Parte III

Evaluación, conclusiones y líneas abiertas

7

Evaluación del rendimiento en la cooperación kernel-aplicación

“To err is human, to really foul things up requires a computer”

()

ABSTRACT: Todo proyecto de investigación que lleve inmerso desarrollo de software, acaba inequívocamente en una cuestión de comparación con lo ya existente: en qué es mejor esta aplicación que sus predecesoras, en cuánto, para qué se ha hecho ese esfuerzo innovador, qué se aporta al mundo de la ciencia y la tecnología.

A partir de microbenchmarks evaluamos el aumento de rendimiento en la ejecución de las aplicaciones paralelas al llevar ellas la gestión que hasta ahora llevaba el sistema de planificación de flujos. También al ofrecer mecanismos de planificación más de acuerdo a las necesidades de la aplicación y conocimiento de sus flujos (hints, handoffs) en lugar de las políticas más generalistas que se utilizan (prioridades, round-robin).

Comentamos las herramientas utilizadas para las mediciones, tanto software, como hardware. Algunas gráficas y tablas, nos dan idea del funcionamiento de nuestro entorno frente al ofrecido originalmente.

A partir de los resultados, concluimos en lo positivo de potenciar una cooperación cada vez mayor entre la aplicación y el kernel en la gestión de los recursos.

Capítulo 7: Evaluación del rendimiento en la cooperación kernel-aplicación

7.1. Introducción

En este capítulo vamos a evaluar el rendimiento que pueden obtener las aplicaciones trabajando en el entorno eXc que hemos desarrollado.

Explicaremos los métodos y herramientas de medición utilizados, tanto software como hardware. Algunos de ellos nos los ofrecía ya el sistema, otros los hemos tenido que desarrollar. Asimismo comentaremos los benchmarks y microbenchmarks que hemos utilizado para obtener los valores que buscábamos.

En primer lugar, justificaremos el traslado a nivel de usuario de funciones que tradicionalmente residen en el sistema. Mostramos para ello el coste de creación y gestión de objetos en cada uno de los dos niveles.

También la planificación tiene una mayor sobrecarga en el sistema, tanto en cuanto a los algoritmos empleados como en cuanto al "peso" asociado a los objetos. Esto lleva a un coste superior en los cambios de contexto, hasta de un orden de magnitud de diferencia.

Daremos también los tiempos empleados en las diferentes partes de ejecución de upcalls, desde su reconocimiento en un momento de *scheduler activations*, hasta la puesta en marcha de un flujo de la aplicación por parte del flujo nulo correspondiente. Estas medidas son variables dependiendo de la upcall correspondiente.

A partir de un microbenchmark de un servicio con bloqueo en el sistema, hacemos una comparación del rendimiento que se obtiene en el entorno tradicional de Mach, y en un entorno de eXc. En este último caso hemos evaluado la diferencia de tiempos cuando la gestión del evento la lleva el sistema y cuando la trata el usuario.

7.2. Caracterización de la carga, benchmark

Actualmente se está dando un impulso, o por lo menos se fomenta, el desarrollo de la programación paralela. Desde el punto de vista del diseñador del sistema operativo, hay que identificar y aislar los puntos débiles en su diseño. También poder comparar el rendimiento y mejoras de nuevas realizaciones o servicios. El objetivo principal es reflejar la utilización del SO que se encuentra en los programas de usuario.

Para que las medidas que se van a tomar del sistema sean indicativas de su funcionamiento real, hay que poner al sistema en condiciones normales de trabajo y, además, éstas han de poder repetirse [HINN88]. El trabajar directamente sobre la carga real del sistema es costoso en tiempo y en recursos y es por ello que, generalmente, se dispone de cargas simuladas, consistentes en un programa, o conjunto de ellos, que mantengan ocupado al sistema con un trabajo equivalente a las situaciones reales del entorno en que esa máquina funciona, o del que pretendemos que soporte. Es lo que se conoce como benchmark [GIL91].

Hay, sin embargo, un gran desfase entre las aplicaciones que realmente están ejecutándose para resolver problemas, y las técnicas en programación paralela y explotación de arquitecturas paralelas (a nivel hardware y software).

Generalmente, los estudios que se han realizado de rendimiento son a nivel de microbenchmarks, repitiendo la ejecución de una única llamada o servicio. Este tipo de medidas es importante para ver la eficiencia de su realización, pero no llevan por sí solas a ninguna interpretación, en lo que concierne a los usuarios, sobre el rendimiento de las aplicaciones de alto nivel que corren en el sistema. Y es ahí, precisamente, donde la interacción de diferentes componentes es crucial y determinante.

