Un Modelo de Threads Reactivos Usuario-Kernel para Aplicaciones Internet de Alto Desempeño

Carlos Pineda G.
Centro de Investigación en Computación
Instituto Politécnico Nacional
Unidad Profesional Adolfo López Mateos
México DF 07738
carlospinedag@hotmail.com

Hugo C. Coyote
Programa de Investigación en Matemáticas Aplicadas y Computación
Instituto Mexicano del Petróleo, México
Eje Central Norte Lázaro Cárdenas 152
hcoyote@hotmail.com

Jordi García A.
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Jordi Girona 1-3 C6-E207 Campus Nord
Barcelona 08034, España
Fax: (+34) 93 4017055
jordig@ac.upc.es

Resumen

En este trabajo se presenta un modelo de threads usuario-kernel de propósito general, eficiente, altamente escalable, y orientado a la construcción de aplicaciones Internet de alto desempeño que requieren crear cientos de threads y mantener igual número de conexiones simultáneas (p.e. servidores Web y servidores proxy). El modelo incluye un mecanismo de planificación reactiva que permite gestionar en forma implícita operaciones de I/O no bloqueantes sobre threads de usuario cooperativos.

Palabras clave: User-Kernel Threads, Reactive Threads, MxN Threads, Reactive Scheduling, ukthreads, Internet Application.

1 Introducción

1.1 Aplicaciones Internet

Una *aplicación Internet* (AI) es un cliente, un servidor, o un cliente-servidor, que acepta conexiones de otros clientes y puede conectarse a otros servidores. Ejemplos de aplicaciones Internet son los servidores Web y los servidores proxy. Una AI se caracteriza por el número de transacciones por segundo que puede realizar (*throughput*), su escalabilidad ante el incremento de la carga (*load scalability*), y su escalabilidad ante el incremento de los recursos del sistema (*system scalability*). El throughput de una AI crece linealmente con respecto al número de conexiones hasta alcanzar un máximo. La escalabilidad de carga es una medida del número de clientes adicionales que puede soportar la aplicación manteniendo el throughput en una vecindad de su valor máximo. Evidentemente los recursos del sistema que permiten a una aplicación aceptar y procesar conexiones son limitados, por lo que a partir de un número determinado de conexiones el throughput comenzará a disminuir. La escalabilidad del sistema

es una medida de qué tanto aumenta el throughput máximo de la aplicación cuando se incrementa los recursos (número de CPUs, tamaño de la memoria, etc). El número de conexiones simultáneas que una AI puede soportar es denominada *concurrencia virtual*. Los recursos que utiliza la aplicación es la *concurrencia real* o paralelismo del sistema. A las entidades de ejecución que el sistema operativo planifica se les denomina *vehículos de ejecución* o threads.

El funcionamiento de una AI típica incluye las siguientes operaciones: aceptar una conexión, leer un requerimiento, ejecutar una tarea, y enviar el resultado al cliente que originó el requerimiento. Para cada conexión la aplicación debe crear un vehículo de ejecución, lo cual implica que el throughput y la escalabilidad de la aplicación dependen del número de threads que puede crear y mantener el sistema. Actualmente en Linux el throughput y la escalabilidad de una AI se ven limitados principalmente por la incapacidad del sistema operativo para crear un gran número de threads [1].

En este artículo se presenta el diseño de un modelo de threads para Linux que permite crear miles de threads de usuario sobre uno o más threads de kernel, bajo un esquema de planificación reactiva que integra el manejo de operaciones no bloqueantes de I/O en forma implícita. La combinación del modelo de threads usuario-kernel con los mecanismos para la gestión de operaciones I/O no bloqueantes permite construir aplicaciones Internet con la capacidad de crear cientos de threads que se ejecuten desacoplados de las operaciones de I/O lo cual maximiza el throughput y la escalabilidad de la aplicación.

1.2 Modelos de threads

Actualmente existen tres modelos de threads [4], el primero es conocido como modelo Mx1o modelo de threads de usuario, donde es posible iniciar M co-rutinas que se ejecutan en el espacio de usuario dentro de un solo proceso. La principal ventaja de este modelo es que los cambios de contexto se realizan de manera muy eficiente en el espacio de usuario. Sin embargo, y debido a que los threads ejecutan en un solo proceso no es posible aprovechar el paralelismo de un sistema multiprocesador [6][11].

