Diseño y caracterización de un núcleo orientado a aplicaciones dedicadas

Matias Ezequiel Vara
Facultad de Ingeniería
Universidad Nacional de La Plata
Argentina
mvara@barcala.ing.unlp.edu.ar

Alejandro L. Veiga
Facultad de Ingeniería
Universidad Nacional de La Plata
Argentina
veiga@fisica.unlp.edu.ar

Abstract— **En este trabajo se presentan los criterios de diseño de un kernel portable para aplicaciones dedicadas o monopropósito, escrito enteramente en Pascal. El núcleo utiliza la asignación dedicada de recursos y la planificación cooperativa de hilos a fin de optimizar el desempeño en entornos multicore. Se compara su desempeño con dos sistemas operativos de propósitos generales, utilizando un algoritmo de ordenamiento paralelizable.**

Kernel; pascal; dedicado; embebido; multicore; portable; multihilo

I. INTRODUCCION

Los Sistemas Operativos (SO) surgieron como una forma de homogeneizar el hardware de las computadoras, facilitando el acceso a los recursos al usuario y realizando un aprovechamiento óptimo de éstos [1]. Debido a esta generalización, cuando se utiliza un SO de uso general para un único propósito puede no estar sacándose provecho del hardware de manera óptima.

Llamamos aplicaciones dedicadas o monopropósito a programas como servidores web o datadrivers, los cuales se encuentran sobre una red pública (internet) o privada y deben atender solicitudes de servicios de miles de usuarios de forma simultánea. Otro ejemplo de este tipo de aplicaciones son los problemas de cálculo numérico y simulaciones para física de altas energías, física de materia condensada, genoma humano, meteorología, entre otros.

Por otro lado tenemos los sistemas embebidos, que también son dedicados para una tarea en particular. En estos tenemos otras restricciones como ser: bajo consumo, memoria reducida y tiempos de respuesta rápidos.

Para este tipo de aplicaciones no existe actualmente una solución particular, sino que se utilizan un SO de uso general ejecutando aplicaciones tradicionales. Se suele dedicar un hardware particular para realizar una tarea específica. Como sólo se ejecuta una aplicación en la computadora, los SO en estos ambientes se denominan SSP (del inglés System for Single Porpouse).

El siguiente trabajo tiene como objetivo mostrar el diseño de un kernel optimizado para el caso de aplicaciones monopropósito y comparar el desempeño del sistema resultante con SO de uso general.

II. DISEÑO DEL KERNEL DEDICADO

El kernel desarrollado es muy sencillo, con llamadas al sistema muy simples, de forma que la mayor parte del tiempo los procesadores estén realizando tarea útil para la aplicación de usuario. De éste modo, las capas del kernel y de arquitectura son muy ligeras. Esto significa que brindan los procedimientos mínimos para que el programador desarrolle la aplicación.

El lenguaje elegido para el desarrollo del proyecto fue Pascal, utilizando el compilador Freepascal [2]. Esta decisión se basa en que la sintaxis permite que el código sea entendido fácilmente, permitiendo simplificar los procedimientos de testeo y resultando de lectura directa para el estudiante.

A continuación se presentarán las premisas de diseño del sistema dedicado y sus diferencias con las implementaciones en SO de uso general.

A. *Kernel incluido en la aplicación de usuario*

Una de las premisas en el diseño del núcleo fue que éste debería estar optimizado para la ejecución de una única aplicación. Para esto se decidió incluir todo el kernel y la aplicación de usuario en un único ejecutable. El kernel fue escrito en forma de unidades sobre el lenguaje Pascal [3].

El compilador ve como un único programa a la aplicación y al kernel. Al compilar el programa dedicado se genera un único ejecutable denominado *toro.exe*. En éste se encuentran linkeadas tanto las librerías del kernel como la aplicación de usuario, siendo transparente para el programador. El resultado es un ejecutable especial debido a que es capaz de arrancar por si solo a la PC, inicializar el kernel y luego ceder la ejecución a la aplicación de usuario.

