Modificación de una Herramienta de Supervisión de Procesos

Caropreso Fernanda, Pascual Silvia, Tirelli Juan Pablo, Zuzek Alejandro
{mc3o,sp7l,jt6i,az9k}@dc.uba.ar
Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Abstract

En el presente trabajo se modificó un sistema supervisor de tareas para que pueda ser ejecutado bajo la versión de tiempo real del sistema operativo Minix. El sistema original tiene restricciones de tiempo real pero se ejecuta sobre un sistema operativo de propósito general. La modificación consistió en reestructurar el sistema para aprovechar los servicios adicionales que provee RT Minix. Obtuvimos un sistema más simple, con mejor rendimiento y con comportamiento predecible.

1. Introducción

La importancia de los sistemas que controlan procesos en el mundo real crece día a día (control de tráfico aéreo, centrales de energía nuclear, robots, control de condiciones ambientales en edificios, control de cadenas de producción, etc.). La función de estos sistemas, llamados de **tiempo real**, es la de interactuar con dispositivos del mundo externo. El orden de ejecución de estos sistemas está determinado, por lo tanto, por el transcurso del tiempo o por la ocurrencia de **eventos externos** y los resultados de los cálculos pueden depender del momento en que se ejecutaron o del tiempo en que se tardo en hacerlo. A diferencia de otros sistemas, que para resolver efectivamente un problema deben ser correctos y hacer un uso razonable de los recursos, los sistemas de tiempo real deben además cumplir con restricciones importantes de tiempo (**metas**), tener alto nivel de seguridad, facilidad de interface con dispostivos externos y estar preparados a dar respuesta a eventos externos. Si los cálculos del sistema causados por un evento deben terminarse en un tiempo máximo en cada y toda ocasión, se dice que sus metas son **duras**, en cambio si el sistema debe tener un tiempo medio de ejecución, tal que medido en un intervalo de tiempo definido, es inferior al tiempo máximo especificado, se dice que sus metas son **blandas**.

Algunos de los sistemas de tiempo real son los sistemas de **supervisión**. Estos sistemas permiten que la computadora provea informes completos del estado de los procesos controlados al operador de la planta, usando habitualmente sistemas de visualización de información más complejos que simples indicadores. Proveen además facilidades para interactuar con la planta. Sus metas suelen ser "blandas", aunque algunas pueden ser "duras", particularmente las asociadas a las condiciones de alarma, que son acciones que se deben tomar de inmediato cuando algún dispositivo indica valores fuera de lo normal o que pueden implicar peligro.

El presente trabajo tiene como objetivo analizar una herramienta formada por una biblioteca de rutinas (**SSDT** [Wai93]) especialmente diseñadas con el fin de desarrollar aplicaciones simples de supervisión de procesos en tiempo real. A partir del análisis de la herramienta se modificaron las rutinas para poder ser ejecutadas bajo la versión en tiempo real del sistema operativo Minix [Wai94].

En la siguiente sección se describen dos algoritmos de Planificación para Sistemas de Tiempo Real provistos por RT-Minix. En la sección 3 de describe el Sistema Operativo RT-Minix y en la sección 4 se describe brevemente la herramienta SSDT. En la sección 5 se describen las modificaciones hechas a la herramienta SSDT para aplicar los servicios de RT-Minix. En la sección 6 se hace una descripción detallada de todos las etapas del trabajo. Finalmente, en las secciones 7 y 8 se describen los resultados y conclusiones.

2. Algoritmos de Planificación para Sistemas de Tiempo Real

Un sistema de Tiempo Real debe ser capaz de responder a tiempo a las tareas que se presentan de acuerdo a las demandas del mundo externo. Para cumplir este objetivo, la planificación de tareas se vuelve un factor decisivo.

Cada tarea debe cumplir su meta (o sea, su ejecución debe realizarse antes de determinado instante). La planificación debe lograr una respuesta **predecible** (es decir, asegurar que cada una cumplirá su meta), inclusive ante eventos urgentes. Debe lograr estabilidad ante sobrecargas, de manera tal que si se pierden metas, se pueda garantizar el cumplimiento de las correspondientes a tareas críticas.

