Despachador de Tareas de Tiempo Real por Eventos Temporizados

Mg. Omar Alimenti – Sr. José C. Mosquera
D^to de Ing. Eléctrica - Universidad Nacional de Sur
Avda. Alem 1253 – (8000) Bahía Blanca
Contacto: iealimen@criba.edu.ar

Resumen

El desarrollo de despachadores presenta una serie de inconvenientes a la hora de implementarlos (conmutación de tareas, manejo de pila, etc), sin embargo el cálculo del segmento de tiempo asignado a la ejecución de una tarea (ranura) es muy importante cuando se desea lograr una buena performance. En [1], [2], [6] y [8] se demostró que el rendimiento de Sistemas de Tiempo Real (STR) funcionando en un esquema de Prioridades Fijas (PF) ordenados por períodos monotónicos crecientes (PMC), es muy sensible al tiempo de duración de la ranura elegida.

El presente trabajo describe un despachador PMC, cuyas ranuras, disparadas por un timer, no son fijas, sino que se adecuan a la próxima tarea a ser despachada, logrando un rendimiento óptimo.

Dicho esquema básicamente reduce el "overhead" y evita todo tipo de bloqueo inherente en los sistemas de ranuras fijas, pues se comporta prácticamente como un esquema de despacho de tareas manejado por eventos.

Palabras Claves: Despachador, Tiempo Real, Vencimiento, Ranura, Eventos Temporizados, Conmutación, Timer y Prioridades Fijas.

1. Introducción

Un Sistema de Tiempo Real (STR) es un sistema que debe trabajar correctamente bajo restricciones de tiempo, es decir que no sólo debe proveer resultados aritmético-lógicos correctos, sino que debe realizarlo antes de un cierto tiempo llamado vencimiento. Típicamente un STR está formado por un *subsistema de control* (computadora) y por un *subsistema controlado* (ambiente físico). Las interacciones entre ambos son las operaciones de *muestreo, procesamiento y respuesta*, que deben realizarse dentro de tiempos específicos.

Un gran número de aplicaciones se pueden resolver realizando un único programa monolítico corriendo sobre la plataforma de un determinado procesador. Sin embargo, cuando las aplicaciones crecen, agregando tiempos críticos de respuesta, sirviendo a numerosos dispositivos externos al mismo tiempo, etc., es más atractivo un sistema multitarea donde al procesador se le da el manejo de numerosas tareas que parecen simultáneas.

Un *Sistema Operativo de Tiempo Real* (RTOS) ayuda a la formalización de las relaciones entre las diferentes actividades, simplificando el desarrollo del software. Este esquema hace posible escribir separadamente cada una de las tareas de la aplicación [14]. La división de un proyecto en piezas funcionales (tareas) permite el desarrollo de un software modular y simplifica el manejo de demandas (señales de entradas-salidas) de tiempo real y las comunicaciones entre tareas [15].

Estos beneficios no son “gratis” ya que la implementación multitarea tiene un costo importante que es necesario minimizar: *tiempo de CPU*. La interrupción periódica (ranura) para actualizar tablas, determinar la próxima tarea a ejecutar, el tiempo de respuesta a la interrupción, etc, impactan directamente sobre el tiempo de uso de la CPU, restándole tiempo de ejecución a las tareas. En este punto surgen algunos inconvenientes:

- Cómo reducir la complejidad extra de particionar las tareas y comunicar los procesos entre sí, para asegurar los temporizados y requerimientos de confiabilidad.
- Cómo depurar [3] y monitorear [11] un sistema que está expuesto a errores de temporizado producido por los costos fijos (“overhead”) que surge del proceso de división de tareas.

El despachador de tareas es la parte del sistema operativo que realiza la conmutación de tareas, asignando el uso de la CPU de acuerdo a algún criterio de selección. Para ello cada tarea tendrá una prioridad y en caso de requerimientos simultáneos, el despachador brindará el uso del recurso (CPU), de acuerdo a una *disciplina de prioridades [10]*. Estas pueden ser fijas o variables.

Desde el punto de vista de una implementación tradicional, un despachador de tareas se puede clasificar en dos grandes categorías:

- Manejadas por Eventos
- Manejadas por Temporizador “Timer”.

En las implementaciones manejadas por *eventos*, los dispositivos externos generan interrupciones que marcan los períodos de generación de las tareas. En las manejadas por *temporizador*, se emplea un timer programable conectado a una interrupción, que el despachador utiliza para evaluar los períodos de las tareas y decidir en que momento invocarlas. En esta última, el tiempo se considera ranurado y su duración se denomina *Tiempo de Ranura*. Este es indivisible y no es posible efectuar relevos de tareas.

Un factor importante a tener en cuenta en la implementación de un despachador es la elección del Tiempo de Ranura para un determinado conjunto de tareas. En [1] y [8] se presentan diversas metodologías para incorporar el costo de la diagramación dentro del

contexto de sistemas de tiempo real con prioridades fijas ordenado por PMC. Allí se define el concepto de "ranura óptima" a aquella *que produce el mayor tiempo sobrante para un determinado conjunto de tareas*. Si bien estas alternativas mejoran la performance de un STR, atenuando los problemas clásicos de "overhead" y bloqueo, no logran eliminarlos.

La alternativa de contar con un despachador de tareas que maneje la duración de la ranura en función de los parámetros de la próxima tarea a ser despachada, logra un rendimiento óptimo cercano al producido por la implementación "mediante eventos" (controlador de interrupciones), ya que elimina completamente los bloqueos y reduce considerablemente el "overhead" producido por los sistemas de ranura fija.