El segundo modelo de threads es conocido como modelo 1x1 o modelo de threads de kernel, donde cada thread de usuario corresponde a un thread de kernel. Este modelo es el que implementa actualmente Linux en su biblioteca de threads POSIX, y tiene la ventaja de que permite aprovechar el paralelismo de un sistema multiprocesador, pero tiene la desventaja de que cada thread corresponde a un proceso Linux, lo cual restringe el número de threads que el sistema puede crear y mantener en ejecución. El número máximo de pthreads que se puede crear sobre un procesador Intel-32 es 8192 (menos 1 para el *manager*), sin embargo la creación de cientos de threads degrada el acceso al file system /proc, ya que cada thread aparece como un proceso separado.

El tercer modelo de threads conocido como MxN, integra las ventajas de los modelos anteriores ya que permite iniciar miles de threads de usuario sobre uno o más threads de kernel. En este modelo los cambios de contexto se realizan en forma muy eficiente en el espacio de usuario, al mismo tiempo es posible aprovechar el paralelismo de un sistema multiprocesador. Actualmente existen implantaciones comerciales del modelo de threads MxN en las plataformas Sun-Solaris, SGI-IRIX, y DEC-Tru64.

2 Trabajos relacionados

Alpern *et al.* del T. J. Watson Research Center de IBM, desarrollaron un sistema de threads MxN quasi-preventivos para Jalapeño [5], una máquina virtual para servidores Java construida por IBM. Jalapeño mapea los threads Java sobre procesadores virtuales que son implementados

como pthreads de AIX. Para cada procesador físico se establece un procesador virtual. Es posible utilizar procesadores virtuales adicionales para gestionar la latencia de operaciones I/O. En Jalapeño los threads son cooperativos y sólo pueden ceder el control en puntos predefinidos por el compilador (*yield points*), lo cual es requerido para el funcionamiento del *garbage collector* de la máquina virtual. Los locks se implementan a cuatro niveles: *processor* locks, *thin* locks, *thick* locks, y monitores.

Welsh *et al.* de la Universidad de California [3], diseñaron una arquitectura que permite construir aplicaciones Internet de alto desempeño mediante etapas dirigidas por eventos interconectadas por colas (*SEDA: staged event-driven architecture*). SEDA usa un conjunto de controladores dinámicos para condicionar la carga en las etapas bajo un régimen de grandes fluctuaciones de carga. Utiliza una biblioteca de clases para Java denominada NBIO (*non-blocking* I/O) que permite implementar operaciones de I/O no bloqueante mediante la llamada a sistema `poll`.

SGI desarrolló una biblioteca conocida como State Threads [2] derivada de la biblioteca Netscape Portable Runtime library (NSPR), que permite escribir aplicaciones para Internet eficientes y altamente escalables. State Threads combina el modelo de threads cooperativos Mx1 con una arquitectura de máquina de estados dirigida por eventos (*event-driven state machine*), sin embargo no se trata de una biblioteca multithread de propósito general sino orientada específicamente al tipo de aplicaciones Cliente/Servidor de Internet. Como parte de este proyecto, SGI desarrolló una versión del servidor Web apache utilizando State Threads. En la sección 5.4 presentamos algunos resultados experimentales que obtuvimos dónde es posible observar que la versión de apache implementada con State Threads tiene un mejor desempeño que la versión original desarrollada por la ASF (Apache Software Foundation).

3 El modelo de threads reactivos

El modelo de los threads reactivos es MxN: M threads de usuario que ejecutan sobre N threads de kernel. Los threads de kernel son creados mediante la llamada a sistema de Linux *_clone*. Cada clone es un proceso con su propio PID y área de stack, sin embargo, a diferencia de los procesos creados con la llamada a sistema *fork*, los clones pueden compartir el espacio de datos y las tablas de signals. El mecanismo de planificación de los clones es preventivo basado en *time slice*, y tiene la capacidad de distribuir los procesos sobre los procesadores existentes lo cual permite aprovechar el paralelismo en sistemas multiprocesador.