Las políticas tradicionales de planificación no son las más apropiadas para este tipo de sistemas. La forma más sencilla de planificar tareas es ejecutarlas en orden de llegada (**FCFS** - First Come First Served). Esta política es muy simple de implementar, pero tiene varias desventajas. En particular, es de utilidad nula para sistemas de tiempo real, ya que es altamente probable que una tarea con una meta crítica tenga serios retrasos. En FCFS, cualquier tarea con metas blandas que haya llegado primero, será planificada delante de las tareas con metas duras que lleguen luego. A pesar de ello, el algoritmo es utilizado en distintos sistemas operativos para tiempo real, ya que si el conjunto de tareas consiste solamente en tareas periódicas, liberarán el procesador a intervalos regulares. Si las tareas cumplen las metas en su primera invocación, también lo harán en todas las demás.

Algunos sistemas operativos para tiempo real usan la estrategia **Round Robin**: cada tarea posee un tiempo breve de ejecución, llamado **quantum**, luego del cual el procesador se otorga a la próxima tarea en el orden de llegada. Una tarea urgente con alto uso de procesador puede ser desalojada por otra menos urgente. Además, si una tarea tiene una meta mayor que el quantum, es improbable encontrar un plan predecible. Por ende, los sistemas operativos de tiempo real que usan esta estrategia suelen permitir variar esta tasa de interrupción.

La política tradicional que tiene mayor utilidad es la basada en **prioridades externas**: se selecciona la tareas con mayor prioridad entre todas las que estén listas para ejecutar. Las prioridades pueden ser fijas o variables. Los sistemas de prioridad fija son menos flexibles, ya que las prioridades de las tareas no pueden cambiar.

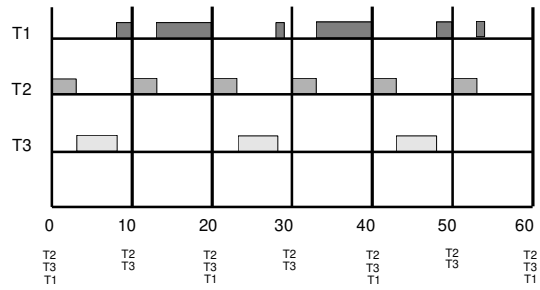

Figura 1 - Planificación según TM

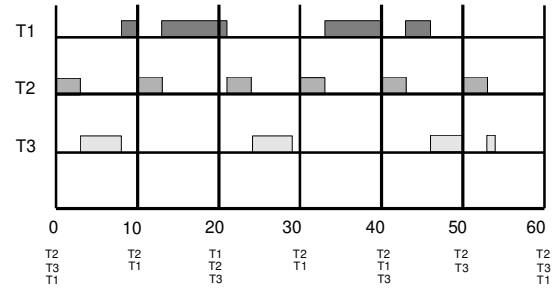

Figura 2 - Planificación según MCP

Los sistemas de prioridades variables pueden permitir que se cambien las prioridades de las tareas durante su ejecución.

Analicemos ahora dos algoritmos tradicionales de planificación en Tiempo Real. El primero llamado de **Tasa Monotónica** [Liu73] usa prioridades fijas y sirve para planificar tareas periódicas independientes ordenándolas en forma monotónicamente creciente dando mayor prioridad a las de menor período de ejecución. Las tareas de alta prioridad pueden remover a las de baja prioridad. Se ha demostrado que este algoritmo es óptimo, en el sentido de que si no existe un plan basado en tasa monotónica para que un conjunto de tareas periódicas independientes cumplan sus metas, no existe ningún otro que pueda hacerlo.