Existen dos posibilidades de realizar programas de benchmarks:

1.- Escribir programas que sinteticen las propiedades que se requieren.

2.- Agrupar código real de usuario.

En el primer caso, se puede entender con detalle los servicios del sistema que se utilizan, lo que permite realizar benchmarks para probarlos individualmente y analizar en más profundidad su funcionamiento aislado. También facilita una parametrización en los benchmarks para que puedan adaptarse a diferentes entornos y sistemas. De este modo, con un conjunto de ellos, se puede cubrir el rango total de los servicios importantes y significativos del sistema.

Es importante asegurarse de que los programas de un benchmark sintético reflejan de manera representativa la utilización de los servicios del sistema de los programas de aplicación. Para ello se utilizan también traceos, perfiles del sistema y utilidades de contaje (vmstat, iostat, ms, sar, sa, entre otros).

Del código real de usuario, se extraen e identifican las llamadas al sistema clave de la aplicaciones que interesan. Junto a los datos extraídos estadísticamente se construyen benchmarks realmente representativos.

El problema principal de estas técnicas es que, si bien utilizan los mecanismos que se quieren medir, no siempre lo hacen en el entorno real; muchas veces, incluso explotan las situaciones al máximo, o sesgan la multiplicidad de factores a un mínimo o extremo inaceptable.

La mayor parte de los benchmarks son teniendo en cuenta un uso exhaustivo de CPU (CPU *bound jobs*), con lo que no son significativos de las funciones del SO. Seguramente, la realidad es más sencilla, pero también más rica.

7.3. Herramientas de monitorización software y hardware

Es importante contar con las herramientas adecuadas para registrar datos acumulativos sobre la demanda de utilización de los recursos del sistema y que, además, faciliten el análisis de problemas de rendimiento del sistema. Estas herramientas pueden ser tanto de tipo software como de tipo hardware e, incluso, una combinación de ambas.

Dicha instrumentación facilita la extracción de información sobre los ***cambios en el estado del sistema***, y su forma depende del tipo de información que se desee obtener de los componentes individuales y de su interacción. Por ejemplo, con cuánta frecuencia se realiza una determinada operación; es decir, con qué frecuencia se entra en un determinado estado: número de llamadas a un procedimiento concreto, frecuencia de los fallos de página, frecuencia de errores de paridad de la memoria...

Más complicada es la extracción del ***tiempo empleado*** en las actividades observadas, como puede ser el porcentaje, en media, que el sistema está en un estado concreto: utilización de la CPU o espera en una transferencia de los canales de E/S.

Interesa a veces, sin embargo, medir los ***tiempos de instancias individuales*** de una actividad específica -medir, cada vez que se entra en un estado específico, el tiempo que el sistema permanece en él-: distribución de los tiempos de servicio del procesador, tiempo transcurrido entre errores tales como un *crash* del cabezal del disco o un error de paridad en memoria.

En los tres casos expuestos se asume la posibilidad de detectar los cambios pertinentes en el estado del sistema. En el segundo caso, no obstante, no es necesario saber exactamente el momento en el que el sistema cambia de estado: una buena aproximación de la medida consiste en examinar el estado del sistema en momentos aleatorios de tiempo -aleatorios en el sentido en que no están relacionados con las actividades a medir- y el porcentaje de tiempo pasado en un estado determinado se calcula como la razón entre las observaciones en que el sistema estaba en el estado de interés y el número total de observaciones. Éste es el modo en que trabajan los medidores hard-

ware de tiempos y algunos software.

7.3.1. Mecanismos hardware

Los mecanismos hardware se basan en comparadores y contadores añadidos a la arquitectura de forma que detecten cambios en el estado del sistema y cuenten sus ocurrencias sin afectar por ello el funcionamiento del sistema, como ocurre con los mecanismos software, que introducen instrucciones de medida provocando una sobrecarga que ralentiza la ejecución global.

Generalmente, son aparatos prototipo, elaborados “ad hoc” para cada máquina en concreto, y pueden variar de muy sencillos -algún *led* que indique cuando se produce un cambio determinado-, a altamente complejos -conexiones con terminales gráficos que monitoricen los distintos niveles por los que pasa el sistema, extracción directa de estadísticos, etc- [LARR91].