3.1 Procesadores virtuales

Para cada thread de kernel se construye una estructura que llamamos *procesador virtual de tipo CPU* (cpu_vp). Cada cpu_vp cuenta con cuatro colas de threads: la cola de ejecución, la cola de inicio, la cola de entrada, y la cola *garbage*. La cola de ejecución es una cola circular doblemente enlazada que contiene los threads listos para ser ejecutados. La cola de inicio es una cola lineal doblemente enlazada que contiene los threads creados cuyo contexto no ha sido todavía inicializado (stack, stack pointer, etc). La cola de entrada es una cola lineal doblemente enlazada que sirve como interfaz de entrada para threads ya inicializados, los cuales han sido enviados desde otro procesador virtual (ver sección 3.7). La cola *garbage* es una cola lineal simple que contiene los threads terminados y listos para ser eliminados.

3.2 Creación de threads reactivos

Un thread reactivo puede ser creado en estado *running* o en estado *suspended* mediante la función *uk_createThread* del API. El thread es colocado en la cola de inicio solo si es creado en estado running. Un thread creado en estado suspended puede ser colocado en la cola de inicio posteriormente utilizando la función *uk_resume*.

En la cola de ejecución existe un thread cooperativo permanente el cual realiza la función de despachador de threads. Con el fin de optimizar los cambios de contexto, el acceso a la cola de ejecución lo realiza exclusivamente el thread despachador, por lo que esta cola no requiere sincronización. El acceso a las colas de inicio y de entrada requiere ser sincronizado mediante locks, ya que diferentes procesadores virtuales pueden tratar de insertar simultáneamente threads en dichas colas.

Cuando el thread despachador recibe el control, inicializa el contexto de los threads que están en la cola de inicio. Posteriormente los threads inicializados son transferidos al final la cola de entrada. Entonces los threads de la cola de entrada son transferidos a la cola de ejecución, y el thread despachador cambia al contexto del siguiente thread en la cola de ejecución, o bien, entra en un estado de espera pasiva si no hay más threads (en la cola de ejecución, en la cola de inicio, o en la cola de entrada).

Es posible crear dos tipos de threads: *detach* y *join*. Un thread de tipo *detach* libera automáticamente la memoria que ocupa cuando termina, utilizando para tal efecto, la cola *garbage* del cpu_vp donde ejecuta. Por otro lado, cuando termina un thread de tipo *join*, no libera por sí mismo la memoria que ocupa, en su lugar, notifica que ha terminado y es algún otro thread que espera la terminación del primero, el encargado de liberar la memoria correspondiente.

Con el fin de optimizar la creación de threads, los stacks de los threads se adquieren de un *pool* de bloques de memoria pre-alojados al inicializar la biblioteca. Cada vez que un thread termina, el bloque de memoria correspondiente es nuevamente enlazado al pool, de tal manera que queda disponible para la creación de otro thread. Si no hay más bloques disponibles en el pool de threads, la función *uk_createThread* ejecuta *uk_yield* hasta que otro thread termine. El tamaño de cada bloque en el pool es una constante potencia de dos y múltiplo del tamaño de página del sistema.

3.3 Inicialización del contexto

La inicialización del contexto de un thread consiste en establecer la dirección para el inicio de la ejecución, el contenido inicial del stack, y los registros *stack pointer* y *base pointer*. Debido a que actualmente los threads reactivos no soportan la expansión dinámica del stack, con el propósito de contar con un mecanismo de protección en caso de desbordamiento del stack, la primera página del stack se protege de la lectura y escritura mediante la llamada a sistema *mprotect*, de tal manera que si el stack llegara a desbordarse entonces se produciría una señal SIGSEGV. En el microprocesador Pentium el stack crece desde las posiciones altas a las posiciones bajas de memoria, de tal manera que el desbordamiento del stack se detecta cuando el procesador trata de escribir las posiciones más altas del stack. La figura 1 muestra el contenido del stack inicial de un thread reactivo.