Existe para este algoritmo un test que permite verificar la predictibilidad de un conjunto de tareas dado. Si se cumple que $\sum t_i/p_i \leq n.(2^{1/n}-1)$ entonces se asegura que el conjunto de tareas puede ser planificado por este algoritmo. Si no se verifica esta condición nada puede afirmarse; en otras palabras, la condición es suficiente pero no necesaria. La cota tiende a ln(2) cuando n tiende a infinito, por lo tanto se asegura un 69 % de uso del procesador. El teorema de Zona Crítica [Liu73] permite realizar otro chequeo que es una condición necesaria y suficiente para garantizar un conjunto de tareas. Según este teorema, si una tarea cumple sus metas cuando todas comienzan al mismo tiempo, siempre se cumplirán para toda combinación de tiempos de comienzo. Una forma de aprovechar este teorema es ver si las tareas cumplen su primer meta cuando todas comienzan simultáneamente (como el conjunto de tareas es periódico, luego del mínimo común múltiplo de los períodos de las tares la combinación temporal se repetirá). Ante la aparición de una tarea esporádica, se la puede considerar periódica de una sola instancia y en los cálculos su período es igual a su meta.

Tomemos como ejemplo un conjunto de tareas y veamos que son planificables por Tasa Monotónica. El conjunto consta de tres tareas periódicas e independientes (T_i) con los siguientes períodos (p_i) y tiempos de ejecución (t_i): t_1=10, p_1=30; t_2=3, p_2=10 y t_3=5, p_3=20. En este caso no se verifica la condición $\sum t_i/p_i \leq n.(2^{1/n}-1)$ pues 0,883 > 0,779. Pero usando el teorema de la Zona Crítica se puede verificar que el conjunto de tareas es planificable por Tasa Monotónica porque al cabo de un intervalo de tiempo que es el mínimo común múltiplo de los períodos de las tres tareas (60), ninguna tarea perdió su meta. La Figura 1 muestra este resultado. En cada punto de planificación se muestra debajo del tiempo transcurrido el ordenamiento de la cola de listos. Por el teorema de la Zona Crítica si esto sucede para esta combinación particular de tiempos de inicio de

ejecución (todas comienzan simultáneamene), lo mismo sucederá para cualquier combinación de tiempos de inicio de ejecución.

El segundo algoritmo se denomina **Meta Más Corta Primero** [Liu73]. Es un algoritmo de prioridades variables con remoción, óptimo para planificar tareas con tiempos de llegada aleatorios. Sin embargo, este algoritmo sufre el "efecto dominó" cuando el sistema está sobrecargado, pues ante la pérdida de una meta de una tarea, se pueden llegar a perder todas las subsiguientes. Con este algoritmo las tares con metas más próximas tendrán mayor prioridad. Si para un conjunto de tareas se cumple que $\sum t_i/p_i \leq 1$ entonces ese conjunto de tareas puede ser planificado por Meta Más Corto Primero.

Se puede verificar que el conjunto de tareas que se usó para ejemplificar la planificación por Tasa Monotónica también es planificable por Meta Más Corta Primero, pues $\sum t_i/p_i = 0{,}883 \leq 1$. En la figura 2 se puede observar cómo sería la planificación si todas las tareas se iniciaran simultáneamente. Notar que el orden de ejecución en el instante 20 es T1, T2, T3. Esto es así, porque aún cuando T2 debe cumplir su meta en el mismo tiempo que T1, T1 tiene prioridad por haber estado antes en la cola de listos [Wai95b].

3. El Sistema Operativo RT-Minix

3.1 Cambios en la interface con el usuario

El RT-Minix [Wai95a] es una modificación del sistema operativo Minix ([Tan87], [Tan91]), que introduce algunas facilidades de tiempo real. RT-Minix mantiene la semántica del System V [Bac86] que provee Minix e incorpora 11 nuevas llamadas al kernel para el manejo de tareas con restricciones de tiempo real y tres señales adicionales mediante las cuales el kernel notifica a las tareas la pérdida de metas o sobrecarga del sistema, para que estas puedan tomar acciones correctivas (Tablas 1 y 2).

3.2 Modificaciones hechas al kernel

El planificador de tareas Round-Robin del Minix fue reemplazado por un planificador con una estructura **multicolas** con remoción con tres colas de listos. En la primera se alojan las tareas de tiempo real ordenadas según el algoritmo de planificación Tasa Monotónica o el Meta Más Corta Primero. Estos dos algoritmos están específicamente diseñados para planificación de tareas de tiempo real y fueron abordados con más detalle en la sección anterior. La cola de segundo nivel está dedicada a tareas interactivas, administrada por algoritmos Round-Robin (100 ms de quantum) y de prioridades (fijas o variables). La tercera cola es usada para planificar tareas CPU-bound en background por Round-Robin (200 ms quantum).

