Requisitos....18
2.2 Definición de Tarea....18
2.3 Momento de activación de las Tareas....18
2.4 Atributos de las Tareas....19
2.5 Código de las Tareas....19
2.6 Definición de Instancia de una Tarea....20
2.7 La Pila....20
2.8 La Continuación....21
2.9 Las Instrucciones....21
2.10 Tareas generadas por eventos de usuarios....21
2.11 Contexto de Ejecución....22
2.12 Instrucciones Síncronas y Asíncronas....22
2.13 Instrucciones del Núcleo....23
2.14 Módulos Externos....23
Capítulo 3: Desarrollo de Extensiones....25
3.1 Las extensiones....25
3.2 ¿Cómo desarrollar una extensión para el paralelismo?....26
3.3 ¿Cómo desarrollar una extensión para la planificación?....27
3.4 ¿Cómo desarrollar una extensión para las instrucciones?....28
Capítulo 4: Demostrativo....31
Conclusiones....33
Glosario de Términos....34
Referencias bibliográficas....35
Bibliografía....36

Introducción

En el polo de Hardware y Automática de la Universidad de las Ciencias Informáticas existen proyectos que realizan un conjunto de acciones que presentan puntos en común. Ejemplo de algunas de estas actividades lo constituyen: la lectura y escritura de variables en dispositivos, la persistencia de las variables recolectadas, ya sea en archivos con determinada estructura o en una base de datos, el envío de información de un lugar a otro, la compresión de datos, la generación de nuevos valores a partir de las variables ya encuestadas de los dispositivos, la generación de distintos tipos de alarmas, la planificación de las acciones a realizar, el trabajo de algunas de estas actividades en paralelo, entre otras.

Lo anteriormente expuesto se puede ilustrar con el siguiente ejemplo:

Uno de los proyectos del polo consiste en un módulo de recolección de variables. En el mismo se realiza la lectura y escritura de variables en los dispositivos de campo. Estas variables tienen asociadas un período de muestreo que es usado por el módulo de recolección para realizar una planificación de las lecturas que deben ser realizadas. Una vez que se realicen estas lecturas, el módulo debe transmitirlas hacia otro módulo para su procesamiento.

Otro de los proyectos del polo consiste en un módulo de procesamiento de variables, denominado por algunos autores como “Base de Datos de Tiempo Real”. En este módulo se procesan las variables para determinar si alguna de ellas debe generar algún tipo de alarma. Este procesamiento debe hacerse en tiempo real y también usa un planificador para la determinación del orden en que deben ejecutarse las tareas. Por último algunas de estas variables deben ser enviadas hacia otro módulo que es el que garantiza la persistencia de las mismas.

Entre las acciones principales que deben realizar estos dos módulos, existen dos de ellas que se repiten en ambos: implementación de un sistema de planificación y envío de variables hacia otros módulos.

De forma general, cada uno de los proyectos del polo que presentan algunas de estas acciones comunes, le dan una implementación específica para resolver solamente sus propias necesidades. Debido a esto, una misma funcionalidad posee diferentes implementaciones en diferentes proyectos.

Esto trae consigo un mayor esfuerzo de los desarrolladores al tener a varios grupos no sólo desarrollando sobre la misma funcionalidad, sino además dándole mantenimiento, lo que trae como consecuencia una mala planificación de los recursos humanos y un gasto de tiempo innecesario por parte de sus miembros.

Tomando esto como base la dirección del polo ha decidido que es necesario contar con una solución que permita agrupar toda la lógica que es común entre estas actividades y permita además extender su funcionalidad para añadirle las particularidades que son específicas de cada proyecto.

Para la realización de este conjunto de actividades se debe contar con un ejecutivo que sea capaz de hacer un control y planificación de las mismas. Los ejecutivos son aplicaciones que permiten la planificación, ejecución y monitoreo de un conjunto de tareas haciendo un uso eficiente de los recursos con los que cuenta el sistema.

Por lo anteriormente expuesto se define como **problema a resolver** ¿cómo desarrollar un ejecutivo que permita agrupar la lógica común de las actividades del subsistema de adquisición de cada proyecto?

Según el problema expuesto anteriormente se define como **objeto de estudio** a los sistemas ejecutivos.

Para dar solución al problema planteado se ha trazado como **objetivo general**: Desarrollar un ejecutivo que permita agrupar la lógica común de las actividades del subsistema de adquisición y que sea extensible con las particularidades de cada proyecto.

El **campo de acción** del presente trabajo lo constituyen los ejecutivos de tiempo real extensibles.

Para cumplir con el objetivo trazado se definieron las siguientes **tareas**:

- Revisar bibliografías acerca de las características de las aplicaciones de tiempo real.
- Realizar un estudio acerca del uso de las más recientes técnicas y herramientas de programación concurrente.
- Investigar acerca de los mecanismos existentes para la administración de la memoria.
- Analizar las tecnologías existentes para el desarrollo de un sistema ejecutivo.
- Desarrollar un sistema ejecutivo.

- Realizar un demostrativo en el que se ponga en práctica la funcionalidad del ejecutivo.

Para el cumplimiento de estos objetivos se llevan a cabo varios **métodos y técnicas en la búsqueda y procesamiento de la información** como son:

Métodos Teóricos.

- **Método analítico-sintético:** Para el estudio de las concepciones y los conceptos empleados en los sistemas ejecutivos, analizando los documentos elaborados por desarrolladores, para la extracción de los elementos más importantes.
- **Método histórico-lógico:** Para la comprensión de los antecedentes y las tendencias actuales referidas a la evolución en el mundo de los sistemas ejecutivos.

Métodos Empíricos.

- **Experimento:** Elaboración de un ejecutivo para la administración y control de las tareas.

Capítulo 1: Fundamentación Teórica

1.1 Definición de ejecutivo

Un ejecutivo de tiempo real es un núcleo que incluye bloques de memoria privatizados, servicios de entrada y salida, y otras características complejas. La mayoría de los núcleos comerciales de tiempo real son ejecutivos. Por último, un sistema operativo es un ejecutivo que provee, para una interfaz de usuario generalizada, seguridad y un sistema administrador de archivos. [1]

Un sistema ejecutivo es una serie de subprogramas que son controlados por un “manejador”, una subrutina especial que puede reconocer palabras claves de tareas y entonces dirigir el control hacia los subprogramas apropiados para su análisis. Los pasos que se requieren para programar un ejecutivo, incluyen: (1) definición de las palabras claves para la identificación de las tareas, (2) implementación de la estructura de control de ramificaciones para alcanzar los subprogramas apropiados, y (3) construcción de subrutinas de manipulación de datos y la definición de formatos de datos reconocidos por estas rutinas. [2]

Existen tres arquitecturas de software principales que se usan en los sistemas operativos de tiempo real. El ejecutivo de tiempo real, el núcleo monolítico y el micronúcleo. El ejecutivo de tiempo real es compilado como un binario enorme monolítico que contiene todas las funcionalidades requeridas que se encuentran normalmente en un sistema operativo y la aplicación. [3]

La principal unidad operativa en un ejecutivo de tiempo real es una “Tarea”. [4]

Los ejecutivos son aplicaciones que permiten dado un conjunto de tareas que compiten por un número restringido de recursos de ejecución hacer un uso eficiente de la asignación de estos recursos a cada tarea. Para esto los ejecutivos se valen de un planificador, que es quien define cuales son las políticas de planificación a seguir en la aplicación. Las tareas pueden estar sujetas a múltiples restricciones como: un valor que indique un cierto nivel de prioridad, el tiempo en el cual deben comenzar su ejecución, el tiempo de duración, entre otras, lo cual conlleva a que para asignarle un recurso para su ejecución hay que tener en cuenta determinados criterios para la planificación como: atender primero a las tareas cuyo valor indicativo del nivel de prioridad sea mayor, atender primero a las tareas con menor tiempo de duración, etc. Los algoritmos empleados para esta asignación dependen de los intereses de la solución.