2. Análisis de Sistemas de Prioridades Fijas

La teoría de diagramación parece ser un medio que, a priori, permitiría validar los correctos temporizados de las aplicaciones de un STR. Sin embargo existe un gran salto entre la teoría de diagramación y su implementación práctica sobre el "kernel" de un sistema operativo ejecutándose sobre un hardware específico.

Los costos de diagramación para un sistema basado en prioridades fijas son:

- "*Overhead*": (*costos fijos*) es el tiempo que le insume al "kernel" realizar un servicio para una tarea, como invocarla o terminarla.
- *Bloqueo*: también llamado inversión de prioridades, es el tiempo insumido en el "kernel" o en una tarea, que impide que otra tarea de mayor prioridad pueda ejecutarse.

El bloqueo degrada la diagramabilidad de un conjunto de tareas de TR (testear si un conjunto de tareas es diagramable o si es capaz de cumplir o no con los vencimientos). Entendiendo los costos de bloqueo y "overhead" aplicados a una dada implementación, es posible extender las ecuaciones de diagramabilidad de prioridades fijas, para proveer límites más reales de la diagramabilidad de un sistema, que los que presenta la teoría. Esto permitirá realizar la ingeniería de diseño de un diagramador, minimizando dichos costos.

En [5], [9] y [12] se plantean las restricciones para el análisis de diagramadores basados en prioridades fijas:

- Los pedidos de todas las tareas son periódicos con el peor caso de carga. Los tiempos de procesamiento son conocidos. Las tareas están listas al comienzo del período.
- Cada tarea debe ser ejecutada totalmente antes del próximo pedido.
- Las tareas son independientes, apropiativas y no se sincronizan ni bloquean entre sí.
- El costo de la apropiación, incluyendo manejo de interrupciones, diagramación, cambio de contexto, etc. se considera despreciable.

En 1973, Liu and Layland [5], dieron el primer tratamiento teórico sobre diagramación con Prioridades Fijas aplicado a tareas periódicas e independientes e introdujeron el concepto de asignación de prioridades por Períodos Monotónicos Crecientes (PMC) (en inglés, *Rate Monotonic Scheduling- RMS*). El mismo consiste en asignar la mayor prioridad a la tarea que posea el menor período de pedido.

La importancia de la asignación por PMC se basa en que es óptima entre las disciplinas de prioridades fijas, ya que "*si existe un ordenamiento de tareas por prioridades fijas que haga diagramable un sistema, también lo será por PMC*". Estos autores demostraron además que dado un conjunto de m tareas existe un cota suficiente que garantice la diagramabilidad del sistema, tomando como peor caso de carga, la generación simultánea de todos los pedidos. La principal ventaja de esta cota es la facilidad de cálculo, pero lamentablemente es muy pesimista en la determinación de la diagramabilidad de un STR, pues existen muchos STR´s

que siendo diagramables, no cumplen con dicha cota. La condición es suficiente aunque no necesaria.

Lehoczky, Sha y Ding [9] (y otros autores [12] y [13]) desarrollaron condiciones necesarias y suficientes para determinar la diagramabilidad de un STR por PMC:

- El mayor tiempo de respuesta para una tarea τ_i, ocurre en el instante crítico, que se da cuando una tarea τ_i comienza su ejecución C_i unidades de tiempo antes que finalice el período T_i.
- Se alcanzarán todos los vencimientos de las tareas, usando el algoritmo de prioridades fijas, si el primer pedido de cada tarea lo alcanza dentro del instante crítico.
- Un conjunto de n tareas periódicas τ_1, τ_2, …., τ_n es diagramable si se verifica la siguiente ecuación:

$$\forall i, 1 \le i \le n, \min_{0 < t < D_i} \sum_{j=1}^{i} \frac{C_j}{t} \left\lceil \frac{t}{T_j} \right\rceil \le 1 \qquad (1)$$

Notar que (1) evalúa cada tarea τ_i sobre su período, pero solo hasta su vencimiento y la suma de la función trabajo es evaluada en cada punto. Si el mínimo valor de la función trabajo normalizada por el tiempo, es menor que uno, entonces el sistema es diagramable. La evaluación de esta ecuación requiere el empleo de técnicas iterativas.

Aquí es necesario que las tareas sean periódicas y que sus vencimientos sean menor o igual que el período. Las tareas aperiódicas, caracterizadas por estrictos requerimientos de tiempo de respuesta y arribos aleatorios, podrían ser incluidas empleando un servidor periódico. En este trabajo no serán tenidas en cuenta; sin embargo podrían caer dentro de esta misma metodología.

2.1 Evaluación de Performance

En [8] se definió una metodología que permite evaluar la performance y realizar una ingeniería de sistemas multitarea/monoprocesador en base a una técnica denominada “Breakdown Utilization” (U^*).

En [1] y [2] se plantea una alternativa que permite evaluar la performance en base a la medida del tiempo de ejecución de una tarea ociosa; es decir, mayor tiempo ocioso nos da una medida del margen de error admisible en la estimación del tiempo de ejecución de las tareas o la posibilidad de agregar nuevas tareas dentro del mismo esquema de diagramación, manteniendo la factibilidad del sistema. Ambos trabajos emplearon una estructura que permite evaluar la toma de decisiones para el diseño de hardware y software en aplicaciones de TR que maximicen la diagramabilidad de un conjunto de tareas periódicas. En el análisis se incorporó el costo de la diagramación en diferentes tipos de implementaciones. Además se empleó un razonamiento cuantitativo que permite comparar la performance de los mismos, empleándose como unidad de medida el tiempo de ejecución de una tarea ociosa [4]. Por último se identificaron los segmentos de procesamientos atómicos que ocurren en cada conmutación de contexto, interrupciones, etc., para incluirlos como costos de bloqueo y/o “overhead” [7] en las ecuaciones de estudio de diagramabilidad (1).