El driver del **clock** administra una nueva cola de instancias bloqueadas para demorar tareas de tiempo real. Al comenzar un nuevo período de una tarea, el driver del **clock** remueve su bloque de control de proceso de la cola y lo inserta en la cola de listos de tiempo real. Se prestó atención especial en mantener el overhead de la administración de estas colas al mínimo.

La **granularidad** del reloj del sistema pudo ser aumentada hasta una precisión de 100 microsegundos, sin que el incremento en el overhead de la interrupción de reloj produjera cambios en el comportamiento del sistema.

Llamada al Sistema	Función que provee
set_grain(ticks);	Cambia la granularidad del timer.
ticks=get_grain();	Lee el actual valor de granularidad del timer.
clock();	Da el tiempo en ticks de reloj.
set_per(Ti,Ci);	Declara una tarea en ejecución como tarea de tiempo real periódica. Ti=período de la tarea, Ci=peor tiempo de ejecución.
set_aper(Dl,Ci);	Declara una tarea en ejecución como tarea de tiempo real aperiódica. Dl: meta de la tarea.
inst_end();	Marca la finalización de una instancia de una tarea periódica.
set_pri_int(pri);	Determina la prioridad de una tarea interactiva.

Llamada al Sistema	Función que provee
set_rt_sch(kind);	Selecciona un nuevo planificador de tiempo real.
set_int_sch(kind);	Selecciona un nuevo planificador interactivo.
start_count();	Inicia una nueva cuenta de metas perdidas.
c=miss_result();	Retorna la cantidad de metas perdidas.

Tabla 1 - Nuevas Llamadas al Sistema

Señal	Evento ocurrido
SIG_OVERLD	Se detectó una sobrecarga.
SIG_DL_MISS	Se perdió una meta.
SIG_UNCERTAIN	El planificador avisa que es incierto el cumplimiento de la meta de la tarea.

Tabla 2 - Nuevas Señales

3.3 Ventajas de RT-Minix

Las ventajas que ofrece RT-Minix sobre el Minix para desarrollar aplicaciones de tiempo real son:

- Reducción del tiempo de desarrollo en aproximadamente un 50%, dado que evita tener que configurar "a mano" las prioridades de las tareas y realizar pruebas exhaustivas para asegurar el cumplimiento de las metas ante cada modificación.

- Mayor confiabilidad de los algoritmos de planificación en la ejecución predecible de conjunto de tareas dado.

- Relativa facilidad de incorporación de nuevos algoritmos de planificación de tiempo real.

4. Descripción de la Herramienta para Construir Supervisión de Procesos Original

Se conoce como **sistemas supervisores** a aquellos que proveen al operador (en tiempo real) información detallada de cada uno de los procesos del mundo real con los que interactúa.

Dada la complejidad del desarrollo de sistemas de tiempo real, debida principalmente a la escasez de software de base, técnicas de diseño y verificación para estos sistemas, se presenta SSDT como una herramienta para facilitar el desarrollo de sistemas supervisores en tiempo real.

Esta herramienta fue desarrollada utilizando lenguaje C [Ans90], sobre el sistema operativo Minix, y adoptando Mascot [Jac75] como metodología de diseño. De esto resultó un conjunto de bibliotecas de tipos abstractos de datos, que permiten mantener una imagen del mundo real para controlar y actualizar sus valores, por medio de tares concurrentes que comparten pools, se comunican a través de canales y se sincronizan mediante señales.

Las principales bibliotecas son:

Imagen: Almacena un valor y un estado para cada proceso del mundo real. Mantiene un canal de entrada y uno de salida entre las tareas supervisoras y el mundo real.

Lista de tareas: Mantiene información (como el nombre y la longitud del período) para cada una de las tareas que se ejecutan en el sistema supervisor.

Lista de procesos: Permite asociar dispositivos interactuando con el mundo real con la Imagen del sistema. Mantiene información de los procesos que controla el sistema.