1.2 Sistemas de tiempo real

En una computadora de tiempo real que controla dispositivos o procesos, los sensores proveerán lecturas en intervalos periódicos y la computadora debe responder enviando señales a los encargados de realizar dicha operación. Pueden haber eventos inesperados o irregulares y estos eventos pueden recibir respuestas. En todos los casos, la respuesta se dará en un límite de tiempo. La habilidad de la computadora para procesar todos estos pedidos depende de su capacidad para ejecutar las operaciones necesarias en el tiempo dado. Si varios eventos ocurren casi al mismo tiempo, la computadora necesitará planificar los cálculos para que cada respuesta se realice dentro del marco de tiempo requerido. Puede ocurrir que el sistema sea incapaz de realizar todas las posibles demandas que ocurren de forma inesperadas. En este caso se dice que el sistema escasea de suficientes recursos; un sistema con recursos ilimitados y con una velocidad de procesamiento infinita, podría satisfacer cualquier restricción de tiempo. No cumplir con la restricción de tiempo para una respuesta puede tener diferentes consecuencias; puede no haber ningún efecto, los efectos pueden ser menores o corregibles, o los resultados pueden ser catastróficos. Cada tarea que ocurre en un sistema de tiempo real presenta ciertas propiedades de tiempo. Estas propiedades de tiempo se pueden considerar cuando se planifican tareas en un sistema de tiempo real. Las propiedades de tiempo de una tarea dada se refieren, de forma general, a los siguientes aspectos:

- **Tiempo de activación:** tiempo en el que la tarea está lista para su ejecución.
- **Tiempo de vencimiento:** tiempo en el que la tarea debe haber finalizado, una vez que la tarea haya sido activada.
- **Demora mínima:** tiempo mínimo que debe transcurrir antes de que comience la ejecución de la tarea, una vez que la tarea haya sido activada.
- **Demora máxima:** máximo tiempo permitido que debe transcurrir antes de que comience la ejecución de la tarea, una vez que la tarea haya sido activada.
- **Peor caso de tiempo de ejecución (o de respuesta):** máximo tiempo que toma terminar la tarea, una vez que la tarea haya sido activada.
- **Tiempo de ejecución:** tiempo que toma completar la tarea sin interrupción, una vez que la tarea haya sido activada.
- **Prioridad (o peso):** urgencia relativa de la tarea.

Los sistemas de tiempo real abarcan un amplio espectro de complejidad desde micro controladores muy simples hasta sistemas altamente sofisticados, complejos y distribuidos. Algunos ejemplos de sistemas de tiempo real incluyen sistemas de control de procesos, sistemas de control de vuelos, aplicaciones de elaboración flexible, robótica, sistemas inteligentes de carretera, y sistemas de alta velocidad y comunicación multimedia.

Un sistema de tiempo real usualmente tendrá que atender muchas demandas en un tiempo limitado. La importancia de las demandas pueden variar con su naturaleza o con el tiempo disponible para la respuesta. Es por esto que la asignación de los recursos del sistema necesita ser planificada para que todas las demandas sean atendidas antes de su fecha de vencimiento. Esto usualmente se logra usando un planificador, quien implementa políticas de planificación que determinan como se le asignan al programa los recursos del sistema. Las políticas de planificación pueden ser analizadas de forma matemática para que la precisión de la especificación formal y los estados del desarrollo del programa se puedan complementar por un análisis matemático de tiempo de las propiedades del programa.

Los sistemas de tiempo real se definen como aquellos sistemas en los que el buen funcionamiento del mismo no sólo depende del resultado lógico del cómputo sino también del tiempo en el que se producen los resultados. Si no se cumplen las restricciones de tiempo del programa, se dice que ha ocurrido una falla en el sistema. Es por esto que es de suma importancia que en un sistema se garantice el cumplimiento de las restricciones de tiempo. Para garantizar comportamientos de tiempo el sistema tiene que ser predecible. Predicción significa que cuando una tarea comience es posible determinar su tiempo de culminación con certeza.

Una aplicación de tiempo real está normalmente compuesta por múltiples tareas con distintos niveles de prioridades. Aunque no cumplir los plazos de culminación de las tareas no es algo deseable en un sistema de tiempo real, algunas tareas de tiempo real suave pueden no satisfacer estas restricciones de tiempo y aún así el sistema seguiría funcionando correctamente. Sin embargo, esta violación de restricciones de tiempo conducirá a tener que pagar algunas consecuencias no deseadas. Por otra parte, las tareas de tiempo real duro no pueden violar ninguna de estas restricciones de tiempo, de lo contrario, en el sistema se producirán resultados fatales. Existe otro grupo de tareas de tiempo real, conocidas como tareas de tiempo real estricto, las cuales mientras más temprano terminen de realizarse, mejor se comportará la aplicación.

Se define de forma formal un sistema de tiempo real como sigue:

Un sistema compuesto por un conjunto de tareas, $T = \{T_1, T_2, \ldots, T_n\}$, donde el peor caso de tiempo de ejecución de cada tarea $T_i \in T$ es C_i, se dice que es un sistema de tiempo real si existe al menos una tarea $T_i \in T$ que cumpla con una de las siguientes categorías.

1. La tarea T_i es una tarea de tiempo real duro (su ejecución debe completarse antes de su tiempo de vencimiento).
2. La tarea T_i es una tarea de tiempo real suave (mientras más se demore la tarea en terminar

su ejecución después de su tiempo de vencimiento D_i, mayor es la penalización que tiene que pagar). Se define para la tarea una función de penalización $P(T_i)$. Si $C_i \leqslant D_i$, la función de penalización $P(T_i)$ es cero. De lo contrario $P(T_i) > 0$. El valor de la penalización es una función creciente de $C_i - D_i$.

3. La tarea T_i es una tarea de tiempo real estricto (mientras más rápido la tarea termine su ejecución antes de su tiempo de vencimiento D_i, más bonificaciones gana). Se define para la tarea una función de bonificación $R(T_i)$. Si $C_i \geqslant D_i$, la función de bonificación $R(T_i)$ es cero. De lo contrario $R(T_i) > 0$. El valor de la bonificación es una función creciente de $D_i - C_i$.

El conjunto de tareas de tiempo real $T = \{T_1, T_2, ..., T_n\}$ puede ser una combinación de tareas de tiempo real duro, estricto y suave.

La función de penalización para el sistema es la suma de todas las penalizaciones de las tareas de tiempo real suave y la función de bonificación para el sistema es la suma de todas las bonificaciones de las tareas de tiempo real estricto.

Para un conjunto dado de demandas, el problema general de la planificación es establecer un orden para ejecutarlas de forma tal que se cumplan con las restricciones de cada una de las demandas. De forma general una demanda se caracteriza por su tiempo de activación, tiempo de ejecución, tiempo de vencimiento y recursos que requiere. La ejecución puede o no ser interrumpida. Sobre el conjunto de tareas puede existir una relación de precedencia la cual es una de las restricciones a tomar en cuenta cuando se establece el orden para la ejecución, o sea, una tarea no puede ejecutarse hasta que todas las tareas de las cuales ella depende se hayan terminado de ejecutar. Básicamente el problema de la planificación es determinar un orden para ejecutar las tareas antes de su tiempo de vencimiento.

La planificación apropiada para un sistema de tiempo real se diseña basada en las propiedades del sistema. Estas propiedades pueden ser:

- **Tareas de tiempo real suave, duro o estricto:** las tareas se clasifican en tareas de tiempo real suave, duro o estricto.
- **Tareas periódicas, aperiódicas y esporádicas:** todas las tareas son generadas mediante eventos o señales, las tareas periódicas y las aperiódicas se activan mediante eventos de tiempo, sólo difieren en que las periódicas se generan de forma regular tras transcurrir un determinado tiempo (período de la tarea) y generalmente presentan restricciones que indican que debe ejecutarse una instancia de la tarea por cada período, mientras que las aperiódicas

sólo se generan una vez (en su hora de comienzo). Por último, las tareas esporádicas son generadas mediante eventos impredecibles, o sea, eventos que se generan al azar por diversos motivos y de los cuales no pueden calcularse su tiempo de activación con antelación, como por ejemplo: al presionarse un botón, al cumplirse una determinada condición en el programa, entre otros.