B. Independencia de la arquitectura

Una de las principales ventajas de TORO es ser independiente de la arquitectura sobre la que se ejecuta. Se logra a partir de la utilización de la unidad denominada *Arch* (por architecture en Inglés). Ésta cuenta con un conjunto de procedimientos que son los mismos para todos los procesadores. Hasta el momento fueron portadas las arquitecturas x86 y x86-64 [4].

La unidad *Arch.pas* es la única que posee código en lenguaje Assembler. Debido a que el Assembler es

dependiente de la arquitectura donde se ejecuta, se evitó su uso en el resto del código y sólo está presente en las partes que son procesador-dependiente como pueden ser los drivers.

C. *Ejecución en modo kernel*

En TORO tanto el código del kernel como el de usuario se ejecutan con el máximo nivel de privilegio, por lo tanto no existe distinción entre el código del kernel y el de usuario. Al intentar comunicarse la aplicación de usuario con el kernel para pedir algún servicio, lo hace simplemente ejecutando un procedimiento de una librería del kernel. Así no es necesario implementar una interrupción para lograr la comunicación, con todo lo que ella conlleva: cambios de contextos, cambios de niveles de privilegio, etc.

D. *Planificación cooperativa*

En TORO la unidad de ejecución son los hilos. El encargado de la asignación de hilos es el planificador; éste utiliza el algoritmo de hilo cooperativo. La asignación de hilos es siempre local, por lo que cada planificador es independiente del resto de los procesadores. Una vez asignado un hilo a un procesador, el hilo retorna el control al kernel a través de una llamada al sistema, informando que su tarea terminó y que puede planificar otro hilo.

La elección de este algoritmo de planificación forma parte del diseño intrínseco del kernel, y está acompañado por otros puntos de diseño. El modelo de hilo cooperativo está íntimamente relacionado con los mecanismos necesarios para hacer cumplir la exclusión mutua y con las aplicaciones que se ejecutarán sobre TORO.

Un punto muy importante en un SO es la manera en que se implementa el cambio de contexto. Esta es una tarea crítica debido a que se ejecuta de forma continua. El cambio de contexto implementado en TORO es por software. Se utilizan técnicas de programación para no depender de los mecanismos de cambio de contexto que brinda un hardware en particular. Como el cambio de contexto se realiza siempre luego de haberse invocado una llamada al sistema, el planificador supone que en ese instante los registros del procesador no están siendo utilizados por la aplicación de usuario. De esta forma se limita únicamente a salvar el estado de la pila del hilo que debe ser removido.

El cambio de contexto implementado en TORO es más rápido que el de hardware y que el implementado en un SO de uso general. La elección del método de cambio de contexto se encuentra directamente relacionado con el modelo de hilo cooperativo.

E. *Migración de hilos entre procesadores*

En ambientes multicore es preciso contar con la capacidad de crear hilos no sólo en el procesador local sino en procesadores remotos. TORO brinda la posibilidad de crear hilos en cualquier procesador, desde cualquier procesador, a través de una llamada al sistema.

Se pueden diferenciar dos procedimientos en el kernel:

i. La emigración de hilos: cuando los hilos se dirigen hacia otro procesador diferente al que está ejecutando el hilo que los creo.

ii. La inmigración de hilos: cuando el procesador huésped encola en el planificador local los hilos que provienen de otros procesadores.

Éste es el único punto del núcleo en el cual se requiere de algún tipo de mecanismo de sincronización entre los procesadores para poder enviar y recibir los procesos que deben migrar. La inmigración y emigración de hilos entre procesadores se realiza únicamente cuando se invoca al planificador (cuando el hilo cede el procesador al kernel).

F. *Dedicación de recursos*

En TORO, para reducir los problemas provocados por estructuras de memoria compartidas entre los procesadores, se hace uso de la dedicación de recursos (disco duro, memoria, interfaz de red, etc) [5]. La dedicación se realiza a un procesador dado, de forma que sólo este procesador accederá al recurso. La dedicación queda en manos del programador quien decide a qué procesador dedica cada recurso. Esto evita la competencia entre procesadores para acceder a una región de memoria dada, debido a que para cada recurso el kernel debe tener estructuras que guardan información acerca del él. Cuando se realiza la dedicación de un recurso, sólo ése procesador puede realizar operaciones de escritura/lectura sobre él.