Lista de alarmas: Mantiene información sobre la lista de alarmas de cada tarea (máximas y mininas, tipo de alarma, tarea a ejecutar si se detecta una alarma).

Ports: Se utilizan diversas rutinas para manejar los ports de comunicación.

Sincronizador: Realiza planificación de alto nivel, por encima de la planificación realizada por el sistema operativo. Lo hace mandando señales a las tareas en el momento que le corresponde ejecutar.

El supervisor se construyó haciendo uso de estas bibliotecas y está compuesto por seis tareas que se ejecutan en paralelo. Estas son: *sincr, input, display, driver, historic* y *alarm*. La primera es el padre de las cinco siguientes. Su función consiste en despertar a cada una de sus tareas hijas, ejecutar un ciclo infinito que controla si a alguna de las tareas hijas le toca dispararse y verificar que estén todas vivas aún (si alguno de los procesos muere, el sistema aborta).

Dentro de este ciclo se incrementan los tiempos de espera de las tareas. Cuando el tiempo de espera alcanza al período, comienza su ejecución. Los períodos de estas tareas están en un archivo.

La sincronización se realiza mediante **señales**. Las tareas *display, driver, historic* y *alarm* tienen una inicialización y luego entran en un ciclo infinito. Cada vez que se ejecuta el ciclo, lo primero que hacen es habilitarse para recibir la señal “PETATE” y luego hacen un “pause”, lo que implica que la tarea quedará bloqueada hasta que alguien le envíe dicha señal. El encargado de enviarla es el sincronizador (*main*) cuando detecta

que a la tarea en cuestión le toca ejecutar. La excepción es *input*, que no se bloquea a la espera de una señal, pero si lo hace esperando un *<Enter>* en el teclado. Esta es la única tara interactiva, el resto son todas periódicas; no hay tareas aperiódicas.

5. Aplicación de los servicios de RT-Minix en la modificación de SSDT

Parte de la modificación de la herramienta SSDT consistió en aprovechar las llamadas al sistema adicionales que provee RT-Minix respecto de la versión original de este sistema operativo. La herramienta consiste, como fue descrito en la sección anterior, de cuatro tareas periódicas (*display, driver, alarm e historic*), una interactiva (*input*) y una que las mantiene sincronizadas (*sincr*). Se decidió liberar la tarea *sincr* de la tarea scheduling y dejar al sistema operativo esta función. En vez de bloquear las tareas periódicas en espera de una señal proveniente de *sincr* después de la ejecución de un período, se optó por invocar la llamada que establece una tarea como periódica (set_per) indicando su tiempo de ejecución y su período. Al final de cada período es se invoca la llamada inst_end, que notifica el kernel la finalización de la instancia.

Estas modificaciones tienen sentido para las tareas periódicas, pero para la tarea *input* no son aplicables. Tampoco es conveniente que se ejecute como tarea aperiódica, ya que al ser una tarea interactiva, no se le puede calcular un tiempo de ejecución máximo. Se decidió por lo tanto dejarla como estaba, confiando en que RT-Minix la planificaría como tarea interactiva.

RT-Minix provee una forma muy sencilla de mostrar el estado del contador de **metas perdidas** y de volverlo a cero. Se accede a estas operaciones mediante el teclado y sus resultados aparecen en la pantalla. Se usó este método para saber si no se perdían metas por lo que los tiempos asignados eran correctos o, si por el contrario, se estaban perdiendo metas y los tiempos de las tareas debían ser modificados.

6. Trabajo Realizado

6.1 Modificaciones del SSDT

Para adaptar el sistema existente a RT Minix, básicamente había que liberar al sincronizador del trabajo complejo que estaba realizando, ya que el sistema operativo está preparado para ese tipo de funciones. Se debían usar las nuevas llamadas provistas por RT Minix para definir las tareas periódicas como tales con sus respectivos períodos y tiempos de ejecución y además, definir cuando terminaba una instancia de ejecución.

Las llamadas usadas fueron setper(Ti,Ci) y inst_end() respectivamente. Todo esto se hizo posibilitando que el mismo código pueda ser usado para generar una versión para Minix estándar o para RT-Minix mediante directivas de compilación.