- **Tareas interrumpibles o no interrumpibles:** las tareas no interrumpibles son aquellas que una vez que comiencen a procesarse no se detienen hasta que no hayan completado su ejecución. Sin embargo en algunos algoritmos de planificación de tiempo real, una tarea debe ser interrumpida si otra con mayor nivel de prioridad está lista para ejecutarse. Es por esto que en estos tipos de escenarios es necesario que las tareas puedan ser detenidas antes de su culminación.
- **Sistemas monoprocesadores o multiprocesadores:** El número de procesadores disponibles es uno de los principales aspectos a tener en cuenta cuando se planifica un sistema de tiempo real. En sistemas multiprocesadores, los algoritmos de planificación deben prevenir el acceso simultáneo a recursos y dispositivos compartidos. Además se debe proveer la mejor estrategia para reducir el costo de la comunicación.
- **Tareas con prioridades fijas o dinámicas:** en algoritmos de planificación orientados a prioridades, a cada tarea se le asigna un nivel de prioridad. Esta asignación puede ser de forma fija (en caso de que la prioridad se conozca a priori) o de forma dinámica (en caso de que la prioridad sólo se conozca en tiempo de ejecución y su valor puede variar en el transcurso de la aplicación).
- **Sistemas fijos o flexibles:** para la planificación de sistemas en tiempo real se necesita tener suficiente información acerca de cada una de las tareas, como demora mínima y máxima, fecha de vencimiento, entre otras. La mayoría de los sistemas asumen que gran parte de esta información se encuentra disponible a priori, y por tanto, se basan en un diseño fijo. Sin embargo existen otros sistemas de tiempo real que son diseñados de forma dinámica y flexible.
- **Tareas dependientes o independientes:** existen tareas que para su ejecución requieren recibir información de otras tareas. Es por esto que hay tareas que dependen de otras para ejecutarse. Este es el concepto de dependencia de tareas. Las tareas dependientes usan información compartida o datos de comunicación para transferir información generada por una tarea y requerida por otras. Para estos casos la planificación tiene que tener en cuenta la dependencia que existen entre las tareas del sistema.

Para que el planificador pueda hacer una planificación correcta de tareas con recursos y dispositivos compartidos en un sistema multiprocesador es necesario emplear mecanismos para prevenir el

acceso simultáneo de estas tareas a dependencias compartidas. Para esto los sistemas operativos brindan funcionalidades como los semáforos para la sincronización entre procesos. Sin embargo la programación con estos tipos de funcionalidades requiere un diseño más complejo y generalmente se cometen algunos errores en el programa difíciles de corregir posteriormente. Además, el costo de la comunicación usando estas funcionalidades trae consigo no sólo una penalización en el rendimiento de la aplicación sino además un desgaste en el carácter predecible de la misma.

1.3 Objeto Activo

Para la eliminación de las complejidades creadas por las primitivas provistas por los sistemas operativos, se ha creado un mecanismo denominado **Objeto Activo**, cuya funcionalidad es similar pero con la diferencia que las operaciones que acceden a recursos compartidos se almacenan en una lista de solicitudes en la cual las operaciones se extraen una a una, o sea, de forma que sólo se atienda una a la vez, eliminando así los peligros ocasionados por la concurrencia entre procesos, como: competencia, abrazo fatal, inanición e inversión de prioridades. La principal ventaja de este mecanismo es que separa la invocación de una operación de la ejecución de la misma para mejorar la concurrencia y simplificar el acceso sincronizado de un objeto residente en su propio hilo de control. Este patrón permite además que las operaciones que se encuentran en la lista de solicitudes se invoquen en un orden determinado y no en el orden que fueron insertadas en la lista, por lo que es posible especificarle ciertas políticas de planificación para evitar que las tareas más prioritarias tengan que esperar por la ejecución de tareas con niveles más bajos de prioridad.

1.4 Asignación de memoria

En las aplicaciones de tiempo real, el proceso de asignación de memoria es crítico. Primero, porque el tiempo de la asignación es importante ya que esta debe ocurrir en un tiempo conocido a priori para garantizar que la aplicación sea predecible, a diferencia de las aplicaciones que no son de tiempo real, las cuales generalmente hacen una búsqueda en una lista de longitud indeterminada para encontrar un bloque de memoria disponible. Segundo, porque la memoria puede fragmentarse a medida que las regiones de memoria que se liberan se van separando de las regiones de memoria que se encuentran en uso. Esto puede traer como consecuencia que el programa deje de funcionar correctamente debido a que no encuentra memoria disponible que asignar, incluso, cuando realmente sí disponga de memoria libre que puede ser usada por el programa. Los algoritmos de asignación de memoria que acumulan pequeñas fragmentaciones pueden funcionar bien para ordenadores que son reiniciados de forma periódica mediante pequeños intervalos de tiempo, dígase una semana o un mes; pero son inaceptables para aplicaciones que deben estar ejecutándose por años sin reiniciarse.

Los algoritmos basados en una cantidad fija de bloques de memoria, en el que cada bloque presente

además un tamaño fijo, funcionan bien para pequeñas aplicaciones de tiempo real que pueden mantenerse en ejecución durante largos períodos de tiempo. [3]

1.5 Continuación

Como ya fue abordado con anterioridad, el planificador debe ser capaz de interrumpir una tarea que se esté procesando si otra con mayor nivel de prioridad está lista para su inmediata ejecución. Una vez que una tarea se interrumpe, esta pasa a una lista de tareas interrumpidas hasta que el planificador decida que debe continuar con su ejecución. Cuando una tarea de las que estaban interrumpidas pasa nuevamente al estado de ejecución, esta no debe empezar la ejecución desde el principio, sino desde la instrucción que le sigue a la última que ejecutó. Este fenómeno es conocido con el nombre de “Continuación”.

Una continuación es una representación abstracta del estado de control, o el “resto del cómputo” o “resto del código que va a ser ejecutado”. Esta representación está muy ligada a conocer cual parte del programa se está procesando: cual función y cual línea se está ejecutando actualmente. La “continuación actual” son las instrucciones que serán ejecutadas después de que la línea de código actual se ejecute.

El término “continuaciones” también se puede usar para referirse a “continuaciones de primera clase”, las cuales son construcciones que le dan al lenguaje de programación la habilidad de salvar el estado de ejecución en cualquier punto para luego volver a retornar a él en la próxima llamada a la función.

Los programas deben asignar espacio en memoria para almacenar las variables que se usan en sus funciones. Para esto, muchos lenguajes de programación usan una pila porque les permite un rápido y simple mecanismo para la asignación y devolución automática de memoria. La instrucción ***goto*** es la forma más básica de esto. Estructuras de control como: declaraciones con ***if***, ciclos, ***break***, ***continue*** and ***return*** son formas más estructuradas de manipulación del orden de la ejecución de las instrucciones, y son esencialmente declaraciones ***goto*** limitadas.

También existen construcciones más complejas, por ejemplo en C, ***setjmp*** se puede usar para realizar un salto desde cualquier parte de una función hacia cualquier parte de otra función. Otros ejemplos de esto son el uso de corutinas en Simula 67, generadores en Icon y Phyton, fibras en Ruby, entre otros.

1.6 *Bytecode*

El *bytecode* o código de *byte* es un código intermedio más abstracto que el código de máquina. Habitualmente es tratado como un archivo binario que contiene un programa ejecutable similar a un

módulo objeto, que es un archivo binario producido por el compilador cuyo contenido es el código objeto o código máquina. El *bytecode* recibe su nombre porque usualmente cada código de operación tiene una longitud de un *byte* (entre 0 y 255), aunque la longitud del código de las instrucciones varía. Como código intermedio, se trata de una forma de salida utilizada por los diseñadores de lenguajes para reducir la dependencia respecto del hardware específico y lograr una mejor interpretación. Los programas en *bytecode* suelen ser interpretados por un intérprete de *bytecode* (en general llamado máquina virtual, dado que es análogo a un ordenador). Su ventaja es la portabilidad: el mismo código binario puede ser ejecutado en diferentes plataformas y arquitecturas. Es la misma ventaja que presentan los lenguajes interpretados. Sin embargo, como el *bytecode* es en general menos abstracto, más compacto y más orientado a la máquina que un programa pensado para su modificación por humanos, su rendimiento suele ser mejor que el de los lenguajes interpretados. A causa de esa mejora en el rendimiento, muchos lenguajes interpretados se compilan para convertirlos en *bytecode* y después son ejecutados por un intérprete de *bytecode*. Entre esos lenguajes se encuentran Perl, PHP y Python. [4]

1.7 *Opcode*

Un *opcode* (*operation code*) o código de operación es la porción de una instrucción del lenguaje de máquina que especifica la operación a ser realizada. Una instrucción completa del lenguaje de máquina contiene un *opcode* y, opcionalmente, la especificación de uno o más operandos. Algunas operaciones tienen operandos implícitos o de hecho ninguno. Los operandos sobre los cuales los *opcodes* aplican pueden ser valores en memoria, valores en la pila, puertos de entrada/salida, entre otros. Las operaciones que un *opcode* puede especificar pueden incluir aritmética, copia de datos, operaciones lógicas, entre otras. Los *opcodes* también pueden ser encontrados en los *bytecodes*. En estos se crea una arquitectura de conjunto de instrucciones para ser interpretada por software en vez de un dispositivo de hardware. Es por esto que los intérpretes de *bytecode* trabajan con tipos de datos y operaciones de más alto nivel que el de un conjunto de instrucciones por hardware, ejemplo de estas operaciones pueden ser: leer variable del dispositivo, enviar para una base de datos, entre otras.