Si bien este mecanismo incrementa el trabajo del programador, la aplicación posee una ejecución más limpia debido a que se sabe de antemano qué procedimiento se ejecutará en cierto procesador y permite dividir el problema principal en problemas independientes que se ejecutan en procesadores de forma paralela.

G. *Manejo de memoria*

El modelo de memoria implementado es plano, o sea que no se hace uso de la paginación ni de la segmentación. Con ésto se gana portabilidad y la asignación de memoria resulta más sencilla. La memoria –como todo recurso- es dedicada. El modelo utilizado es el NUMA. Los asignadores de memoria que se ejecutan en cada procesador son independientes uno del otro, evitando la necesidad de sincronización entre ellos.

Ésto se hizo con el fin de aprovechar las últimas tecnologías de acceso a memoria como HyperTransport [6] e Intel QPI [7]. Debido a que la asignación de memoria para un dado procesador está restringida a una región, no hay problemas de que una misma línea de memoria se encuentre en dos cache diferentes, reduciendo el *cache line boucing*. Se fuerza la utilización de memoria local para cada procesador, reduciendo la utilización de memoria remota, la cual presenta una alta latencia.

H. *Acceso no bloqueante a la red*

Como todo recurso, el acceso a red es dedicado a un procesador dado y es el usuario quien realiza la dedicación. La principal diferencia que presenta el stack TCP/IP con respecto a otros SO, es que los procedimientos que se utilizan para el acceso a la red son no-bloqueantes. Ésto quiere decir que cuando se realizan operaciones de red de escritura o lectura el hilo no se duerme esperando que los datos estén listos. Por el contrario, se utilizan flags que informan al hilo. Mientras espera que los datos estén listos, el hilo puede estar realizando

otra actividad, como por ejemplo leyendo o escribiendo sobre otra conexión.

Esto permite que un único hilo manipule miles de comunicaciones a la vez, y de esta forma se descongestiona el planificador debido a que se disminuye el número de hilos y el tiempo que se pierde al realizar el cambio de contexto entre cada hilo.

III. EVALUACIÓN DEL SISTEMA

TORO saca el máximo provecho de aplicaciones multihilo. Por lo tanto, al momento de elegir el algoritmo para realizar las pruebas, se buscó uno que pudiera ser descompuesto en problemas más sencillos y que no involucre excesiva comunicación entre las unidades de ejecución. Si bien esta restricción parece arbitraria, es una característica que representa a una gran parte de las aplicaciones que requieren performance masiva como pueden ser: servidores web, análisis del genoma humano, física de partículas, etc.

La aplicación elegida fue ejecutada en entornos TORO, Linux y Windows. Debido a que TORO permite configurar el hardware en el que se ejecuta la aplicación de manera óptima, y que la ejecución del SO está dedicado a esta aplicación, se espera que se obtenga un mejor rendimiento en comparación con un SO de uso general.

A. El algoritmo evaluado

El algoritmo elegido fue el de ordenación de números denominado *MergeSort* u ordenamiento por mezcla. A diferencia de los algoritmos de ordenación seriales como el de la burbuja o *QuickSort*, éste permite ser paralelizado muy eficientemente. A partir de una tabla de números no ordenada, se divide la tabla en grupos y se realiza la ordenación parcial de cada uno (sort). Finalmente se intercalan (merge) los grupos ordenados.

B. Parámetro comparado

Se realizaron medidas del número de ciclos de reloj que utiliza la aplicación para ordenar N elementos. Se prefirió medir los ciclos de reloj en vez de tiempos, debido a que la velocidad de reloj varía máquina a máquina. En cambio si medimos ciclos de reloj, obtenemos una medida mucho más general mientras utilicemos siempre la misma arquitectura de computadora.