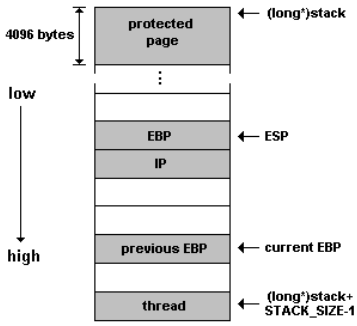

Fig 1: Stack inicial de un thread reactivo

3.4 Cambio de contexto

El cambio de contexto es realizado por la función *uk_switch*. Al hacer un cambio de contexto no es necesario comprobar que la cola de ejecución esté vacía debido a que por lo menos existe el thread despachador el cual fue creado al momento de iniciar el cpu_vp y permanece en ejecución siempre. Tampoco es necesario utilizar un lock para sincronizar el acceso a la cola de ejecución, ya que los nuevos threads creados y aquellos que son enviados por otros procesadores virtuales son colocados en las colas de inicio y de entrada, respectivamente (las cuales sí se sincronizan mediante locks). Sólo el thread despachador puede transferir los threads de las colas de inicio y de entrada a la cola de ejecución, lo cual se hace en el momento que el thread despachador recibe el control. Este mecanismo permite ejecutar primero los nuevos threads y después los threads enviados por otros procesadores virtuales, esto es: después de que se inicializa el contexto de los threads de la cola de inicio, estos son insertados al principio de la cola de entrada, la que a su vez es insertada en la cola de ejecución, entre el thread despachador y el primer thread reactivo (si existe alguno). Tampoco se requiere sincronizar la operación de extracción de un thread de la cola de ejecución, ya que sólo el thread puede extraerse a sí mismo de la cola.

3.5 Área local de almacenamiento

En un ambiente multithread hay variables que son sintacticamente globales al nivel de proceso pero semánticamente locales al nivel de thread (con un valor por cada thread). Este es el caso de la variable *errno* usada por libc con el propósito de indicar el número de error que eventualmente se produzca. Para obtener la dirección de la variable errno en el área local de almacenamiento (*local storage*) del thread actual, el API de los threads reactivos incluye la implementación de la función *int *__errno_location ()*.

3.6 Sincronización

Los threads reactivos implementan mecanismos de sincronización al nivel de cpu_vp y al nivel de thread de usuario. Al nivel de cpu_vp se implementa *spin locks* y locks FIFO. Un spin lock es un entero de 32 bits igual a 1 si se encuentra bloqueado o igual a 0 si se encuentra desbloqueado. La adquisición de un spin lock es en base a un ciclo de espera activa (*busy-wait*). Por otra parte, un lock FIFO cuenta con una cola de espera FIFO que permite asegurar el que no se presente una situación de *starvation* en los cpu_vp. La implementación de los spin locks y los locks FIFO se hace mediante una instrucción de intercambio atómico, en particular para el Pentium se utiliza la instrucción de intercambio a 32 bits *xchg m32,r32* la cual intercambia el registro r32 y la memoria m32 en forma atómica.

La sincronización al nivel de threads de usuario se realiza mediante mutex recursivos [10], implementados mediante una instrucción del tipo “comparar e intercambiar” en forma atómica. En particular para el Pentium se utiliza la instrucción de comparación e intercambio a 32 bits *lock cmpxchg m32,r32* la cual compara el contenido de la memoria *m32* con el contenido del registro EAX, si son iguales entonces el contenido del registro *r32* se escribe a la memoria *m32*, de lo contrario el registro EAX recibe el contenido de la memoria *m32*.

3.7 Planificación reactiva

Cuando un thread ejecuta una operación bloqueante éste puede ser transferido a otro procesador virtual que denominamos *procesador virtual reactivo*, con el fin de liberar al cpu_vp de cambios de contexto superfluos en la cola de ejecución. Los procesadores virtuales reactivos se implementan como clones de Linux y funcionan como monitores de los threads que ejecutan operaciones bloqueantes.

Actualmente tenemos definidos tres tipos de procesadores virtuales reactivos, los de tipo *net_vp* que gestionan aquellos threads que ejecutan las funciones *recv*, *send* y *accept* no completadas, los de tipo *io_vp* que gestionan aquellos threads que ejecutan las funciones *read* y *write* no completadas, y los de tipo *wait_vp* que gestionan los threads que ejecutan la función *uk_wait*. Los procesadores virtuales net_vp, io_vp y wait_vp utilizan la llamada a sistema `poll` para entrar en un estado de espera pasiva hasta que el sistema operativo les notifique que una o varias operaciones bloqueantes han sido completadas. Entonces los procesadores virtuales reactivos insertan los threads que han completado las operaciones bloqueantes en las colas de entrada de los cpu_vp correspondientes.