7.3.2. Mecanismos software

Es importante que el sistema operativo nos proporcione herramientas de acumulación de datos para evaluar la utilización de sus propios recursos. Consisten en una serie de contadores, incrementados ante determinadas acciones del sistema, que abarcan una extensa gamas de datos de medida, incluyendo la utilización de la CPU por los programas, actividades de E/S de discos, cintas y terminales, uso de los *buffers*, llamadas al sistema, actividad sobre memoria (paginación, *swapping*), cambios de contexto, etc [BACH86] [CARR86].

El perfilamiento da una medida sobre cuánto tiempo el sistema ha estado ejecutando en modo usuario, en modo kernel, y cuánto tiempo ha empleado en ejecutar determinadas rutinas en el kernel.

Es importante hacer notar que estos mecanismos no tiene en cuenta el tiempo empleado en ejecutar el manejador del reloj ni el código que bloquea las interrupciones de nivel de reloj, ya que éste no puede interrumpir dichas regiones críticas de código y no puede, por tanto, llamar al manejador del “profile”. Ésto es una lástima porque estas regiones son, con frecuencia, las partes más importantes y necesarias a perfilar.

7.4. Entorno de trabajo

La máquina sobre la que se ha realizado el trabajo es un Digital DEC433MP -llamado Mestres- con cuatro procesadores i486 con el microkernel Mach 3.0 MK4.1 sobre el cual corre el subsistema OSF/1 1.1 como servidor del entorno operativo para las aplicaciones (Figura 7-1).

Hemos trabajado con CThreads, librería de threads desarrollada junto a Mach para la programación con múltiples flujos de control dentro de una task. Sus primitivas más importantes permiten la creación de flujos, la sincronización por *spin locks* y *mutex locks* y el bloqueo sobre variables de condición.

Resumen de las características principales del entorno de desarrollo:

Máquina: DEC433MP

Procesadores: 4 x i486 at 33Mhz

Memoria cache: 256 Kbytes por procesador

Memoria principal: 32 Mbytes

Bus: Corollary bus, write-back con protocolo de coherencia, transferencia a 64 Mbytes por segundo

Kernel: Mach 3.0 MK14.0.6

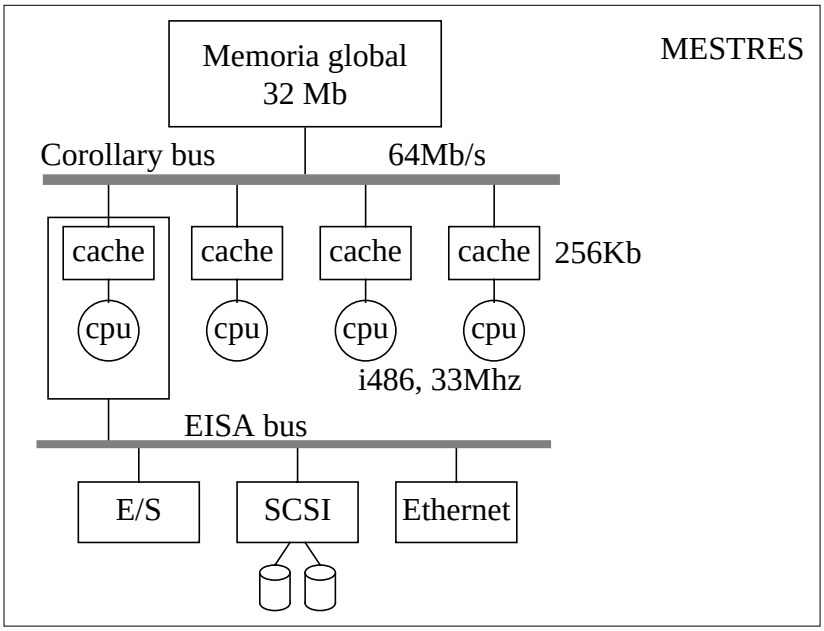

Figura 7-1 Arquitectura del DEC433MP

7.5. Hardware de medición: High Resolution Clock

En un multiprocesador, si no se dispone de un reloj que nos ofrezca un acceso a una base de tiempo suficientemente precisa, se puede uno permitir el lujo de dedicar un procesador y convertirlo en un contador muy rápido, simplemente incrementando infinitamente una posición de memoria.

Utilizando el reloj básico del sistema, la base de tiempos más fina es el tic -tiempo transcurrido entre dos interrupciones de reloj-, que en el DEC433 es de 10 milisegundos.