El tiempo que tarda la aplicación en realizar el ordenamiento parcial de los elementos (tiempo de sort), puede ser escrito como:

$$t_{sort\,total} = t_{sort} + t_{planificador} \qquad (1)$$

Siendo t_{sort} el tiempo neto destinado al ordenamiento parcial y $t_{planificador}$ el tiempo que se pierde en los cambios de contexto del planificador y tareas del sistema.

Luego, podemos decir que:

$$t_{sort} \propto n_G . n_e^2 \qquad (2)$$

$$t_{planificador} \propto n_G \qquad (3)$$

Siendo n_G el número de grupos y n_e el número de elementos por grupo. La ventaja de este algoritmo radica en que cada grupo de elementos puede ser ordenado independiente del resto de los grupos. Cada grupo puede ser ordenado por un único hilo o por múltiples hilos corriendo en procesadores diferentes.

Luego si $n_e^2 = (\frac{n_E}{n_G})^2$, con n_e el número de elementos total, reemplazando en (1) podemos decir que:

$$t_{sort\,total} = k'.\frac{{n_E}^2}{n_G} + k_{sistema}.n_G \qquad (4)$$

Siendo $k_{sistema}$ y k' constantes de proporcionalidad característica del SO sobre la que se ejecuta la aplicación. Para $k'.(\frac{n_E}{n_G})^2 \ll k_{sistema}.n_G$ el tiempo tendrá un comportamiento lineal, determinado por $k_{sistema}$. Conocer el valor de $k_{sistema}$ será útil para observar cómo se comportan los distintos SO ante un incremento de la paralelización.

C. Implementación

La aplicación de prueba fue escrita en lenguaje Pascal, utilizando el compilador Freepascal versión 2.4.0, siendo ésta la última versión oficial publicada en el momento de realizar las experiencias. Se utilizó el mismo compilador tanto en TORO como en Windows/Linux, descartando posibles diferencias a nivel compilador, que podrían perjudicar la comparación. Fue utilizado el mismo lenguaje que para escribir el núcleo.

En entornos Windows/Linux se utilizaron las librerías abiertas MTProcs [8]. Estas librerías permiten la manipulación de hilos en ambientes Windows/Linux, facilitando su creación y migración. En TORO se utilizaron las llamadas al sistema que éste brinda para la manipulación y creación de hilos en entornos multicore. El algoritmo utilizado para ordenar cada grupo es el mismo que en la implementación Windows/Linux.

El número de hilos utilizados fue fijado igual al número de procesadores. A cada hilo se le asignó una cantidad de grupos a ordenar. En las experiencias se fue incrementando el número de grupos y observando el comportamiento del sistema.

La aplicación fue ejecutada en un sistema AMD Turion64 x2 de 2.8 GHz Las pruebas se hicieron sobre los SO de 64 bits: Ubuntu, Windows 2003 Server y TORO. La versión de Ubuntu utilizada fue la 9.10, con el kernel Linux versión 2.6.31.

D. Mediciones sobre sistema monoprocesador

Esta prueba se realizó utilizando un único procesador, con un único hilo. Se incrementó el número de grupos involucrados hasta 1000, midiéndose el número de ciclos de reloj consumidos.

El número de elementos a ordenar se mantuvo constante. Se realizaron 4 series de medidas por cada SO y luego se promedió el valor para cada grupo. Los resultados que se obtuvieron en los SO antes mencionados se presentan en la Figura 1, donde se puede observar que el número de ciclos de reloj consumidos en los tres sistemas es similar. Los tres siguen aproximadamente la misma curva. Se deduce que no se

obtienen mejoras al correr esta aplicación en SO generales con respecto al núcleo dedicado.

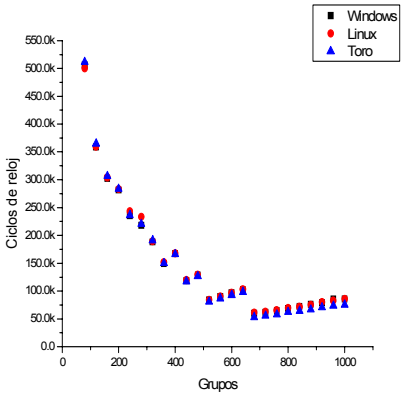

Figura 1. Prueba sobre sistema monoprocesador. Los valores corresponden a promedios en un grupo dado.