Extraer un thread de la cola de ejecución representa un costo, por lo tanto sólo es conveniente extraer los threads cuando el tiempo de bloqueo es considerable. La heurística que utiliza un cpu_vp para decidir cuándo extraer un thread de la cola de ejecución es asociar una constante SPIN a cada operación bloqueante de acuerdo al algoritmo mostrado a continuación, dónde es posible observar que la heurística esta orientada a reducir el tamaño de la cola de ejecución cuando se ejecutan operaciones bloqueantes.

```
for (;;)
{
  spin = SPIN;
  while (spin > 0)
    if (operacion_bloqueante_completada)
      return;
    else
    {
      spin = spin – 1;
      uk_yield ();
    }
  uk_extractThreadRunningQueue;
  uk_insertThreadVpQueue;
  uk_yield ();
}
```

La figura 2 muestra el ciclo de vida de un thread reactivo. Primeramente el thread es creado tomando un bloque del pool de threads, este thread es insertado en la cola de inicio. Entonces el despachador del cpu_vp crea un contexto para el thread y lo envía a la cola de entrada. Después el despachador envía los threads existentes en la cola de entrada a la cola de ejecución. Si un thread ejecuta una operación de I/O bloqueante, el thread es enviado a un procesador virtual reactivo seleccionado mediante un mecanismo de *round robin*. Cuando la operación de I/O bloqueante es completada por el procesador reactivo, el thread correspondiente es enviado a la cola de entrada de un cpu_vp seleccionado también usando round robin . Si un thread termina entonces es enviado a la cola garbage. Cuando el thread despachador ejecuta, los threads en la cola garbage son transferidos al pool de threads.

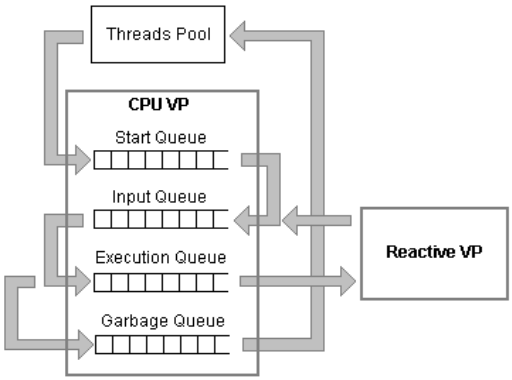

Fig. 2: Ciclo de vida de un thread reactivo

La figura 3 muestra un ejemplo de una aplicación en la que se ha definido dos cpu_vp, un cpu_vp por cada CPU físico. En este ejemplo se ha definido un io_vp para gestionar las operaciones sobre disco, y un net_vp para gestionar las operaciones sobre la red.

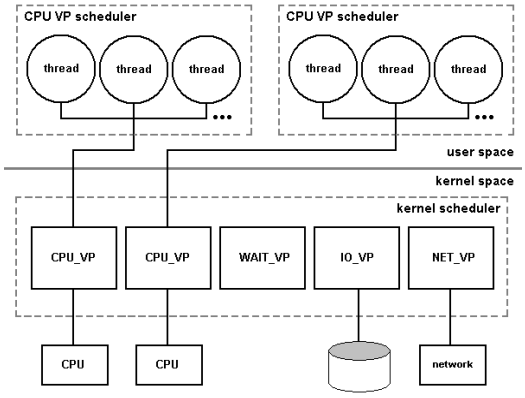

Fig 3: Una aplicación con dos procesadores virtuales cpu_vp.

3.8 Conjuntos de cpu_vp

Los threads reactivos son threads cooperativos que ejecutan cuándo se les otorga explícitamente el control, esto puede limitar la concurrencia en algunas aplicaciones. Sin embargo es posible construir aplicaciones que utilicen threads reactivos que sean cooperativos y al mismo tiempo preventivos por grupos. Considerando que los cpu_vp son implementados mediante threads de kernel preventivos, el API de los threads reactivos permite definir conjuntos de cpu_vp (*cpu_vp_set*) mediante la función *uk_createCpuVpSet*, y crear threads en determinados conjuntos de cpu_vp mediante la función *uk_createThread*. El conjunto *default_cpu_vp_set* es definido al momento de inicializar la biblioteca y contiene todos los cpu_vp creados. La función *uk_getNextCpuVp* es utilizada por *uk_createThread* para obtener, de un cpu_vp_set, el siguiente cpu_vp a seleccionar mediante round robin.