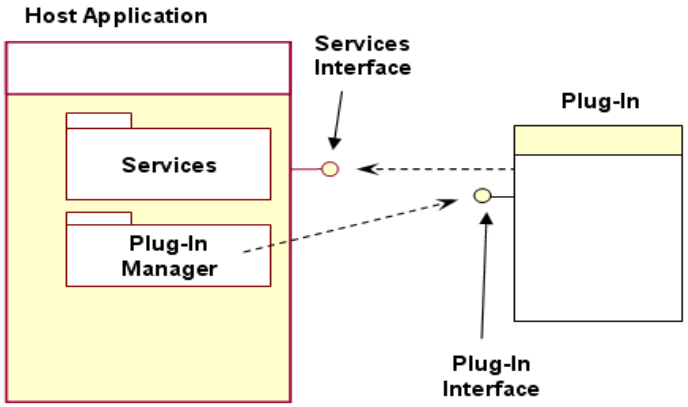
1.8 Máquina Virtual

Una máquina virtual es, en esencia, un intérprete de *bytecode.* La máquina permite ejecutar una tarea expresada mediante un código de *byte*. Para esto se auxilia del repertorio de instrucciones definidas por el programador. En este repertorio se detallan las instrucciones que la máquina virtual puede entender y ejecutar, o lo que es lo mismo, el conjunto de todos los operadores implementados. Las instrucciones que no estén incluidas en este conjunto de operaciones no podrán ser interpretadas por la máquina, por lo que no deben estar incluidas en el código de *byte*. Al proceso de interpretar un

código de operación de los contenidos en todo el código de *byte* de la tarea se le denomina proceso de decodificación. En este proceso es donde se interpreta y ejecuta la operación que se encuentra asociada a un determinado código de operación.

1.9 Desarrollar una Extensión

Una extensión es una aplicación que se relaciona con otra – denominada aplicación hospedera – para aportarle un conjunto de funciones nuevas y generalmente muy específicas. Las extensiones son ejecutadas por la aplicación hospedera y ambas interactúan por medio de una interfaz. Para esto, la aplicación hospedera provee servicios que pueden ser usados por las extensiones. Estos servicios incluyen una forma que permite que las extensiones se registren con la aplicación hospedera y un protocolo para el intercambio de datos. Las extensiones dependen de los servicios provistos por la aplicación hospedera y usualmente no funcionan por ellas mismas. Por otra parte, la aplicación hospedera sí funciona independientemente de las extensiones. Esto permite que se puedan adicionar y actualizar extensiones de forma dinámica sin la necesidad de realizar cambios en la aplicación hospedera. Actualmente las extensiones existen como una forma de extender la funcionalidad de aplicaciones, de manera que se puedan añadir sin afectar las ya existentes ni complicar el desarrollo de la aplicación principal. Las razones por las cuales las aplicaciones soportan extensiones son varias, entre sus principales se encuentran: posibilitar a otros desarrolladores crear funcionalidades nuevas para extender el funcionamiento de la aplicación y reducir el tamaño de la aplicación.



Las extensiones programadas en C++, generalmente consisten en bibliotecas compartidas que son cargadas de forma dinámica por el programa principal. En estas bibliotecas se encuentran un conjunto de funciones que son invocadas por la aplicación hospedera. Cuando se exporten estas funciones debe asegurarse que el nombre con el que se almacene en la biblioteca sea exactamente igual al nombre provisto por la interfaz. Es posible que estas bibliotecas no pueden ser compiladas con un compilador de C debido a que quizás algunas de las funciones de la interfaz hagan uso en sus prototipos de tipos de datos que no son compatibles con C. Por lo tanto es posible que las bibliotecas sólo pueden ser compiladas con compiladores de C++. Las funciones en C++ no necesariamente se

almacenan en la biblioteca con el nombre que les fue dado por el programador de la biblioteca, sino que en ocasiones se les suele agregar al nombre ciertos símbolos extras. Este comportamiento es conocido como decoración de nombre y no es el deseado, dado que si la aplicación hospedera necesita que la biblioteca exporte una función con el nombre iniciarFuncionalidad, la biblioteca tiene que exportar la función con este nombre y no con el nombre ZE4iniciarFuncionalidad7Y, entre otras variantes de nombres inválidos. Para resolver este problema el lenguaje C++ provee la frase **extern** "C". En las plataformas UNIX basta con agregarle esta frase delante de la declaración de cada función que se quiera exportar. Sin embargo en las plataformas de Microsoft, aunque esta frase elimina la decoración de nombre, no garantiza que la función pueda ser cargada de forma dinámica. Es por esto que en las plataformas de Microsoft hay que agregar además __declspec(dllexport). Estas dos variantes deben ser tenidas en cuenta si se desea que el código sea multiplataforma.

1.10 Cargar de forma dinámica una biblioteca compartida

Para que la aplicación principal pueda cargar dinámicamente una biblioteca compartida, debe hacer uso de algunas funciones provistas por los sistemas operativos.

UNIX	Microsoft	Descripción
dlopen	LoadLibrary	Carga en memoria la biblioteca.
dlsym	GetProcAddress	Busca una de las funciones exportadas en la biblioteca dado el nombre de la misma.
dlclose	FreeLibrary	Libera la memoria asignada a la biblioteca.
dlerror	GetLastError	En caso de que ocurra un error en alguna de las funciones anteriores, devuelve dicho error.
dlfcn.h	Winbase.h	Archivo que presenta la definición de las cuatro funciones anteriores.
libdl.so	Kernel32.dll	Biblioteca donde se encuentra el código de las cuatro funciones anteriores.

Estas dos variantes deben ser tenidas en cuenta si se desea que el código sea multiplataforma.

Capítulo 2: Desarrollo de la Solución

2.1 Requisitos

La solución que necesita el polo debe ser capaz de hacer un uso eficiente de los recursos con los que cuenta y posibilitar el desarrollo de las principales actividades básicas (antes mencionadas) que realizan el resto de los proyectos. Para lograr esto es necesario que entre las características fundamentales que brinde la solución, se encuentren:

- Ejecución de tareas de forma periódica.
- Ejecución de tareas orientadas a la generación de ciertos eventos.
- Trabajo con distintas políticas de planificación para que cada aplicación que se valga de esta solución decida cual es la que mejor cumple con sus necesidades.
- Trabajo con aplicaciones que requieran de tiempo real.
- Habilidad de interrumpir las tareas que se estén ejecutando antes de su tiempo de culminación, para en caso de que en un instante de tiempo el planificador decida que la tarea debe ser suspendida, no haya que esperar hasta que la misma termine de ejecutarse, sino que puede ser interrumpida lo antes posible.
- Aumento del repertorio de funcionalidades mediante extensiones.
- Interacción síncrona y asíncrona con las extensiones.
- Portabilidad.

2.2 Definición de Tarea

Una tarea consiste en una terna formada por: datos acerca del momento en el cual la tarea se encuentra lista para su inmediata ejecución, un conjunto de atributos para la planificación, y un código de *byte* que expresa la secuencia de instrucciones que la tarea debe realizar.