2.2 Análisis de Implementaciones de Diagramación

En esta sección veremos el análisis de diagramadores sobre dos tipos de implementaciones, donde se mostrarán los costos de bloqueo y “overhead” para un diagramador perfectamente apropiativo.

A continuación se resumirán las suposiciones asumidas para el análisis:

1. El manejo de interrupciones debe salvar un contexto mínimo de registros que permita procesar una interrupción y obtener información de la tarea. Si el diagramador no realiza apropiación el contexto de registros es restaurado y la tarea activa continúa su ejecución.
2. Un cambio de contexto requiere salvar el contexto actual y cargar el contexto de la nueva tarea. Luego esta última pasa como *tarea activa*.
3. Cuando una tarea finaliza su ejecución se podría: dejar que se venza el tiempo remanente hasta el fin del tiempo de ranura, o implementar una función de salida al kernel del diagramador. De cualquier manera sería necesario restaurar el Bloque de Control de Proceso (PCB) de la tarea finalizada en la *cola de espera* y seleccionar la nueva *tarea activa* de la *cola de listas*.
4. Cuando no hay tareas en la *cola de listas*, el sistema correrá una tarea ociosa que posee el más bajo nivel de prioridad. Ésta se asume como una tarea más, por lo tanto requiere cambio de contexto para salvar y restaurar sus registros, simplificando el análisis.
5. Se considera que una tarea que se encontraba como activa ha finalizado, cuando su PCB es restaurado en la *cola de espera*. Esto garantizará que se preserve el estado correcto de la tarea para la próxima generación.
6. El diagramador es perfectamente apropiativo, es decir que no existen rutinas del "kernel" que no puedan ser interrumpidas.

2.2.1 Definiciones y Nomenclaturas

- Sean dos tareas τ_i y τ_j con prioridades P_i y P_j respectivamente. Si la prioridad de la tarea τ_i es mayor que la de la tarea τ_j, luego $i < j$ ($P_i > P_j$).
- C_{int}: Tiempo de manejo de una interrupción. Este incluye salvar los registros para procesar la interrupción e invocar al despachador.
- C_{desp}: Tiempo de ejecución del despachador para determinar la próxima tarea a ejecutar. Incluye pasar los PCBs de la *cola de espera* a la *de listas* y comparar la primera con la tarea activa.
- C_{alm}: Tiempo utilizado para guardar el estado de la tarea activa e insertar su PCB en la *cola de listas*.
- C_{carga}: Tiempo de carga de la nueva tarea activa desde la *cola de listas*.
- $C_{sys} = C_{int} + C_{alm} + C_{desp} + C_{carga}$: Tiempo de apropiación del sistema.
- C_{recup}: Tiempo necesario para retornar a la tarea activa cuando no se produce apropiación. Incluye el tiempo para recuperar el contexto de registros almacenado por el manejo de la interrupción y retornar a la ejecución normal de la tarea.
- C_i : Tiempo de duración de la ejecución de la tarea i.
- T_i: Período de generación de los pedidos de las tareas.
- T_r: Período de interrupción del timer o tiempo de ranura.
- C_{trap}: En el caso que se implemente la devolución del uso del procesador por parte de una tarea antes que finalice la ranura. Este incluye almacenar el PCB de la tarea en la *cola de espera* y seleccionar de la cola de listas a la nueva tarea activa.

De aquí en adelante trabajaremos con las condiciones de diagramabilidad dadas en (1), donde se le introducirán las condiciones de bloqueo y "overhead", aplicadas al peor caso. Este caso de carga no ocurre nunca en la práctica, sin embargo es importante emplearlo, ya que garantiza una mayor diagramabilidad que la calculada por modelo y mucho mayor aún que la desarrollada por los cálculos teóricos de un sistema ideal.

2.2.2 Esquemas de Diagramación

Los diagramadores se dividirán en dos grandes categorías: *Por Eventos y Por Timer*

Las implementaciones *por eventos* dependen de un hardware externo que generan las interrupciones que determinan los períodos de las tareas. Las implementaciones *por timer* dependen de una interrupción provocada por un timer en forma periódica, que permite que corra el despachador de tareas, llevando los valores de tiempo para evaluar los períodos de las tareas y decidir cuándo invocarlas.

2.2.2.1 Diagramadores por Eventos

Esta implementación se basa en un controlador de interrupciones por hardware cuyas prioridades coinciden con las prioridades asignadas por software a las tareas. Todas las tareas son inicializadas por una interrupción externa. En el comienzo del período de la tarea, una interrupción arriba al procesador. La apropiación del procesador se producirá únicamente si la prioridad de la interrupción que arriba es estrictamente mayor que la prioridad de la tarea activa. Por lo tanto no se producirán bloqueos o inversión de prioridades debido al manejo de interrupciones de tareas de menor prioridad. Si múltiples interrupciones arriban simultáneamente, únicamente la de mayor prioridad será atendida. Todas las características antes mencionadas, requieren de un hardware externo específico.