El hecho de utilizar un único procesador hace que la aplicación se comporte de manera similar en todos los sistemas. No se observan diferencias notables debido a los distintos algoritmos de planificación que utiliza cada sistema.

E. Mediciones sobre un sistema con procesador dual

Esta prueba se realizó utilizando dos procesadores, por lo tanto se utilizaron dos hilos. Se fue incrementando el número de grupos involucrados hasta 1000, midiéndose el número de ciclos de reloj consumidos.

El número de elementos a ordenar se mantuvo constante. Del mismo modo que en la prueba monoprocesador, se realizaron 4 series de medidas por cada SO y luego se promediaron los resultados para cada grupo. Debido a la dispersión hallada al realizar las pruebas, se debió estimar un error a partir de la resta del valor máximo obtenido para un grupo dado menos la media para ese mismo grupo.

Los resultados que se obtuvieron en los SO antes mencionados se presentan en la Figura 2.

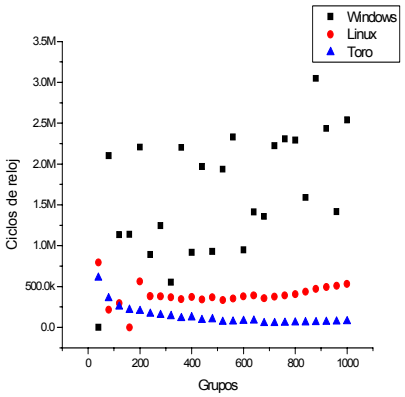

Figura 2. Comparación entre los distintos SO en la experiencia con procesador dual.

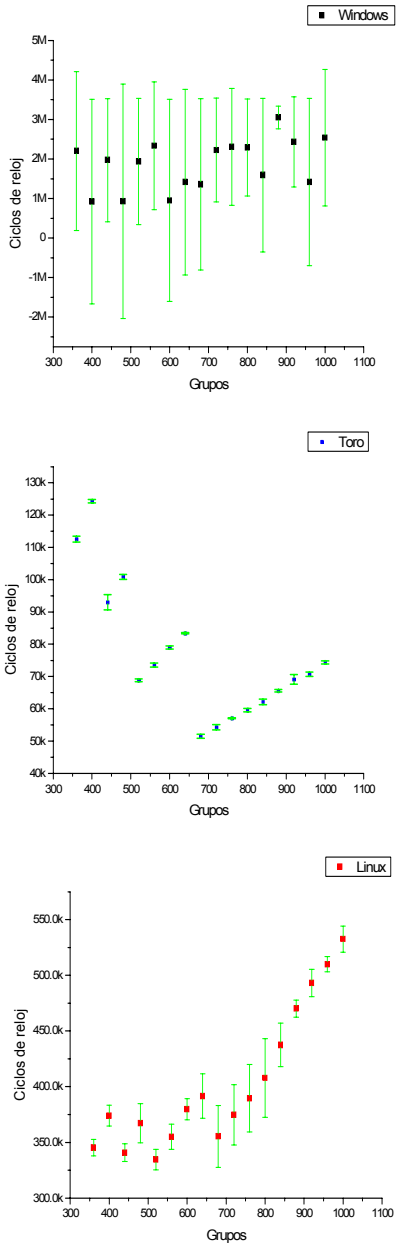

Figura 3. Pruebas sobre sistema dual. Las barras de error corresponden a la dispersión observada en las experiencias.

En la Figura 3 se observa la gran diferencia encontrada entre los tres sistemas, si bien el comportamiento fue asintótico como se esperaba.

F. Interpretación de los resultados

En el caso de Windows se observa una gran dispersión de los resultados. Suponemos que esto se debe a que en el

momento de realizar las pruebas hay servicios del núcleo corriendo por detrás y el planificador de Windows decidió darle más prioridad a los servicios que a la aplicación. Resultó muy difícil la toma de valores sobre Windows, debiéndose realizar varias veces la experiencia por la gran variación de los resultados. El valor de K obtenido para Windows fue de 2755±1334.98. La gran dispersión hizo imposible lograr una estimación correcta, de modo que el valor obtenido del ajuste solo puede ser utilizado como una posible cota.