En las figuras 3 y 4 se puede comparar las principales diferencias entre las tareas periódicas originales y las tareas periódicas modificadas.

```
func()
{
inicializacion
while(siempre) {
     signal(PETATE,shh);
     pause();

//   codigo particular de cada tarea

     }
}
```

Figura 3 - Esquema del código de las tareas periódicas originales

6.2 Problemas en la implementación de la modificación

6.2.1 Eliminación de un archivo

El sincronizador (*main*), en su versión original, accede a un archivo con todos los datos de las tareas que controla (PID, nombre, período, tiempo máximo de ejecución, tiempo transcurrido desde la última ejecución, etc.). Para la presente modificación, de toda esta información, sólo se necesita el PID para verificar si todos los procesos hijos aún están vivos. Se decidió no usar más el archivo y reemplazarlo por un array de cinco posiciones para almacenar exclusivamente los pids .

6.2.2 Cambios en las tareas INPUT y DISPLAY

Los inconvenientes más importantes aparecieron con las tareas *input* y *display*. El primero se encarga de leer los comandos ingresados por teclado y modificar los datos del sistema de acuerdo a lo ingresado, pero tiene una comunicación especial mediante un pipe con el segundo (el resto de las comunicaciones entre tareas se hace mediante archivos compartidos). En este pipe, *input* escribe el nombre del proceso sobre el que la tarea *display* debe mostrar los parámetros. *Input* puede también, poner en el pipe la constante Off para indicarle a *display* que deje de escribir en la pantalla. Es propiedad de los pipes bloquear el proceso si éste intenta leer algo cuando el pipe está vacío. Esto es precisamente lo que ocurre en la implementación original del supervisor: la tarea *display* intenta obtener el nombre del proceso sobre el cual mostrar los datos, pero queda bloqueada hasta que *input* no lo escribe. *Input* no escribe nada en el pipe hasta que se ingrese el comando *DISP*, *DIA* o *APAGAR* (en este último caso es cuando escribe Off). En la versión para RT-Minix se debe evitar que una tarea quede bloqueada por algo que no sea la llamada inst_end(), para evitar perder metas. Se solucionó este problema asegurando que en todo momento haya un “token” en el pipe. Siempre que lo lee, la tarea *display* lo vuelve a escribir y luego el *input* lo escribe modificado.

```
func()
{
inicializacion
#ifdef RT
setper(PERIODO, TIEMPO_EJEC);
#endif
while(siempre) {
#ifndef RT
     signal(PETATE,shh);
     pause();
#endif
#ifdef RT
     inst_end();
#endif

//   codigo particular de cada tarea

     }
}
```

Figura 4 - Esquema del código modificado de las tareas periódicas

6.2.3 Comando *REF*

Surgieron también dificultades con el comando *REF*. Este comando fuerza un refresco de pantalla. Originalmente, esto se hace mandando desde la tarea *input* a *display* la misma señal que manda el sincronizador (PETATE), provocando así la ejecución de una instancia de la tarea *display*. Con la nueva versión, ya no es posible hacerlo porque el que administra las instancias de ejecución de las tareas periódicas no es ninguna de las tareas, sino el sistema operativo (que no provee una llamada para forzar la ejecución de una tarea periódica).

Para solucionar este problema, se copió la parte del código de *display* que muestra los datos de un proceso en el código de *input* (en el comando *REF*). Anteriormente se había considerado la posibilidad de realizar el refresco de pantalla en el propio *display*, comunicándoselo desde *input* mediante una señal. *Display* tendría una función asociada a esta señal que haría un refresco de la pantalla, de la misma forma que se hace en el ciclo principal de la tarea. Se intentó llevar a cabo esta alternativa, pero no funcionó, porque para reconocer la señal la tarea debe estar activa y por eso no se ejecuta la función asociada a la señal en el momento de ingresar el comando *REF*, como se pretendía. Además, al ingresar el comando *REF* dos veces consecutivas sin llegar a activar la tarea, ésta moría y por lo tanto el sistema terminaba su ejecución. Otra propuesta que no fue probada, consiste en generar una tarea nueva aperiódica o interactiva que estaría a cargo de estos refrescos y que sería despertada desde INPUT mediante una señal.