2.3 Momento de activación de las Tareas

De acuerdo al momento en el cual las tareas se encuentran listas para el comienzo de su ejecución, las tareas se clasifican en tres tipos: tareas periódicas, tareas aperiódicas y tareas esporádicas. Todas las tareas son generadas por eventos. Sin embargo estos eventos se clasifican en dos grupos: eventos de tiempo y eventos de usuarios, aunque existe un evento especial que no clasifica dentro de ninguno de estos dos grupos, y es un evento que genera el framework al comenzar su ejecución. Las

tareas periódicas y aperiódicas son tareas temporizadas, o sea, tareas que son iniciadas por eventos de tiempo. Su diferencia radica en que las aperiódicas se ejecutan una sola vez, mientras que las periódicas se ejecutan constantemente cada cierto intervalo de tiempo, denominado período. Sin embargo existe una excepción en estas tareas temporizadas en cuanto al momento en que deben ser iniciadas, y es el hecho de que la primera vez que comiencen su ejecución no necesariamente tiene que ser mediante un evento de tiempo sino también mediante el evento especial del framework. Las tareas esporádicas por otra parte sólo se generan por eventos definidos por el usuario y no pueden ser periódicas, o sea, una vez que se ejecuten no se vuelven a ejecutar hasta que no se vuelva a lanzar el evento al cual están asociadas. A un determinado evento, tanto de tiempo como de usuario, o incluso, el evento especial del framework, se le puede asociar más de una tarea, en cuyo caso, al lanzar el evento, el ejecutivo es notificado de que tiene que comenzar la ejecución de todas las tareas que están asociadas a dicho evento. El orden en que las tareas asociadas a un mismo evento son iniciadas depende de las políticas de planificación.

2.4 Atributos de las Tareas

Generalmente, los planificadores necesitan información de las tareas para realizar una correcta planificación de las mismas. A este tipo de información se le denominará atributos de la tarea. Ejemplo de atributos de las tareas pueden ser: el nivel de prioridad, el tiempo de culminación, entre otros. Estos atributos sólo son útiles para el planificador y por lo tanto sólo el planificador puede acceder a los mismos. Cada uno de los atributos puede ser estático o dinámico. La diferencia entre los atributos estáticos y dinámicos radica en que los estáticos son conocidos a priori mientras que los dinámicos sólo pueden determinarse en tiempo de ejecución. La colección de atributos de la tarea puede tener todos los atributos estáticos, todos dinámicos, o algunos estáticos y otros dinámicos. El framework posibilita que todos los atributos, ya sean estáticos o dinámicos, puedan ser modificados por el planificador en tiempo de ejecución.

2.5 Código de las Tareas

El código de la tarea está compuesto por una secuencia lineal de instrucciones a realizar, unas seguidas de las otras. Cada instrucción está compuesta por un operador y de forma opcional por uno o varios operandos. El operador es el responsable de la semántica de la operación, es él quien conoce cuantos operandos requiere y en que orden están situados. También es el operador el encargado de chequear que se cumplan las condiciones indispensables para su procesamiento (condiciones iniciales o precondiciones) y dejar a la tarea que se está ejecutando en un estado consistente para la ejecución de la próxima instrucción. Los operandos, por otra parte, son los representantes de información vital para la realización de las operaciones. El código de la tarea se expresa mediante un código de *byte* que es interpretado por una máquina virtual que el framework

proporciona. Este código de *byte* está compuesto por instrucciones, las cuales a su vez se componen por un código de operación y de forma opcional por una secuencia de *bytes* que representan los datos de la instrucción.

2.6 Definición de Instancia de una Tarea

Como ya se ha visto hasta el momento, el concepto de tarea sólo abarca lo concerniente al momento de activación de la tarea, a los atributos necesarios para su planificación y al conjunto de instrucciones que forman a la misma. Las instrucciones expresan qué debe hacerse y en qué orden. Sin embargo para la ejecución de una tarea es necesario contar con una continuación que permita darle seguimiento a la ejecución de dicha tarea. Es por esto que es necesario contar con un concepto que englobe a los conceptos de tarea y continuación. Este nuevo concepto es el concepto de Instancia de una Tarea. Una vez que el framework es notificado para generar una tarea determinada, lo que realmente crea no es dicha tarea, sino una instancia de la tarea en cuestión. Esto se debe a que el concepto de tarea es un concepto más bien estático, ya que no involucra comportamiento, mientras que el concepto de instancia de una tarea es una extensión del concepto de tarea, que soporta el almacenamiento de los estados por los que la tarea va transitando al pasar de una instrucción hacia la otra, gracias al concepto de continuación. En lo que resta se tratará el concepto de tarea e instancia de una tarea indistintamente.

2.7 La Pila

La pila es una estructura en la cual se almacena la información que es necesaria que persista entre instrucciones del código que se está ejecutando. Las variables locales de cada instrucción no son salvadas y por tanto no ocupan espacio en la pila. Es responsabilidad del desarrollador de las instrucciones decidir que valores se almacenan en la pila y cuales no, y en que orden se colocan. El espacio en memoria utilizado por cada pila es un valor fijo y este valor es el mismo para todas las pilas. Esto permite que todas las pilas puedan ser pasadas por copias, simplificando así el diseño de la aplicación al no tener que tener en cuenta el trabajo con direcciones de memoria (apuntadores de C++), además, cuando se obtiene una pila ya esta viene con toda su información, no con una referencia a la misma, lo que permite que se haga un uso más eficiente de la caché del procesador. Dado el hecho de que el tamaño de la pila sea un valor fijo y este se mantiene constante durante el resto de la aplicación, es necesario que los valores que se decidan almacenar en la pila sean lo más pequeños posibles para lograr que todos los valores que deben ser salvados posean espacio disponible para su inserción. Es por esto que si se desean almacenar en la pila valores que requieran mucho espacio en memoria, entonces debe almacenarse sólo una referencia de los mismos y no su valor en sí.

2.8 La Continuación

La continuación es una estructura que nos permite almacenar el estado en que se encuentra la tarea en cada momento. Su principal objetivo es lograr que cuando una tarea determinada es interrumpida, entonces una vez que vuelva a ser invocada para completar su ejecución, no comience por la primera instrucción, sino por la instrucción que sigue a la última que ejecutó. Para lograr este objetivo, la continuación se vale de un contador que guarda el valor de la instrucción que se está ejecutando, y es responsabilidad de los desarrolladores de instrucciones que antes de concluir el código de su instrucción, establezcan este contador al valor de la próxima instrucción. Con sólo almacenar el contador a instrucciones no es suficiente para lograr este efecto, ya que las informaciones entre instrucciones deben ser persistentes. Es por esto que la continuación de una tarea también se vale de una pila. Con esta combinación entre pila y contador de instrucciones es que se logra un control total del estado de las tareas: instrucción actual que se está ejecutando e información acumulada en la tarea hasta el momento.

2.9 Las Instrucciones

Como ya se ha visto, las tareas tienen asociadas un conjunto de instrucciones, una seguida de la otra, que son las que definen su comportamiento. Las instrucciones constituyen el átomo de las tareas, o sea, las tareas sólo pueden ser interrumpidas entre instrucciones, nunca en medio de una instrucción. Los datos de las instrucciones son establecidos antes de la ejecución de la aplicación, por lo que tienen que ser conocidos a priori por el programador de la tarea a ejecutar. Durante la ejecución de la tarea no se pueden modificar estos valores, ya que son sólo disponibles para su lectura y no para su escritura. Si cierta información no puede ser conocida con anterioridad, sino que su valor depende de cálculos o de otras informaciones que se generan en tiempo de ejecución, entonces este tipo de información no puede ser almacenada en los operandos de las instrucciones, sino en la pila asociada a la tarea.

2.10 Tareas generadas por eventos de usuarios

Las tareas que no son temporizadas, o sea, las que se generan por eventos definidos por los usuarios, pueden estar asociadas a uno o varios eventos. Una vez que se genere uno de los eventos a los cuales la tarea se asocie, el framework es notificado de comenzar la ejecución de la misma. Esto no significa que la tarea se comience a ejecutar inmediatamente, sino que ya está lista para su inmediata ejecución. No importa en qué orden se asociaron las tareas al evento, ya que siempre se ejecutarán de acuerdo a las políticas de planificación que implemente el planificador. Estos tipos de tareas generalmente necesitan para su ejecución información de estado de otras tareas, especialmente de la tarea que lanzó el evento. Por lo tanto, es necesario que la tarea que genere el

evento pueda transmitirle información a las tareas que se crean por ese evento. Para resolver esta necesidad el ejecutivo cuenta con un concepto de **Evento** que posee un identificador y una pila. El identificador es el valor que define de forma única al evento. Asociar una tarea a un evento significa asociarla a un identificador de evento, y para generar un evento, también tiene que tenerse en cuenta este valor. La pila, por otra parte, es la encargada de almacenar la información necesaria para las tareas que se van a generar. Estas nuevas tareas pueden comenzar su ejecución con información en la pila, a diferencia de las temporizadas que siempre comienzan con la pila vacía.