Por otro lado, en el entorno Linux los valores medidos fueron mucho más constantes. La aplicación se comportó de manera más predecible que en el entorno Windows. El valor de K obtenido fue de 576±19.94.

En el caso de TORO, el comportamiento es muy similar al obtenido en el sistema monoprocesador. Los resultados fueron constantes en las pruebas realizadas, esto se deduce de los pequeños valores de error obtenidos. Se obtuvo un valor de K igual a 71±1.39, similar a la experiencia realizada en el sistema monoprocesador.

IV. CONCLUSIONES

Cada vez es más común la utilización de sistemas dedicados a una única actividad, como por ejemplo servidores web, ruteadores, motores de cálculo numérico, etc. En estos ambientes donde la CPU es dedicada, un SO de uso general desperdicia los recursos disponibles. Vimos que si se toma como hipótesis que la CPU es dedicada, el kernel puede ser simplificado, reduciendo complejidad y disminuyendo latencias.

Con los supuestos de un sistema multicore y una actividad dedicada surge TORO. Los conceptos de diseño se basaron en simplificaciones impuestas al dedicar una CPU a una única tarea. Por el momento el kernel está apto para realizar tareas sencillas, pero se necesita mucho tiempo de desarrollo todavía para aplicarlo a entornos más exigentes como Internet.

El algoritmo MergeSort resultó ser un buen ejemplo de un algoritmo paralelo de fácil implementación. La utilización de la librería MTProcs facilitó el trabajo de la escritura de la aplicación, independizándonos del SO. Por otro lado la utilización de esta librería oculta la manera en que el SO crea y migra hilos entre procesadores, reduciendo nuestra capacidad de optimizar los procedimientos implementados.

En la implementación sobre TORO la escritura de la aplicación resulto ser más compleja debido a que hubo que interiorizarse con las llamadas al sistema del núcleo dedicado. Pero ésto permitió programar el algoritmo de forma óptima, aprovechando eficientemente los recursos.

Al realizar las experiencias se observó claramente que el comportamiento de la aplicación al pasar de un procesador a dos no fue lineal, es decir, que los tiempos no se dividieron por dos. Al contrario, se observó que los tiempos en ciertos casos del sistema dual fueron superiores al sistema monoprocesador, independiente del SO sobre el que se corriera la aplicación.

La escritura de aplicaciones sobre el núcleo dedicado supone una situación de compromiso para el programador entre portabilidad y optimización. En general las aplicaciones escritas en TORO no son portables a otros SO pero suponen una utilización óptima de los recursos. De esta manera, aunque el proceso de escribir la aplicación sobre TORO puede resultar un poco más engorroso, vale la pena si a cambio se obtiene un incremento en la performance.

Como trabajo futuro queda continuar el desarrollo del kernel para lograr aplicarlo a entornos que exigen cada vez más al hardware; desarrollando drivers y nuevos módulos e incrementado la aplicaciones sobre TORO con especial énfasis en el área de control. Por otro lado, continuar realizando experiencias del algoritmo MergeSort en sistemas con más procesadores de forma de comparar mejor el kernel dedicado con respecto a los SO generales.

REFERENCIAS

[1] Stallings W., Sistemas Operativos, 2ed. Prentice Hall, Madrid, 1997

[2] Freepascal. http://www.freepascal.org.

[3] Joyanes Aguilar, Luis. Programación en Turbo Pascal, 2 ed.

[4] Intel. IA-32 Intel® Architecture Software Developer's Manual. Vol3. 2004

[5] Paula McKenney: RCU vs. Locking Performance on Different Types of CPUs, 2005

[6] AMD. Software optimization guide for AMD Family 10h Processors, 2008

[7] Intel Corporation. QuickPath architecture white paper, 2008

[8] Freepascal Wiki. http://wiki.freepascal.org/Parallel_procedures.