Definiremos los tiempos de *apropiación* (C_{sys}) y de *salida* (C_{exit}) de una tarea como:

$C_{sys} = C_{int} + C_{alm} + C_{desp} + C_{carga}$ y $C_{exit} = C_{trap} + C_{carga}$

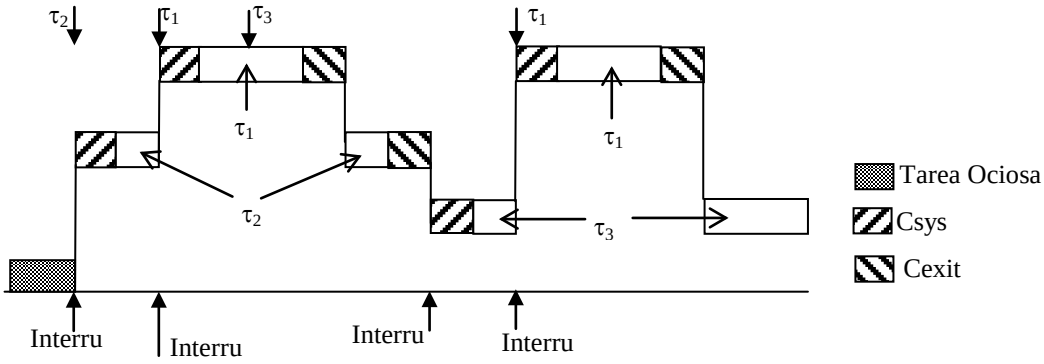


Figura 1: Diagramador por Eventos

En la figura 1 se muestra un esquema de funcionamiento de este tipo de diagramadores. Agregando el costo de diagramación en (1), es posible obtener el peor caso de "overhead" para un sistema por prioridades fijas. Por lo tanto un conjunto de n tareas operando bajo el esquema de eventos integrados será diagramable si cumple:

$$\forall i, 1 \le i \le n, \min_{0 \langle t \langle D_i} \left[\sum_{j=1}^{i} \frac{C_j + C_{sys} + C_{exit}}{t} \right] \le 1 \qquad (2)$$

Este esquema es totalmente apropiativo y no presenta problemas de bloqueos o inversión de prioridades.

2.2.2.2 Diagramadores por Timer

El esquema de manejo por timer, a diferencia del manejo por interrupciones, consiste en un timer periódico que interrumpe la ejecución e invoca al diagramador. Aquí el diagramador actualiza internamente el tiempo de las tareas y evalúa las colas de espera y de listas. En

sistemas operativos comerciales los valores típicos de interrupción oscilan entre 1 a 10 msegs.

En esta implementación el timer interrumpe al sistema cada T_r segundos (ranura) forzando un punto de diagramabilidad. Esta interrupción es manejada por medio de una rutina que se encarga de actualizar el tiempo interno del procesador y ejecutar al diagramador. Este se encarga de mover todas las tareas, cuyo período de generación es mayor que el tiempo corriente, de la cola de espera a la cola de listas y decidir si se apropia del procesador, basado en la prioridad de la tarea que se ubica en la cabeza de la cola de listas. Si una tarea completa su ejecución antes del próximo tic del timer, es posible que: ejecute una función de salida que automáticamente despache la próxima tarea de la cola de listas o bien que deje vencer el tiempo remanente hasta el próximo tic del timer y el diagramador decida a quién le asigna el uso del procesador.

Debido a la granularidad del timer, una tarea de alta prioridad puede ser bloqueada por una de menor prioridad. Supongamos que una tarea τ_i de alta prioridad genera el pedido un infinitésimo ($\varepsilon > 0$) después que una tarea τ_j de baja prioridad comenzó su ejecución. Por lo tanto τ_i será bloqueada hasta el próximo tic del timer.

El "overhad" producido por el manejo de interrupciones del timer (C_{timer}) se considera constante ocurra o no una apropiación y se define como:

$$C_{timer} = C_{int} + C_{desp} + C_{recup} \qquad (3)$$

Los tiempo de apropiación y salida (en caso que se implemente una función de salida antes del vencimiento de la ranura) serán:

$$C_{aprop} = C_{alm} + C_{carga} \qquad (4)$$

$$C_{exit} = C_{trap} + C_{carga}$$

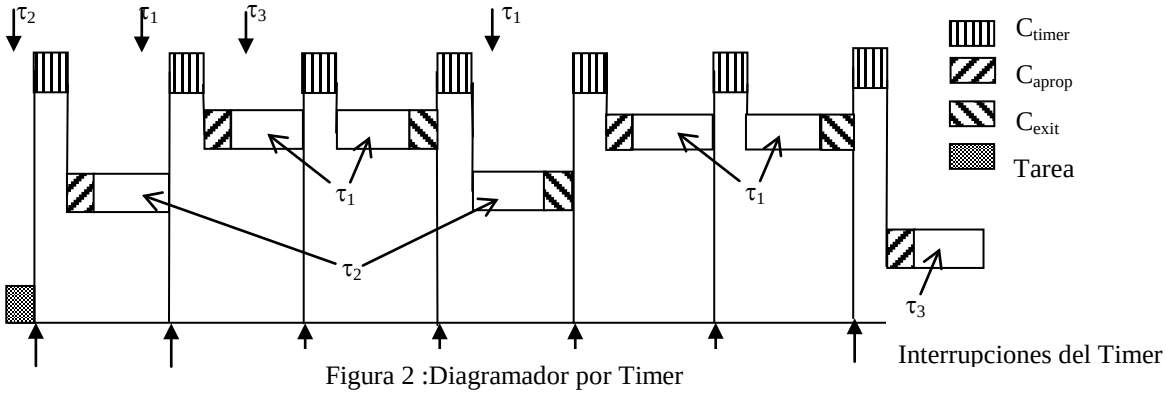


Figura 2 :Diagramador por Timer

La figura 2 muestra un diagramador corriendo bajo PF manejado por timer.