Cuando necesitemos medidas de gran precisión, como por ejemplo rutinas de pocas instrucciones, dedicaremos un procesador a hacer de “reloj”. Es lo que llamamos el High Resolution Clock.

Esta idea ha sido realizada en nuestra máquina gracias a la relación que mantenemos con el Research Institute de la Open Software Foundation, sito en Grenoble, Francia. Se trata de la ejecución de dos bucles imbricados, de los cuales el externo incrementa un contador de 32 bits, el “reloj”. El valor del contador se lee accediendo directamente a su dirección de memoria.

```
int loop_counter;

for (*clock_addr = 0 ;; (*clock_addr)++ )

for (loop_counter = 0;
 loop_counter < loop_iterations_per_tick;
 loop_counter++) {
};
```

La posición de memoria que sirve de contador de tiempo se puede mapear en el espacio de

direcciones de las tasks de usuario, de forma que se acceda sin sobrecarga mediante un puntero hacia esa dirección.

Hemos detectado que dicha realización tiene un problema dependiente de la arquitectura: la gestión del reemplazo de bloques de la memoria cache. Está relacionado con la alineación de las instrucciones en la memoria cache interna del procesador i486 y con la forma que se leen de esta cache hacia la cola de prefetch. En nuestro caso, la cache interna del i486 tiene 8 Kbytes de capacidad y está dividida en líneas de 16 bytes.

Depende de cómo queden alineadas las instrucciones del bucle que hace de contador de tics, unas veces da aproximadamente 2.543.576,8 tics por segundo y otras sólo 2.206.568,7 tics por segundo. El bucle ocupa dos líneas de cache. Si cae mal alineado, obliga al procesador a traer bytes que después no utiliza y ello produce la pérdida de tiempo.

Nos hemos preocupado de que el código quede bien alineado y así medir siempre con el mismo reloj.

7.6. Software de medición

7.6.1. La herramienta de medición JEWEL

Para tomar medidas del número de veces que se ejecuta un trozo de código o el tiempo que tarda en ejecutarse, hemos utilizado un *toolkit* de medición de propósito general que funciona en entornos heterogéneos y que permite, mediante comunicación por red y presentación gráfica de los resultados, una visión global del sistema bajo test. De este software, llamado JEWEL, hemos utilizado la parte de detección de eventos y recolección de datos adaptadas al microkernel Mach 3.0.

JEWEL (Just a nEW Evaluation tooL) es un sistema de medida flexible, de mucha precisión y de interferencia mínima, para la evaluación del rendimiento en tiempo de ejecución. Ha sido desarrollado por GMD (German National Research Center for Computer Science). Permite recoger datos y su posterior procesamiento introduciendo sensores en la aplicación a medir. Cuando la ejecución pasa por uno de estos sensores, se deposita en memoria compartida la información que se ha programado en el sensor: el número de evento, el instante en que ocurre el evento, y los argumentos dependientes del punto en que se está (valor de ciertas variables de la aplicación, por ejemplo).

Esta información es recogida en tiempo real por un servidor (ms_server o recolector) que vacía la memoria compartida, organizada por colas de forma que, si se utiliza correctamente, se pueden introducir y extraer datos en paralelo desde múltiples flujos y tasks sin tener que acceder en exclusión mutua y, por lo tanto, sin contención. El recolector a su vez envía la información hacia un proceso evaluador de los datos extraídos de la aplicación. La comunicación entre el colector y el evaluador se realiza mediante mensajes a través de *sockets* de UNIX y, por ello, permite que estén situados en máquinas diferentes. De esta forma, la evaluación de los resultados, su almacenamiento en ficheros y su representación gráfica no alteran las medidas, al realizarse éstos fuera del sistema en medición.

Permite trabajar con programas multi-task y multi-thread, y también puede servir para tracear su ejecución y depurarlos. Los sensores son macros que se introducen en el código. Por defecto utilizan una cola asociada al thread en ejecución en la memoria compartida con el ms_server. Tiene tres formas de medir el tiempo: a) un “reloj lógico”, que no es más que un contador que se autoincrementa y que permite estudiar la secuencia de los eventos que suceden, b) el reloj del sistema operativo, con una precisión de 10 milisegundos y c) el High Resolution Clock, con una precisión, para nuestra máquina, de 0.39 microsegundos.