4 El API del los threads reactivos

El API de los threads reactivos incluye las funciones para la gestión de threads y sincronización. Se implementó las funciones `accept`, `read`, `recv`, `send`, y `write` para su funcionamiento no bloqueante junto con el mecanismo de planificación reactiva descrito en la sección 3.7. Se escribió funciones puente para `calloc`, `free`, `malloc`, `memalign`, y `realloc`, con el propósito de hacerlas concurrentemente seguras (*thread-safe*), utilizando un spin lock global para sincronizar los accesos a estas funciones. Una solución más escalable será rescribir completamente estas funciones considerando incluir un *heap* privado para cada cpu_vp, lo cual aumentará la concurrencia de las operaciones de alojamiento de memoria [8]. También se rescribieron las funciones de stdio con el propósito de hacerlas concurrentemente seguras, sincronizando el acceso a los descriptores de archivo mediante un vector global de spin locks.

5 Un servidor Web con conexiones persistentes

En esta sección se presenta la aplicación de los threads reactivos en la construcción de un servidor Web con conexiones persistentes compatible con el protocolo HTTP/1.1 [7]. Se construyó una versión del Web utilizando pthreads, y dos versiones utilizando threads reactivos.

Se usó httperf [9] para medir el rendimiento de los servidores construidos y se comparó con el rendimiento del servidor apache en sus versiones 1.3, 2.0, y apache 1.3 implementado mediante State Threads (ver sección 2).

5.1 Servidor Web implementado con pthreads

En la sección 1.1 se explicó que las operaciones básicas de un servidor son: aceptar una conexión, leer un requerimiento, ejecutar una tarea, y enviar el resultado al cliente. En un servidor Web que utiliza pthreads, para cada conexión que establece la función `accept` se crea un pthread que lee el requerimiento mediante `recv` y envía el resultado al cliente utilizando `send`. El número de conexiones simultáneas que puede establecer este servidor se ve limitado por el número de pthreads que puede crear Linux. Debido a que las funciones `accept`, `recv` y `send` se ejecutan en modo bloqueante, el pthread se bloquea en el kernel hasta completar las operaciones de I/O. Para gestionar las tareas que realizan operaciones de I/O bloqueantes Linux utiliza colas de espera (*wait queues*), sin embargo, el número total de pthreads que puede crear y mantener en ejecución es muy limitado. Una solución a este problema sería rescribir el servidor utilizando las funciones `recv` y `send` en modo no-bloqueante y compartir un pthread entre diferentes conexiones. Esto resulta muy complicado.

5.2 Servidor Web implementado con threads reactivos

Ahora considérese el mismo servidor implementado mediante threads reactivos. Para cada conexión que establece `accept` el servidor crea un thread cooperativo que lee el requerimiento mediante `recv` y envía el resultado al cliente utilizando `send`, pero ahora las funciones `accept`, `recv` y `send` son implementadas en el API de los threads reactivos y operan implícitamente en modo no-bloqueante (ver sección 3.7), de tal manera que si las operaciones de I/O no son completadas sólo se bloquea el thread reactivo y no el thread de kernel (cpu_vp). Esto permite a la aplicación gestionar un gran número de conexiones mediante un número igual de threads reactivos, los cuales ejecutan sobre un reducido número de cpu_vp , en este caso un cpu_vp por cada CPU físico. Debido a que el número máximo de descriptores de archivo que puede gestionar Linux es 1024, el número de threads reactivos concurrentes se limitó a 800, ya que cada thread utiliza dos descriptores de archivo (uno para el socket y otro para el archivo solicitado por el método GET de http).

5.3 Servidor Web implementado con threads reactivos y conjuntos de cpu_vp

Al medir el desempeño del servidor implementado con threads reactivos y compararlo con el desempeño de apache, pudimos observar que apache tiene menor desempeño cuando se ejecuta en una computadora con un procesador, pero un mejor desempeño cuando se ejecuta en una computadora con dos procesadores.