2.11 Contexto de Ejecución

El contexto de ejecución es un recurso provisto por el ejecutivo para la ejecución e interrupción de las tareas. El mismo usa la máquina virtual del ejecutivo para la interpretación de las tareas. El contexto de ejecución funciona sólo a petición del planificador, o sea, el planificador le asigna una tarea para que el contexto la ejecute, y la tarea sólo puede ser interrumpida a petición del planificador. Cuando el planificador le ordena detener una tarea, esta no se detiene inmediatamente, a menos que ya haya concluido con la ejecución de una instrucción, sino que el contexto esperará hasta que concluya la instrucción que se está ejecutando para hacer la interrupción. Esto ocurre debido al carácter atómico de las instrucciones de las tareas. Cuando la ejecución de la tarea es interrumpida o simplemente finalizada debido a que ya no le queda más código por ejecutar, el contexto le notifica al planificador que está libre para que este le pueda asignar una nueva tarea. En caso de que el planificador no posea tarea que asignarle, entonces el contexto permanece en estado de reposo hasta que vuelva a aparecer otra tarea. La función invocada por el planificador para ordenar la interrupción de la tarea es asíncrona, o sea, no espera hasta que el contexto la detenga, sino que sólo realiza la orden. El ejecutivo puede presentar más de un contexto de ejecución para hacer un uso más eficiente de los recursos de hardware, por lo que el planificador debe ser capaz de trabajar con varios contextos de ejecución simultáneamente.

2.12 Instrucciones Síncronas y Asíncronas

Debido a que algunas operaciones no son deterministas, ya que dependen de eventos externos que son impredecibles, es necesario para no bloquear el contexto de ejecución y hacer un uso eficiente del mismo, que dichas operaciones se ejecuten de forma asíncrona. Asíncrona significa que la tarea se suspende y se copia hacia otro lugar que va a ser el encargado de la ejecución de la operación asíncrona. Una vez que la tarea es suspendida, el contexto de ejecución se declara como libre de tareas para que el planificador le asigne una de las tareas que están listas para ejecutar. Por otro lado, cuando la operación asíncrona concluya, entonces la tarea es asignada al planificador para que este determine cuando ejecutarla nuevamente. Cuando esta tarea vuelva a ejecutarse no lo hace desde el principio de su código, sino desde el último punto donde se quedó, o sea, la ejecución

comenzaría en la instrucción que le sigue a la instrucción asíncrona por la que tuvo que ser suspendida. El resto de las instrucciones que no necesitan ser interrumpidas, ya que demoran un período de tiempo calculable (las que son deterministas) se denominan instrucciones síncronas. Las instrucciones síncronas no se suspenden por sí mismas, sólo pueden ser suspendidas a petición del planificador. Cuando el planificador solicita que una tarea se suspenda y dicha tarea está ejecutando una instrucción síncrona, esta no pasa a otro sitio para completar su ejecución, sino que termina su procesamiento en el contexto de ejecución y posteriormente la tarea se suspende. En este último caso es responsabilidad del planificador copiar la tarea que mandó a suspender antes de asignarle una nueva tarea al contexto, ya que de no hacerse esto, el resto de la tarea nunca se ejecutará.

2.13 Instrucciones del Núcleo

El núcleo del ejecutivo proporciona un conjunto muy limitado de instrucciones básicas para el procesamiento de las tareas. Estas instrucciones se corresponden con las instrucciones primitivas de los lenguajes de programación de alto nivel: sentencias condicionales si-entonces, ciclos, saltos incondicionales, entre otras. Estas instrucciones son esenciales para la programación de las tareas.

2.14 Módulos Externos

A pesar del número de instrucciones básicas provistas por el núcleo, se pueden formar nuevas instrucciones más complejas. Estas nuevas instrucciones le son provistas al ejecutivo por medios de extensiones. Esto trae como ventajas que las tareas puedan programarse a más alto nivel, ya que una de estas instrucciones puede tener la misma funcionalidad que una combinación más extensa de instrucciones del núcleo. Eso en el caso de que con las instrucciones del núcleo también se pudiera hacer lo mismo, porque es posible que no se pueda, y he aquí otra de las ventajas de permitir extensiones. Por otra parte, estas extensiones también tienen desventajas y es que una tarea expresada con instrucciones básicas lleva un mayor número de instrucciones por lo que en caso de que se solicite su interrupción por parte del planificador, esta se puede detener cuando termine una de las instrucciones primitivas que está ejecutando, mientras que si todas estas instrucciones estuvieran representadas por una sola instrucción, habría que esperar a que esta instrucción más compleja termine, lo que sería análogo a esperar por el cumplimiento de todas las instrucciones básicas.

En el código de estas instrucciones puede usarse información que no radica en la pila de la tarea, ya que hay ciertos datos que son comunes para todas las tareas y en caso de almacenarse esta información en la tarea se estaría desperdiciando memoria. Para resolver este problema las instrucciones que requieren de este tipo de información, almacenan un identificador de la misma en sus operandos. A estos contenedores de información adicional que sólo radican en la extensión, son

denominados por el framework como módulos externos. En una misma extensión pueden existir tantos módulos como se necesite, incluso, puede que no exista ninguno.

Capítulo 3: Desarrollo de Extensiones

3.1 Las extensiones

Al ejecutivo se le pueden incorporar tres tipos de extensiones: extensiones para el paralelismo, extensiones para la planificación y extensiones para las instrucciones.

De las extensiones para la programación en paralelo sólo es cargada por el ejecutivo una sola. De modo que pueden existir varias pero sólo una es usada. Esto se especifica en un archivo de configuración. Este tipo de extensión se usa para crear hilos en los contextos de ejecución, en los objetos activos y en el generador de tareas programadas. Además, en los contextos de ejecución y en los objetos activos se usan mecanismos de sincronización. Ejemplo de mecanismos de sincronización son el trabajo con las variables de condiciones, los semáforos, los *mutexes*, etc. El ejecutivo sólo usa los más simples: bloquear y desbloquear un *mutex*. Para que el ejecutivo pueda realizar tareas de tiempo real, esta extensión debe proveerle hilos y mecanismos de sincronización de tiempo real. De ahí que estas funcionalidades sean cargadas mediante extensiones y no puestas directamente en el código del ejecutivo. El ejecutivo provee una extensión para el paralelismo que usa la biblioteca *boost*, por lo que estas funcionalidades no son de tiempo real, ni siquiera se le puede establecer una prioridad a los hilos de ejecución. En caso de no ser deseada esta extensión, se le puede incorporar otra, usando otra biblioteca como: Qt, OpenThreads, etc, o incluso, en vez de usar una biblioteca se puede usar las API del sistema operativo. En caso de que se provea una nueva extensión sólo habrá que cambiar el archivo de configuración para notificarle al ejecutivo cual es la extensión para el paralelismo que debe cargar, y no compilar nuevamente el ejecutivo.

De forma análoga ocurre con la extensión para la planificación. Aunque pueden existir varias extensiones de este tipo, sólo una es cargada. De igual forma para determinar cual es cargada es necesario establecer en el archivo de configuración del ejecutivo cual es el planificador a cargar. El ejecutivo provee una extensión que planifica según la filosofía: primera tarea que se encuentra lista, primera tarea que se ejecuta. Este planificador nunca interrumpe la ejecución de una tarea, sino que una vez que decide asignársela al contexto de ejecución para su procesamiento, espera a que el contexto le notifique que se encuentra libre para asignarle una nueva tarea. Este planificador además es muy sencillo, ya que no necesita que las tareas presenten atributos para su planificación. Aunque estas políticas de planificación pueden ser muy usadas en un gran número de aplicaciones, puede no ser deseadas en otras, de hecho, esto es inconcebible para la ejecución de tareas de tiempo real. Para resolver este problema, se le puede proveer otra extensión al ejecutivo que implemente las políticas de planificación deseadas y configurarlo para que use el nuevo planificador en vez de el

provisto por él.

De las extensiones para las instrucciones, por el contrario, pueden cargarse todas las que necesite el ejecutivo.

Para desarrollar una nueva extensión que pueda ser compilada en varias plataformas hay que tener en cuenta la decoración de nombre. Para resolver este problema el framework provee una macro en su paquete utilitario que permite exportar la función correctamente para que el ejecutivo pueda cargarla. Esta macro es RTE_EXPORT_FUNCTION y se encuentra en el archivo ParallelManager.hpp de la carpeta Utils.