Por cada experimento que se prevé realizar, se puede escribir un fichero con la lista de los

eventos que interesa medir esa vez. Este fichero se pasa como parámetro al evaluador (ms_supporter) que lo transmite hacia el recolector y éste inicializa los datos compartidos con la aplicación. Así, los eventos que figuran como “disabled” ni siquiera la aplicación pierde tiempo escribiéndolos en memoria, se ignoran desde el origen y no sobrecargan el mecanismo, todo ello sin recompilar siquiera el programa bajo test.

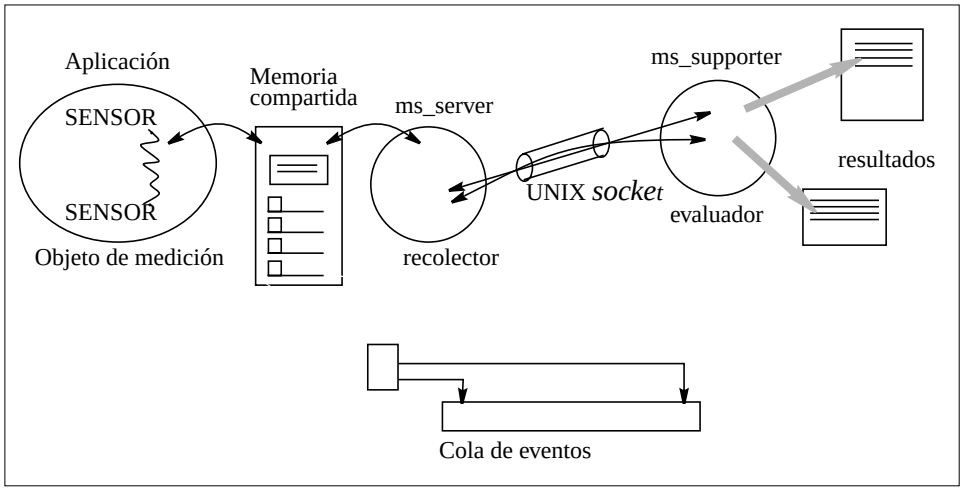

Figura 7-2 Comunicación entre los servidores de JEWEL y la aplicación

7.6.1.1. Sobrecarga debida al mecanismo de medida

Aunque JEWEL tiene una interferencia mínima con la aplicación que se está midiendo, cuando las medidas tienen precisión altísima no se puede despreciar el tiempo que se invierte en escribir el evento en memoria mediante las macros NOTICE que se incluyen en el código -es lo que llamamos sensores. Medimos la sobrecarga que introducen en la aplicación. La sobrecarga depende del número de parámetros que se le pasen, pues los tiene que depositar en la memoria compartida con el servidor que los recoge. Para la medida se ha hecho un bucle que da 400 vueltas con dos NOTICE y se mide el tiempo entre los dos, con 1 procesador físico en 1 pset.

Utilizamos, como haremos en la medidas de alta precisión, el High Resolution Clock, lo que nos da un reloj de 2.563.680 tics/segundo, es decir, 1 tic = 0.39 microsegundos.

Notice	Tics	Tiempo (μs)
NOTICE_0	4.43	1.7277
NOTICE_1	6.2	2.4180
NOTICE_2	7.48	2.9172
NOTICE_3	8.76	3.4164
NOTICE_4	9.98	3.8922
NOTICE_5	11.23	4.3797

Tabla 7-1 Sobrecarga introducida por los sensores pasándoles de cero a cinco parámetros

De la Tabla 7-1 podemos deducir que el tiempo para depositar un parámetro en memoria es de 1.25 tics (0.4875 μs).

En las upcalls, los *waiters* y el *emulator* no se es un flujo reconocido por la librería CThreads y hay que utilizar los sensores diciéndoles en cada caso cuál es la cola de eventos que se ha de usar. Crearemos esas colas de forma estática al principio de cada aplicación y nos quedaremos con los punteros hacia esa memoria compartida. Estos sensores tienen un parámetro más (la cola de eventos) y se les llama EXT_NOTICE.

Notice	Tics	Tiempo (μs)
EXT_NOTICE_1	2.776	1.08264
EXT_NOTICE_2	3.404	1.32756
EXT_NOTICE_3	3.791	1.47849
EXT_NOTICE_4	3.764	1.46796
EXT_NOTICE_5	3.888	1.51632
EXT_NOTICE_6	4.335	1.69065

Tabla 7-2 Sobrecarga introducida por los sensores pasándoles de uno a seis parámetros, cuando no se trabaja con cthreads

Verificamos que los tiempos de un NOTICE usando la cola del campo data del cthread y los EXT_NOTICE que usan una cola específica, son menores en este último caso. Además de que no siempre se pueden usar los NOTICE, utilizaremos los segundos también cuando queramos tener una precisión mayor.