El peor caso de "overhead", para este tipo de diagramador, teniendo en cuenta lo visto en los diagramadores anteriores, sería:

$$C_{aprop} + C_{exit}$$

El peor caso de "overhead" provocado por el timer, en un intervalo t, siendo T_r la resolución del timer, sería:

$$\left\lceil \frac{t}{T_r} \right\rceil C_{timer}$$

Por último podemos decir que el peor caso de bloqueo de una tarea, sería la resolución del timer, T_r. Esto es así, ya que si una tarea τ_i de mayor prioridad arriba al procesador un infenitésimo después del Tic del timer, cuando el diagramador estaba sirviendo a una τ_j de menor prioridad, la tarea τ_i se considera bloqueada hasta la próxima interrupción del timer.

Por último podemos establecer las condiciones de diagramabilidad para un conjunto de n tareas corriendo bajo esta implementación:

$$\forall i, 1 \le i \le n, \min_{0 \langle t \langle D_i} \left[\sum_{j=1}^{i} \frac{C_j + C_{aprop} + C_{exit}}{t} \left(\left\lceil \frac{t}{T_j} \right\rceil - \left\lceil \frac{t - C_{timer}}{T_j} \right\rceil \right) + \frac{C_{timer}}{t} + \frac{T_r}{t} \right] \le 1 \qquad (5)$$

En este punto es posible encontrar los límites de la resolución del timer basados en la diagramabilidad de un conjunto de tareas. El tiempo de ranura se puede disminuir hasta que la CPU pueda correr la rutina de servicio de interrupción del timer y el diagramador. Obviamente en este punto el sistema ya no es diagramable, pues ninguna tarea puede correr. Es posible, aplicando métodos iterativos [8], encontrar el límite inferior. En [1] y [2] se desarrolló un método de aproximación experimental para estimar dicho límite. Por otro lado es posible aumentar el tiempo de ranura hasta un punto tal que el tiempo de bloqueo de las tareas de mayor prioridad comprometan la diagramabilidad del sistema. En dichas referencias se presentan métodos alternativos para hallar el punto de ranura óptimo que le da la mejor performance al sistema.

3. Diagramación por Eventos Temporizados

La diagramación por eventos temporizados consiste en modificar los diagramadores por timer de forma tal que el tiempo de duración de la ranura de trabajo no sea fija, sino que varíe dinámicamente brindando la mejor performance al STR, reduciendo el "overhead" del sistema y eliminando los bloqueos o inversiones de prioridades, llevando a este tipo de diagramador a comportarse como si fuera manejado por eventos.

Reducir el "overhead" del Sistema

Un caso típico de "*overhead*" se produce cuando una tarea tiene un tiempo de ejecución C_i mayor que el tiempo de ranura, T_r. Cuando el tiempo de duración de la ranura es constante, una tarea debe ser interrumpida por el timer a pesar de no haber pedidos de otras tareas de mayor prioridad. En la tarea 3 de la figura 3 se aprecia dicho fenómeno, el cual podría ser evitado si la ranura no fuese de duración fija, sino que se adaptase a la duración de dicha tarea.

Sean: $C_{timer} = C_{int} + C_{desp} + C_{recup}$ y $C_{aprop} = C_{carga} + C_{alm}$

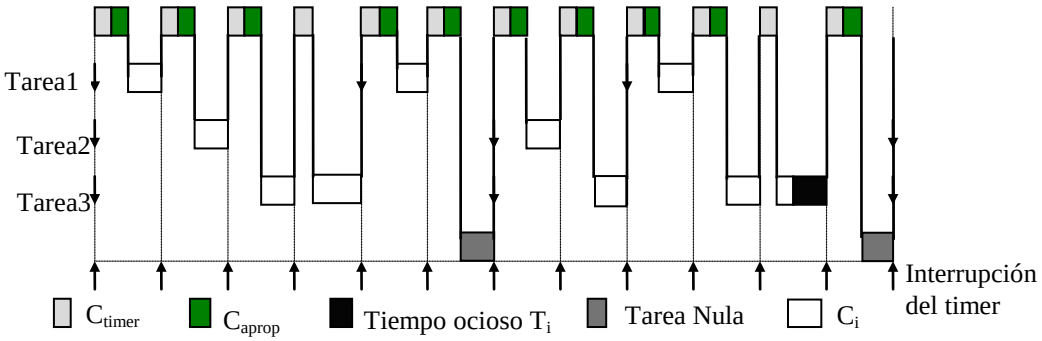


Figura 3

Eliminar los bloqueos o inversión de prioridades

Un caso típico de bloqueo o inversión de prioridades se produce cuando el arribo del pedido de una tarea i no está en sincronismo con el arribo de la interrupción del timer tal como se muestra en la figura 4. Este tipo de problemas puede llegar a complicar la diagramabilidad del sistema cuando el mismo se encuentra en el límite de la saturación.