En el servidor implementado con threads reactivos el thread principal que espera conexiones y ejecuta por omisión en el cpu_vp 0, cede el control a los threads que él mismo crea, lo cual limita la tasa de conexiones que puede aceptar este servidor. La solución a este problema es permitir que el thread principal se comporte en forma preventiva, es decir, que ejecute en forma exclusiva en el cpu_vp 0. Se construyó entonces otra versión del servidor con tres cpu_vp, de tal manera que el thread principal ejecute exclusivamente en el cpu_vp 0 y los otros threads ejecuten en los cpu_vp 1 y 2. En este caso se utilizó la función bloqueante `__libc_accept` en lugar de la función no-bloqueante `accept` del API, con el propósito de que el cpu_vp 0 se bloquee en el kernel en espera de conexiones, lo cual resulta más eficiente que el extraer el thread de la cola de ejecución (ver sección 3.7).

5.4 Resultados experimentales

Se utilizó httperf para medir el desempeño de los servidores construidos y compararlo con el desempeño de apache en sus versiones 1.3 y 2.0. También se midió el desempeño de apache 1.3 implementado mediante State Threads. Los experimentos se realizaron sobre una red de computadoras con Pentium III a 650 Mhz y Linux 2.2.14-5.0smp conectadas mediante un switch Emulex cLAN con un ancho de banda de 1.25 Gb/s full duplex. Cada experimento consistió en 3000 conexiones y 5 requerimientos por conexión. Cada requerimiento solicitó un archivo de 10,000 bytes. Las figuras 4 y 5 muestran el throughput de los servidores en términos de tráfico en la red (Net I/O) y respuestas por segundo, correspondientes a una tasa de 10 a 120 conexiones por segundo (50 a 600 requerimientos por segundo). Las gráficas web_uk corresponden al servidor web implementado con threads reactivos, y las gráficas web_uk_set corresponden a la versión del servidor web que usa threads reactivos agrupados en conjuntos de cpu_vp. Las gráficas web_pth corresponden al servidor web implementado con pthreads. Las gráficas apache 1.3-st muestran el rendimiento de apache versión 1.3 implementado mediante State Threads .

6 Conclusiones

El modelo de threads usuario-kernel cuenta con un mecanismo de planificación reactiva eficiente y altamente escalable, orientado a la construcción de aplicaciones Internet de alto desempeño que requieren crear cientos de threads y mantener un número igual de conexiones simultáneas. Los resultados que hemos obtenido nos hacen ver las enormes posibilidades que tiene esta implementación de threads MxN, sobre todo en aplicaciones que demandan un alto desempeño y la necesidad de crear un gran número de threads al nivel de usuario que realizan operaciones de I/O. Actualmente estamos implementando esta biblioteca de threads en una máquina virtual de Java que hemos construido. Como trabajo futuro integraremos la gestión de señales de Linux al nivel thread de usuario.

Referencias

[1] ULRICH DREPPER, INGO MOLNAR: *The Native POSIX Thread Library for Linux*; Red Hat Inc., January 2003.

[2] SILICON GRAPHICS Inc: *State Threads for Internet Applications*, December 2002, http://state-threads.sourceforge.net/

[3] MATT WELSH, *et al.*: *SEDA: An Architecture for Well-Conditioned Scalable Internet Services*; Computer Science Division, University of California, Berkeley, 2001.

[4] LIAM WIDDOWSON: *An Introduction to Multi Threaded Programming with POSIX Threads and Linux*; Hewlett-Packard Consulting, 2001.

[5] B. ALPEM, C.R. ATTANASIO, *et al.*: *The Jalapeño virtual machine*, IBM Systems Journal, Vol 39, No 1, 2000.

[6] RALF S. ENGELSCHALL: *Portable Multithreading: The Signal Stack Trick For User-Space Thread Creation*; USENIX Annual Technical Conference, San Diego California USA, 2000.

[7] R. FIELDING, *et al.*: *Hypertext Transfer Protocol HTTP/1.1*; Network Working Group, Internet RFC 2616, June 1999.