3.2 ¿Cómo desarrollar una extensión para el paralelismo?

Para desarrollar una nueva extensión que pueda ser usada por el ejecutivo para sus técnicas de programación en paralelo, es necesario desarrollar una biblioteca compartida que exporte la siguiente función:

```
#include <Parallel/ParallelManager.hpp>
#include <Utils/ExportFunction.hpp>

RTE_EXPORT_FUNCTION
void initParallel(RTE::ParallelManager & parallelManager)
{
   //aquí es donde se programa la iniciación de esta extensión.
}
```

La clase ParallelManager sólo provee apuntadores a funciones que deben ser establecidos correctamente en la función initParallel. Estos apuntadores son los siguientes:

Apuntador	Descripción
createThread	Crea un hilo con determinada prioridad y lo pone a ejecutar una rutina dada.
destroyThread	Destruye un hilo dado previamente creado con la función createThread.
createMutex	Crea un mutex.
DestroyMutex	Destruye un mutex dado previamente creado con la función createMutex.
LockMutex	Bloquea un mutex dado previamente creado con la función createMutex.
unlockMutex	Desloquea un mutex dado previamente creado con la función createMutex.

Para ilustrar con un ejemplo, vamos a establecerle sólo dos apuntadores: createMutex y lockMutex.

```
#include <Parallel/ParallelManager.hpp>
#include <Utils/ExportFunction.hpp>
#include <boost/thread/mutex.hpp>

RTE::Mutex createMutex()
{
```

```
    return new boost::mutex;
}

void lockMutex(RTE::Mutex mutex)
{
    static_cast<boost::mutex *>(mutex)->lock();
}

RTE_EXPORT_FUNCTION
void initParallel(RTE::ParallelManager & parallelManager)
{
    parallelManager.createMutex = createMutex;
    parallelManager.lockMutex = lockMutex;
}
```

Aunque en el ejemplo sólo se establecieron dos apuntadores para no hacerlo tan extenso, es necesario para el funcionamiento del ejecutivo que todos los apuntadores sean correctamente establecidos.

3.3 ¿Cómo desarrollar una extensión para la planificación?

Para desarrollar una nueva extensión que pueda ser usada por el ejecutivo para sus políticas de planificación, es necesario desarrollar una biblioteca compartida que exporte la siguiente función:

```
#include <Scheduler/Scheduler.hpp>
#include <Scheduler/ExecutionContextManager.hpp>
#include <Scheduler/SchedulerOptionalAttributes.hpp>
#include <Utils/ExportFunction.hpp>

RTE_EXPORT_FUNCTION
void initScheduler(RTE::Scheduler & scheduler,
                   const RTE::ExecutionContextManager & ecManager,
                   unsigned int contextNumber,
                   const RTE::SchedulerOptionalAttributes & schOpAttrs))
{
    //aquí es donde se programa la iniciación de esta extensión.
}
```

Al igual que la clase ParallelManager del epígrafe anterior, la clase Scheduler sólo provee apuntadores a funciones que deben ser establecidos correctamente en la función initScheduler.

Apuntador	Descripción
configure	Permite la configuración del planificador.
getTaskScheduling	Determina cuales son los atributos de las tareas y crea una estructura para almacenarlos. Por último, devuelve su dirección de memoria.
start	Pone al planificador en un estado listo para procesar las tareas.
finish	Pone al planificador en un estado incapaz de seguir procesando tareas.

initTask	Establece los atributos de la tarea y les asigna su valor inicial.
endTask	Libera los atributos establecidos en la función initTask.
addTask	Adiciona una nueva tarea al planificador.
schedule	Planifica las tareas que han sido adicionadas.
getAsyncTaskCollection	Devuelve una colección para almacenar las tareas asíncronas.
insertAsyncTask	Inserta en la colección provista por la función getAsyncTaskCollection una nueva tarea asíncrona.
getAndEraseAsyncTask	Obtiene la tarea asíncrona más prioritaria de la colección provista por la función getAsyncTaskCollection. Por último, la elimina.
onIdle	Función invocada por los contextos de ejecución una vez que estén libres de tareas.

La clase ExecutionContextManager provee al planificador las siguiente funcionalidades:

Función	Descripción
createExecutionContext	Crea un contexto de ejecución listo para procesar tareas.
finish	Finaliza el procesamiento en el contexto de ejecución.
suspend	Ordena la suspensión de la tarea que se está procesando en el contexto.
resume	Ordena al contexto que ya tiene tarea asignada nuevamente y que debe comenzar a ejecutarla.
getTaskInstance	Devuelve una referencia de la tarea asignada al contexto de ejecución.

El atributo contextNumber le indica al planificador con cuantos contextos de ejecución debe trabajar. Esta indicación le es provista al ejecutivo mediante su archivo de configuración, y debe estar en correspondencia con el número de procesadores físicos del sistema. El planificador no está obligado a usar este atributo para crear los contextos de ejecución, esto es sólo una sugerencia.

La clase SchedulerOptionalAttributes como el nombre lo indica, presenta atributos opcionales que le pueden ser útil al planificador. En esta versión solamente le es útil los apuntadores relacionados con la programación en paralelo. Apuntadores que debieron haber sido establecidos previamente por la extensión del epígrafe anterior.

3.4 ¿Cómo desarrollar una extensión para las instrucciones?

Para desarrollar una nueva extensión que pueda ser usada por el ejecutivo para aumentar su repertorio de instrucciones, es necesario desarrollar una biblioteca compartida que exporte la siguiente función:

```
#include <Modules/DecodingTable.hpp>
#include <Modules/ModuleManager.hpp>
```

```
#include <Modules/ModuleOptionalAttributes.hpp>
#include <Utils/ExportFunction.hpp>

RTE_EXPORT_FUNCTION
void initModuleBegin(RTE::DecodingTable & decTable,
                     RTE::ModuleManager & modManager,
                     const RTE::ModuleOptionalAttributes & modOptAttrs)
{
   //aquí es donde se programa la iniciación de esta extensión.
}
```

En el primer argumento (decTable), se almacenan todas las instrucciones que se han implementado en el núcleo del ejecutivo, más las que ya han sido cargadas de otras extensiones de este tipo. A esta variable se le debe adicionar todas las instrucciones que son propias de la extensión que estamos desarrollando. Para adicionar una nueva instrucción, la clase DecodingTable cuenta con la función addOperator(), la cual recibe como parámetro el operador que se desea incluir. Por lo tanto, esta función debe invocarse tantas veces como operadores se quieran agregar al repertorio de instrucciones.

La clase ModuleManager presenta los dos apuntadores siguientes:

Apuntador	Descripción
createModule	Crea una instancia de un módulo y devuelve su identificador.
configureModule	Configura la instancia creada por la función createModule dado su identificador.

El apuntador configureModule no es obligatorio que sea establecido, pues no todos los módulos que se creen tienen que presentar configuración de archivo. Por otra parte, el apuntador createModule también puede ser opcional, pues puede que no sea necesario que que la extensión presente módulos.

La clase ModuleOptionalAttributes provee tres funciones:

Funciones	Descripción
getParallelManager()	Utilidad provista por el ejecutivo para en caso de que la extensión necesite usar técnicas de programación en paralelo.
getActiveObjectManager()	Utilidad provista por el ejecutivo para en caso de que alguna de las instrucciones de la extensión sea de forma asíncrona.
getSignalEvent()	Utilidad provista por el ejecutivo para en caso de que alguna de las instrucciones de la extensión genere eventos para la creación de nuevas tareas.

Hasta el momento ya se han visto todas los parámetros de la función initModuleBegin(), pero aún no

se ha expresado como crear una nueva instrucción. Para esto es necesario crear una función con el siguiente prototipo:

```
#include <Task/TaskInstance.hpp>

void miOperador(RTE::TaskInstance & task)
{
    //aquí es donde se programa el código de esta instrucción.
}
```

Capítulo 4: Demostrativo

Para probar el funcionamiento del sistema ejecutivo se realizó un demostrativo para uno de los proyectos del polo, el proyecto Zunzún. En el mismo era necesario la lectura de variables de dispositivos instalados en algunas oficinas de ETECSA y visualizar los valores leídos mediante la Web. Para la visualización era necesario además: generar algunos valores que no se encontraban en el dispositivo, sino que su valor se calculaba a partir de otros valores ya leídos, y almacenar ciertos valores leídos en una base de datos para realizar gráficos de tendencias con los mismos. De lo anteriormente expuesto, el sistema ejecutivo se encargó de la recolección de las variables de los dispositivos, de su envío hacia el módulo de visualización Web, de la generación de nuevas variables a partir de las ya existentes, y de la persistencia de las variables seleccionadas hacia un gestor de base de datos.