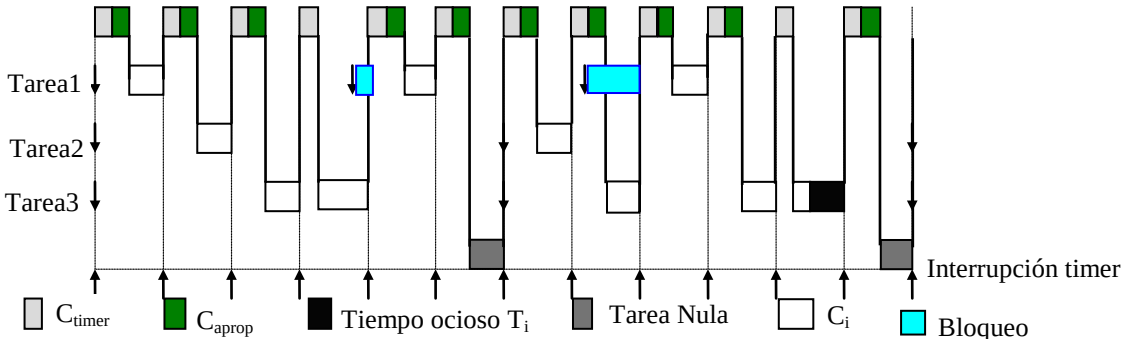


Figura 4

3.1 Arquitectura Propuesta

En esta sección, propondremos una arquitectura que funcione bajo las premisas previamente mencionadas. En principio re-definiremos el término "Ranura":

Se considera Tiempo de Ranura al tiempo transcurrido a partir del momento del arribo de la interrupción del timer hasta que se produzca la próxima interrupción del timer.

En la arquitectura propuesta, los costos fijos impuestos a cada tarea consiste de dos secciones S_1 y S_2 (figura 5) que se ejecutan antes y después de la ejecución de la tarea respectivamente. Las funciones de las secciones son las siguientes:

- S_1 es la sección del sistema que corre únicamente después de la interrupción del Timer (figura 6) y su función es la de programar al Timer para que interrumpa con el arribo de la próxima tarea j, siempre que $P_j \geq P_i$. Esta sección no va a correr en todas las tareas, ya que las tareas de prioridades más bajas, serán despachadas por las secciones S_2 de las tareas de mayor prioridad.
- S_2 es la sección que corre con el fin de cada tarea i. Esta debe correr siempre. La función de esta sección es la de despachar a la próxima tarea k, siempre que $P_k < P_i$. Esta sección además debe actualizar el timer en caso de que existan tareas intermedias entre el nivel i y el k cuyos pedidos se produzcan antes que el valor programado en el timer (el que debería realizar la próxima interrupción). Esto produciría la "ranura variable".

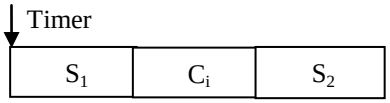


Figura 5 : "Overhead" por Eventos Temporizados

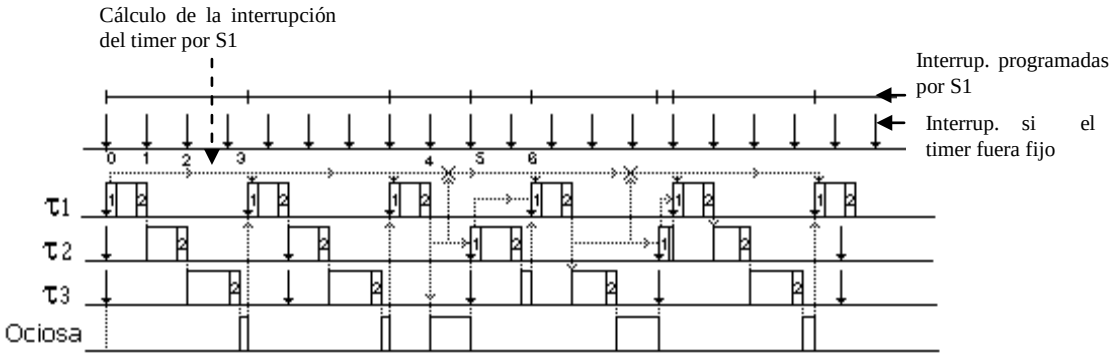


Figura 6: Despachador por Eventos Temporizados

En la figura 6, se representa la evolución de un sistema de tres tareas de TR. En el instante t = 0, se generan los tres pedidos simultáneos (peor caso de carga). Se genera la

interrupción del timer y se ejecuta la sección S_1 asociada a la primer tarea, programando el timer para que interrumpa con el arribo de la próxima tarea de mayor prioridad (en este caso coincide con el arribo del próximo pedido de ella misma por ser la de mayor prioridad). En el instante t =1, termina la ejecución de la sección S_2 de la tarea 1, despachando la tarea 2. Este proceso continúa hasta que se despachan todas las tareas o el sistema es interrumpido por el timer, requiriendo la ejecución de una tarea de mayor prioridad. En el instante t=4, vemos que al correr la sección S_2, no puede despachar ninguna tarea, ya que no hay pedidos pendientes, pero encuentra que es necesario reprogramar el timer, ya que el arribo de la tarea 2 (t=5) se producirá antes que el de la tarea 1 (t=6, tiempo anterior de programación del timer).

Como se aprecia en la figura 6, este mecanismo de despacho de tareas produce el mayor tiempo ocioso, al eliminar las inversiones de prioridades y reducir considerablemente el overhead del sistema.

3.2 Implementación

La implementación del despachador por eventos temporizados consiste esencialmente en un contador por hardware (cuya cantidad de bits depende de los períodos de las tareas) denominado CT_r. Este es un contador ascendente que será inicializado en "0" únicamente por la rutina de atención del timer (es decir por la sección S_1) y cuyo máximo valor de cuenta será el tiempo de la ranura actual.

CT_R es un registro que lleva un valor "estimado" de la ranura actual y es siempre modificado por la sección S_1 y puede ser modificado por la sección S_2, cuando sea necesario cambiar el valor de la ranura.