6.2.4 Comando *CAMREF*

Otro comando que tuvo que ser revisado fue *CAMREF*. Este comando cambia el período de la tarea *display*, es decir, cambia la tasa de refresco de la pantalla. En la implementación para Minix estándar, *input* se limita a modificar dicho parámetro en el archivo que tiene la información de los tiempos de las tareas del sistema. Y como el sincronizador activa las tareas hijas de acuerdo a los datos de este archivo, queda todo resuelto.

Ahora el que activa las tareas es el sistema operativo y una tarea no puede comunicarle que se cambie el período de alguna otra; esto puede realizarse sólo desde la tarea cuyo período se desea modificar mediante la llamada setper(Ti, Ci). En nuestro caso la llamada debía realizarse desde *display*. El problema fue que el dato ingresa por la tarea de input y había que comunicárselo a *display*. Se decidió implementar esta comunicación entre procesos mediante un pipe cuidando que ninguna de las dos tareas quedara suspendida por tratar de leerlo. *Display* consume un dato del pipe en cada ejecución y en caso de que el valor leído es distinto de la lectura anterior, redefine su tiempo de período. En todos los casos *display* vuelve a colocar el dato leido en el pipe para evitar bloquearse por pipe vacío en la lectura siguiente. Cuando desde *input* se desea modificar el período de *display*, se consume el dato viejo y se coloca uno nuevo. Es importante consumir el dato viejo para preservar la semántica de la comunicación. *Input* es también la tarea encargada de inicializar el pipe poniendo en él como dato la tasa de refresco por defecto

7. Resultados Obtenidos

7.1 Medición de los tiempos de ejecución de las tareas

Los tiempos de ejecución y de período de las tareas en el SSDT original estaban en segundos. Esta limitación estaba impuesta por la precisión del sistema operativo original. Las modificaciones introducidas en el RT-Minix permiten trabajar con una mayor precisión del reloj. Este hecho fue aprovechado para calcular nuevos valores de tiempo de ejecución y de período más cercanos a la realidad.

Para efectuar la medición, se corrieron versiones modificadas de las cuatro tareas periódicas, en las que la parte periódica de la tarea se ejecutaba una única vez. Al comienzo y al final de la parte periódica, las versiones modificadas piden el tiempo actual al sistema operativo. De la diferencia entre estos dos tiempos surge el tiempo de ejecución de la tarea. Se repetieron las mediciones un número significativo de veces, obteniendo que en ninguno de los casos las tareas tardaron en ejecutar más de un tick.[1]

No nos fue provisto el conjunto completo de *stubs* para acceder a las llamadas nuevas. Entre los *stubs* que faltaban, estaban el de la llamada get_grain y set_grain. Se midió la granularidad del reloj del sistema con una tarea que en cada período mostraba un mensaje. Probando diversos tiempos de período llegamos a establecer que con un período de 600 ticks, la tarea se activaba exactamente cada 10 segundos. Por lo tanto, la granularidad del reloj del sistema era de 17 milisegundos aproximadamente.

---

[1] Los tiempos de ejecución de las tareas son dependientes del hardware. En nuestro caso se obtuvieron en una PC con procesador Cyrix 486 DX II de 66 Mhz.

7.2 Pruebas del sistema completo

El sistema fue probado con los tiempos de ejecución calculados y períodos de 4 ticks para todas las tareas periódicas, salvo para la tarea *display*, cuyo período depende de lo ingresado por el usuario. Observamos que con el *display* desactivado, no se perdían metas, pero cuando activábamos la tarea *display* siempre se perdían metas. La cantidad de metas perdidas aumentaba a medida que disminuía el período del *display*. En teoría, un conjunto de cuatro tareas de tiempo máximo de ejecución de un tick y períodos de cuatro, es planificable, pero esto no se cumplía en nuestro caso. Este hecho se puede explicar considerando que no todo el tiempo se puede usar para trabajo útil. Una parte se pierde haciendo el cambio de contexto entre tarea y tarea (overhead introducido por el sistema operativo).