Para la realización del mecanismo de recolección de las variables se usaron tres de los manejadores implementados en el polo: Modbus RTU, SNMP y UP1. Para interactuar con estos manejadores se desarrolló una extensión para el ejecutivo, denominada GDI (Generic Drivers Interface, Interfaz Genérica de Manejadores en español). Esta extensión le proveía nuevas funcionalidades como: leer variable numérica, leer variable de tipo arreglo, entre otras. Además, esta extensión permitía ser configurada mediante un archivo en el cual se le establecían los datos de los manejadores a utilizar, los datos de los dispositivos que se presentaban en las oficinas, y los datos de las variables de los dispositivos que eran relevante para el demostrativo. Aunque esta extensión se probó con estos tres manejadores solamente, la misma fue elaborada para trabajar con varios manejadores de cualquier tipo. La única restricción que deben presentar los manejadores para que la extensión funcione correctamente es que cumplan con la Interfaz Genérica de Manejadores del Polo de Hardware y Automática. Por lo tanto, esta extensión puede ser reutilizada para otros proyectos que requieran la recolección de las variables de los dispositivos.

Una vez que se leyeran las variables había que enviarlas al módulo Web mediante un middleware y solamente unas pocas almacenarlas en una base de datos. Para el desarrollo de este demostrativo se decidió usar RIPC como middleware y PostgresSQL como gestor de base de datos.

Para el envío de variables mediante el middleware se desarrolló otra extensión al ejecutivo, denominada RIPC (por ser la tecnología de middleware a utilizar). Esta extensión le proveía funciones como: enviar variable de tipo entero y enviar variable de tipo real. Además, esta extensión también permitía ser configurada mediante un archivo en el cual se le establecía la dirección ip y el puerto en

el cual se encontraba el módulo de visualización Web. Esta extensión puede ser reutilizada en otros proyectos siempre que la tecnología de middleware a utilizar sea RIPC. En caso de ser deseado utilizar otra tecnología como TAO o ICE habría que desarrollar otra extensión que use estos middlewares.

Para el almacenamiento en la base de datos de las variables seleccionadas, se desarrolló una tercera extensión al ejecutivo, denominada PostgresSQL (por ser el gestor de base de datos a utilizar). Esta extensión le proveía funciones como: almacenar variable de tipo entero y almacenar variable de tipo real. Además, esta extensión también permitía ser configurada mediante un archivo en el cual se le establecía la dirección ip y el puerto en el cual se encontraba el gestor de base de datos, el nombre de la base de datos a utilizar, y el usuario y la contraseña para la autenticación. Esta extensión puede ser reutilizada en otros proyectos siempre que la tecnología de base de datos a utilizar sea PostgresSQL. En caso de ser deseado utilizar otra tecnología como BerkeleyDB u Oracle habría que desarrollar otra extensión que use estos gestores de base de datos.

Por último, se desarrolló una cuarta extensión, denominada Zunzún (por ser el nombre del proyecto en el que se usa). En esta extensión se desarrollaron funcionalidades muy específicas del proyecto Zunzún como: convertir a horas, convertir a minutos, convertir a segundos, generar variable Inversor, generar variable Rectificador, entre otras. A diferencia de las otras tres extensiones, esta no requería ser configurada mediante un archivo. Esta extensión es muy poco probable que sea reutilizada por otras aplicaciones, ya que las instrucciones que le aporta al ejecutivo son particulares del proyecto Zunzún.

Combinando estas cuatro extensiones se logró el propósito trazado: realizar la lectura de las variables, enviarlas por el middleware hacia el módulo de visualización Web, generar las nuevas variables y salvar las seleccionadas en un gestor de base de datos.

Para la realización de este demostrativo no se desarrollaron nuevas extensiones para el paralelismo ni para la planificación, sino que se usaron las presentadas por el ejecutivo: paralelismo usando la biblioteca boost, y planificación FIFO (First In First Out, primera tarea que se le adiciona, primera tarea que se procesa). Ambos sin requerimientos de tiempo real ya que la aplicación no lo necesitaba.

Conclusiones

El ejecutivo desarrollado posibilita la ampliación de sus funcionalidades mediante extensiones, lo cual fue corroborado mediante la integración de varias extensiones del subsistema de adquisición: recolección de variables de dispositivos del campo, envío de variables mediante el middleware, persistencia de variables en base de datos y en archivos. Se logró acotar el tiempo de cómputo de las operaciones deterministas a un valor calculable.

El ejecutivo hace un uso eficiente de los recursos del sistema mediante las técnicas de programación en paralelo y de asignación de memoria. Constituye una aplicación portable y con pocas dependencias de terceros debido a la utilización de tecnologías estándares en su implementación.

Glosario de Términos

computadora de tiempo real: computadora en la que las operaciones que se realizan se encuentran sujetas a restricciones de tiempo.

sensor: aparato eléctrico capaz de transformar magnitudes físicas o químicas, llamadas variables de instrumentación, en magnitudes eléctricas.

micro controlador: circuito integrado o chip que incluye en su interior las tres unidades funcionales de una computadora: CPU, memoria y unidades de entrada/salida.

sistema distribuido: conjunto de computadores separados físicamente y conectados entre sí por una red de comunicaciones distribuida.

semáforo: tipo de dato que constituye el método clásico para garantizar el acceso restringido a los recursos compartidos.

dependencias compartidas: recursos que son necesitados por varias entidades simultáneamente.

rendimiento de una aplicación: proporción entre los recursos utilizados para la realización de una aplicación y el comportamiento de la misma (en cuanto a velocidad, consumo de memoria, etc).

pila: estructura de datos que almacena información acerca de las subrutinas activas de un programa.

código de máquina: representación del código al más bajo nivel, solamente entendido por la computadora.

framework: estructura de soporte mediante la cual otro proyecto de software puede ser organizado y desarrollado (pueden incluir programas, bibliotecas, etc).

instrucción: operación formada por un operador y de forma opcional por un conjunto de operandos.

micronúcleo: núcleo de un sistema operativo que provee un conjunto de primitivas o llamadas al sistema mínimas, para implementar servicios básicos como: espacios de direcciones, comunicación entre procesos y planificación básica.

núcleo monolítico: arquitectura donde todo el sistema operativo es ejecutado en modo privilegiado. Implementa servicios del sistema operativo como: concurrencia, administración de procesos y memoria, entre otros.

subsistema de adquisición: sistema formado por el conjunto de operaciones relacionadas a la lectura, escritura y procesamiento de las variables de los dispositivos.

Referencias bibliográficas

[1] “Real-time systems design and analysis”. Third Edition. Phillip A. Laplante.

[2] “Executive system concept for processing geological data”. T.A. Jones, R.A. Baker and W.H. Dumay

[3] “Redesign of the CSP execution engine”. University of Twente, Bart Veldhuijzen.

[4] “The Insider's Guide to the NXP LPC2300/2400 Based Microcontrollers”. An Ingeneer's introduction to the LPC2300 & LPC2400 Series.

Bibliografía

“Technical Report No. 2005-499 Scheduling Algorithms for Real-Time Systems” Arezou Mohammadi and Selim G. Akl

“An Object Behavioral Pattern for Concurrent Programming” R. Greg Lavender y Douglas C. Schmidt.

“Calling conventions for different C++ compilers and operating systems”. By Agner Fog. Copenhagen University College of Engineering. Capítulo 8

“Composing high-performance memory allocators” Emery D. Berger, Benjamin G. Zorn, Kathryn S. McKinley

“Reconsidering Custom Memory Allocation” Emery D. Berger, Benjamin G. Zorn, Kathryn S. McKinley

http://msdn.microsoft.com/en-us/library/a90k134d.aspx

http://kb.iu.edu/data/ahdi.html

http://www.artima.com/underthehood/bytecode.html

http://www.linfo.org/opcode.html

http://www.allapplabs.com/java/java_virtual_machine.htm