Existe además un contador por cada tarea (Ct_i). Estos son contadores descendentes que se cargan con el valor del período de generación (T_i) de cada tarea, en el momento de realizarse el pedido de la misma. Luego se decrementan hasta llegar a "0". Estos contadores son actualizados por las secciones S_1 y S_2 de acuerdo a la siguiente ecuación:

$$Ct_i = Ct_i - \Delta CT_r$$

Se define un bit de pedido por tarea B_i tal que:

- Si $B_i = 0$ implica que no existe un pedido pendiente.
- Si $B_i = 1$ implica que existe un pedido pendiente.

Como se ha dicho, la sección S_2 debe actualizar el timer en caso de que existan tareas intermedias entre el nivel i (último nivel ejecutado) y el k (próximo nivel a ejecutar) cuyos pedidos se produzcan antes que el valor programado en el timer y cuyo B_j sea "0". ($P_i > P_j > P_k$).

Para ello debe comparar el valor de cada contador con el tiempo remanente (t_{rem}) hasta que se produzca la interrupción.

$$t_{rem} = CT_R - CT_r$$

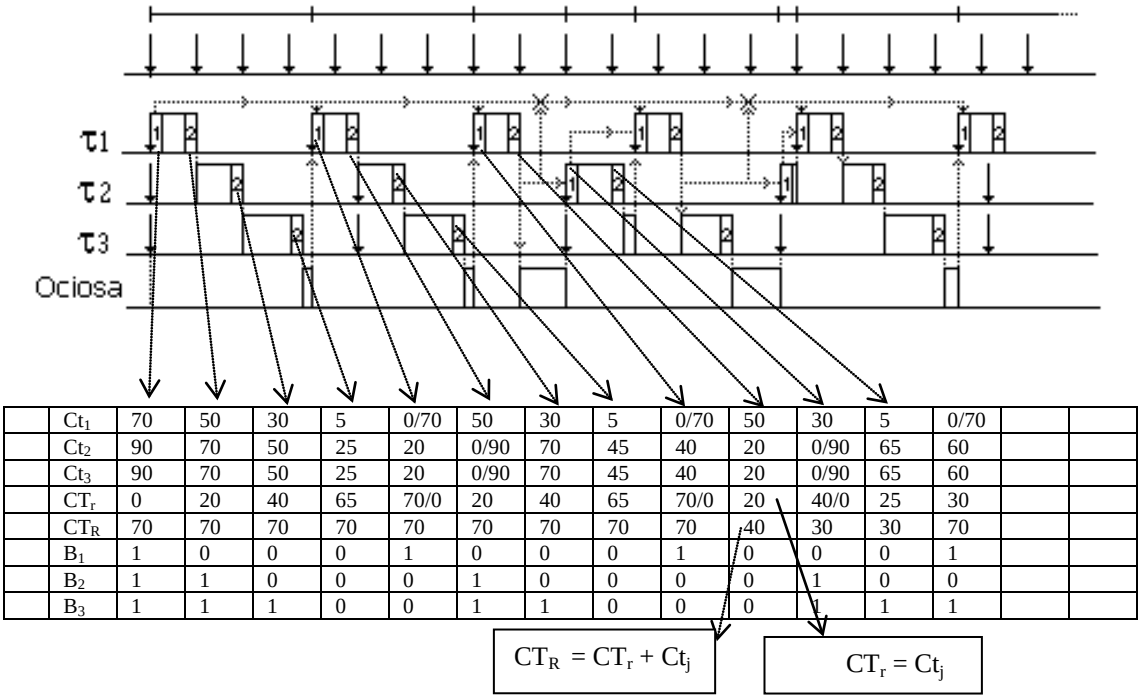
- Si $Ct_j - t_{rem} \geq 0$, no se debe alterar ni el timer ni el contador.
- Si $Ct_j - t_{rem} < 0$ $\therefore$
  - $CT_R = CT_r + Ct_j$
  - $CT_r = Ct_j$

En la figura 7 se muestra la evolución de los contadores (Ct_1, Ct_2 y Ct_3) con los diferentes eventos, los valores del contador por hardware CT_r y del tiempo "estimado" de la ranura actual, CT_R. En la parte final de la tabla se ve los pedidos pendientes Bi.

Como se ha mencionado, ésta propuesta se caracteriza por tener dos secciones de costos fijos para el sistema, creando una complejidad adicional para la implementación del diagramador. Sin embargo, como se aprecia en la figura 7, es muy importante la reducción de

los costos fijos del timer y la eliminación de los bloqueos, generando mayores tiempos ociosos al sistema.

Si comparamos el esquema de eventos temporizados con ranura fija con el despacho de tareas por eventos temporizados (figuras 6 y 8), es posible apreciar, para el último caso, el mayor tiempo de ejecución asignado a la tarea ociosa, ya que el sistema se comporta en forma similar al esquema de diagramación por eventos (figura 1), el cual presenta las mejores características desde el punto de vista de bloqueos y costos fijos. Si utilizamos dicha tarea como una medida de la performance del sistema, notamos que a mayor tiempo ocioso, mejor será el rendimiento. Además es posible incorporar nuevas tareas de TR que no hubieran sido posibles diagramar empleando ranuras fijas, o despachar tareas de no TR.