[8] CHUCK LEVER, DAVID BOREHAM*: malloc Performance in a Multithreaded Linux Environment*; Netscape Communications Corp, 1999.

[9] DAVID MOSBERGER, TAI JIN: *httperf: A Tool for Measuring Web Server Performance.* Hewlett-Packard Research Labs, 1998.

[10] C. J. NORTHRUP: *Programming with UNIX Threads*; John Willey & Sons, 1996.

[11] CHRIS PROVENZANO: *MIT pthreads*, 1993.

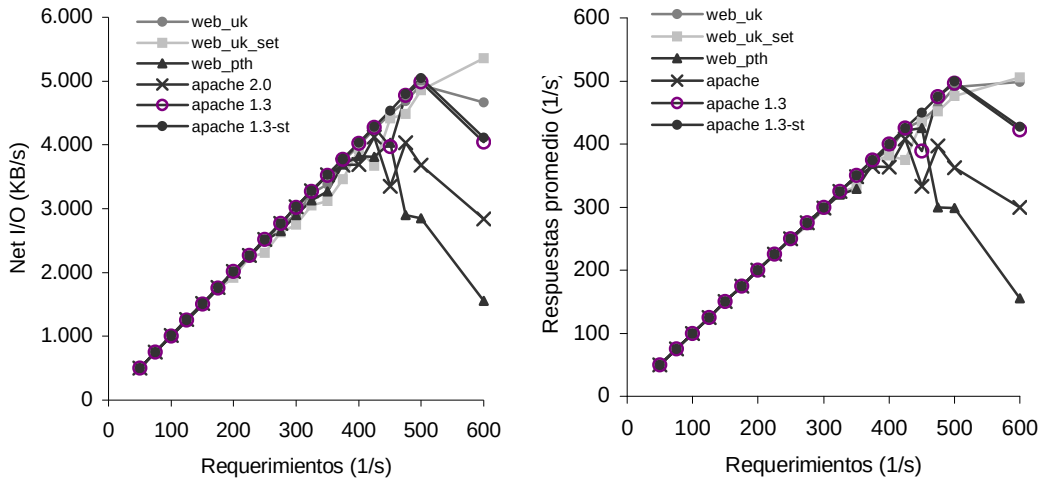

Fig 4: Desempeño de los servidores Web cuando se ejecutan en una computadora con un procesador

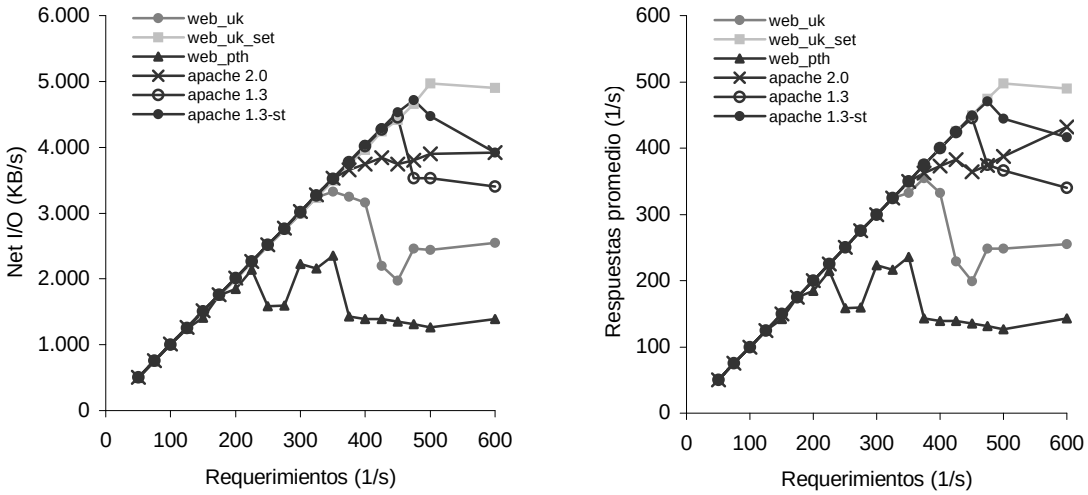

Fig 5: Desempeño de los servidores Web cuando se ejecutan en una computadora con dos procesadores