Los tiempos de período fueron reasignados, esta vez en 5 ticks, salvo para el *display*. Con estos valores no se perdían más metas, independientemente del período de ejecución de la tarea *display*.

Para facilitar la tarea de asignación de tiempos de ejecución y de período de las tareas periódicas, se definieron unas constantes adicionales en *const.h* para cada uno de estos tiempos y para cada tarea periódica. De esta forma con sólo modificar los valores en la definición de las constantes y recompilando se obtenía una nueva prueba. Una forma aún mejor es modificando el sistema para que lea estos valores desde un archivo, eliminando así la necesidad de recompilarlo para cada prueba.

7.3 Trabajo Futuro

Como trabajo futuro se propone analizar el impacto que tiene en el comportamiento de la herramienta la modificación de la granularidad del reloj del sistema. Sería interesante poder calcular los tiempos de ejecución de las tareas aún con más precisión.

8. Conclusiones

Teniendo un sistema operativo que realiza la planificación de la ejecución de las tareas se puede lograr que éstas se ejecuten en forma más segura. Con el supervisor de procesos modificado para correr sobre la versión de tiempo real del sistema operativo Minix se puede saber efectivamente si se están cumpliendo las metas. Por el contrario, si para lograr que las tareas se ejecuten en forma periódica se tiene que realizar la planificación en forma explícita desde alguno de los procesos (el sincr en este caso) que puede ser interrumpido por el sistema operativo en cualquier momento, no se puede asegurar predictibilidad en la ejecución de las tareas periódicas controladas.

Trasladando toda la carga de la administración de la ejecución de las tareas al sistema operativo se logra código más reducido y por lo tanto más fácil de entender y mantener. En nuestro caso, sincr en la versión de RT Minix lanza las tareas en la inicialización y luego sólo debe asegurarse que los procesos activados sigan vivos, mientras que en la versión original, debía además planificar el resto de las tareas.

Como los ticks de reloj del RT Minix son más cortos, permiten achicar los períodos de las tareas (antes eran de por lo menos un segundo, ahora son de 1/12 de

segundo). Por lo tanto, en este trabajo se logró una versión del supervisor con mejor rendimiento que la original.

De lo dicho anteriormente, se desprende que para poder desarrollar aplicaciones con restricciones de tiempo real en forma adecuada, es necesario que el soporte de este tipo de características sea brindado desde el sistema operativo.

9. Bibliografía

[Ans90] American National Standard Institute "American National Standard for Information Systems - C Programming Language", ANSI 1990.

[Bac86] Bach, M. "The design of the UNIX Operating System", Prentice - Hall International, 1986.

[Jac75] Jackson, A., Simpson, H. "Mascot - A modular approach to software construction, operation and test", RRE Tech. Note NO.78, 1975

[Liu73] Liu, C., Layland, J. "Scheduling algorithms for multiprogramming in a Hard Real Time System Environment", Journal of the ACM, Vol 20, No 1, 1973, Págs. 46-61.

[Sha93] Shanley, M.C., Wainer, G.A., "Informe sobre Sistemas de Tiempo Real y desarrollo e implementación de un Sistema Supervisor de Procesos", Tesis de Licenciatura, Director Lic. G.Delbue, Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, Marzo 1993.

[Tan87] Tannenbaum Andrew S. "A Unix Clone with Source Code for Operating Systems Courses", ACM Operating Systems Review, Vol. 21 No. 1, Enero 1987

[Tan91] Tannenbaum Andrew S. et al. "Minix 1.5 Reference Manual", Prentice-Hall, 1991

[Wai93] Wainer G.A., "SSDT - Una herramienta para desarrollar Sistemas Supervisores en Tiempo Real", Anales del I Encuentro Chileno de Computación, Ocutbre de 1993, Págs. 44-52.

[Wai94] Wainer G.A., "Implementing Real Time services in MINIX", Operating Systems Review, Abril 1995, Informe interno del Departamento de Computación FCEyN - UBA 1994.

[Wai95a] Wainer G.A., "Implementing Real-Time Scheduling in a Time-Sharing Operating System", Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires

[Wai95b] Wainer G. A., "Algunos resultados de Planificación en Tiempo Real", Departamento de Computación, FCEyN - UBA.