	Ct_1	70	50	30	5	0/70	50	30	5	0/70	50	30	5	0/70		
	Ct_2	90	70	50	25	20	0/90	70	45	40	20	0/90	65	60		
	Ct_3	90	70	50	25	20	0/90	70	45	40	20	0/90	65	60		
	CT_r	0	20	40	65	70/0	20	40	65	70/0	20	40/0	25	30		
	CT_R	70	70	70	70	70	70	70	70	70	40	30	30	70		
	B_1	1	0	0	0	1	0	0	0	1	0	0	0	1		
	B_2	1	1	0	0	0	1	0	0	0	0	1	0	0		
	B_3	1	1	1	0	0	1	1	0	0	0	1	1	1		

Figura 7: Ejemplo Eventos Temporizados

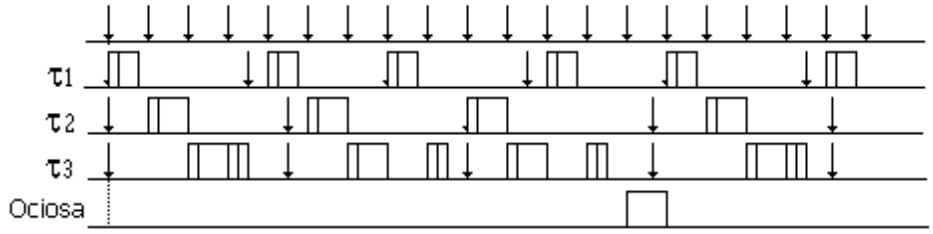


Figura 8: Despacho por ranura fija

4. Resultados y Conclusiones

Un RTOS es una poderosa herramienta diseñada para implementar aplicaciones de tiempo real, permitiendo un desarrollo modular del sistema. Cuando se implementa un despachador no es muy sencillo determinar cuál es el valor apropiado del tiempo de ranura para un dado conjunto de tareas. Obtener la ranura óptima de trabajo brinda la máxima performance del sistema. Sin embargo no logra resolver los problemas típicos de bloqueos ni de “overhead”.

La implementación de despachadores por eventos temporizados, optimizan el empleo de

diagramadores basados en timer, ya que permiten reducir notablemente los costos fijos producidos por las interrupciones del timer sobre tareas que de antemano se sabe que no podrían ser interrumpidas debido a su alta prioridad y eliminar los bloqueos (o inversiones de prioridades) producidos cuando arriba el pedido de una tarea i que no está en sincronismo con el arribo de la interrupción del timer tal como se muestra en la figura 3. Estos problemas pueden llegar a complicar la diagramabilidad de un sistema cuando el mismo se encuentra en el límite de la saturación. Este tipo de diagramadores agrega dos secciones de costos fijos; sin embargo realizando un balance entre complejidad y beneficios, el saldo es muy favorable, ya que mejora considerablemente la performance de los diagramadores basados en timer, permitiendo incorporar nuevas tareas de TR o ejecutar aquellas del tipo no-TR en el tiempo sobrante. Actualmente nos encontramos en una etapa de desarrollo sobre diagramadores basados en arquitecturas Pentiun con el fin de evaluar experimentalmente los beneficios esperados.

La línea de futuros trabajos se basa en la ingeniería de diagramadores integrados implementados por medio de procesadores trabajando en paralelo o empleando circuitos integrados específicos (ASIC).

Referencias

[1] Alimenti O. Mosquera C. y Laiuppa A., "Ranura Optima en un Despachador de Tareas Aplicado a Pequeños Sistemas", Wait´98 - 27° Jornadas Argentinas de Informática e Investigación Operativa (JAIIO). Buenos Aires, Argentina, 31/08 al 04/09 de 1998. pp: 21 – 32.

[2] Alimenti O. y Lagae P. "Monitoreo de un Despachador de Tareas". 25 Jaiio. 1996.

[3] Ardenghi F. y Alimenti O., "Depurador para Multitarea en 386/486DX", Congreso INFOCOM'96. Buenos Aires, Argentina, 11 al 8 15 de junio de 1996. pp: 19-28.

[4] Byung-Do Rhee y otros. "Issues of Advanced Architectural. Features in the Design of a Timing Tool". 11th IEEE Workshop on RTOSS 1994.

[5] C. L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in Hard Real Time Environment", J.ACM 20 (1) 1973, 46-61.3

[6] Chatterjee S. and Strosnider J. "Quantitative Analysis of Hardware Support for Real-Time Operating Systems". Real Time Systems 1996.

[7] Chatterjee S., Reed R., Shehady R. and Strosnider J., "Re-Designing the Real-Time Microkernel for the Intel 80960XA Microprocessor". International Phoenix Conference on Computer and Communications'94". 1994.

[8] Katcher, Arakawa. "Engineering and Analysis of Fixed Priority Schedulers", FTP Server CMU-94.

[9] Lehoczky J.P., Sha L and Ding Y, "The Rate Monotonic Scheduling Algorithm. Exact Caracterization and average case behaviour", Proc. IEEE Real-Time System Symp., 1987.

[10] Orozco J. "Factibilidad deSistemas de Tiempo Real". Tesis Doctoral. UNS 1998.

[11] Plattner Bernhard. "Real-Time Execution Monitoring", IEEE Trans. Software Engineer, Vol. SE-10 6 Nov. 1984, pp756-764.

[12] Santos J. y Orozco J. "Rate Monotonic Scheduling in Hard Real-Time Systems" Information Processing Letters. 1993.

[13] Santos J., Orozco J. y Alimenti O.. "Perfomance Evaluation of Standard Lan Protocols in Time Constrained Enviroments", Anales *IEEE INFOCOM'89*, Ottawa, Canadá, 1989.

[14] Schultz. "C and the 8051 Programming for Multitasking". Prentice Hall. 1993.

[15] Silberschatz, & Galvin . "Operating System Concepts", Addison-Wesley, 1994.