La sobrecarga de un NOTICE se puede expresar de la forma:

```
t_NOTICE = tiempo_fijo + número_argumentos * tiempo_copia_argumento
```

En nuestro caso, cuando utilizamos EXT_NOTICE:

```
t_EXT_NOTICE (en microsegundos)  = 1.082 + número_argumentos * 0.155
```

Este tiempo lo restaremos a las medidas, y así eliminaremos esta componente introducida por el mecanismo de medida.

7.6.2. Traceado y extracción de contadores: nuevas llamadas al sistema

Hemos instrumentado el interior del microkernel para poder evaluar el tiempo que se dedica a operaciones de planificación de threads.

Para ello, hemos insertado, en los puntos necesarios para tomar cada medida, instrucciones que leen el valor del High Resolution Clock y lo depositan también en memoria, en una área que hemos reservado para estas medidas. De esta forma, el código introducido es mínimo y la sobrecarga de la medida ínfima. Podremos así conocer las veces que se ha invocado un procedimiento del kernel y/o el tiempo transcurrido en la ejecución de alguna de sus funcionalidades.

Cuando queramos acceder a estos contadores almacenados en memoria dentro del kernel,

los consultaremos mediante dos nuevas llamadas al sistema que hemos añadido con este fin. La lectura y la puesta a cero de esos contadores se realiza una vez acabada la toma de datos, de forma que no se altere el experimento.

7.7. Rendimiento del traspaso de funcionalidades del kernel a la aplicación

7.7.1. Cambios de contexto en modo usuario y recurriendo al kernel

En primer lugar, presentamos las medidas de tiempos que se tarda en realizar un cambio de contexto entre flujos cuando se utilizan las llamadas de la librería CThreads^{+}, es decir, se efectúa completamente en modo usuario, y cuando se recurre al kernel para realizarlo, como ocurría normalmente con la versión original de la librería.

La medida se ha realizado forzando el cambio de contexto entre dos cthreads, llamando a las rutinas cthread_yield(), cthread_hint() y cthread_handoff(cthread) (Figura 7-3). La primera es la que ofrecía la librería original, pero únicamente medimos aquí cuando provoca un cambio de contexto en usuario, no cuando llama al kernel. Las dos últimas, de la nueva librería, sólo cambian de flujo entre cthreads de usuario.

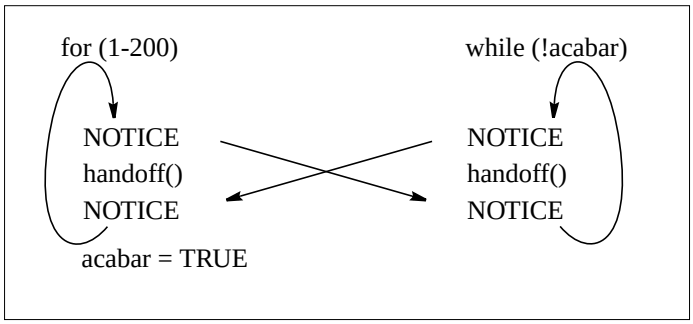

Figura 7-3 Cesión del procesador virtual, en la librería, entre dos flujos de la misma task.

La Tabla 7-3 muestra el tiempo medio para ceder el procesador virtual a otro cthread de las misma aplicación. La medida se ha efectuado entre dos flujos de una misma task dentro de un *processor*-set y sobre un único thread de kernel.

Descripción	Tics	Tiempo (μs)
cthread_handoff	41.7858	16.2964
cthread_hint	41.7119	16.2676
cthread_yield	35.6611	13.9078

Tabla 7-3 Cambio de contexto en usuario

Nuestras rutinas presentan un tiempo un poco mayor debido a que trabajan con colas de flujos con prioridades. Cthread_yield() selecciona otro cthread con política FCFS.

A continuación evaluamos el coste de acceder al kernel para planificar un thread. Utiliza-

mos la rutina cthread_yield() cuando llama al kernel porque los flujos de usuario tienen un thread de kernel asociado. La llamada al kernel es thread_switch() (Figura 7-4).

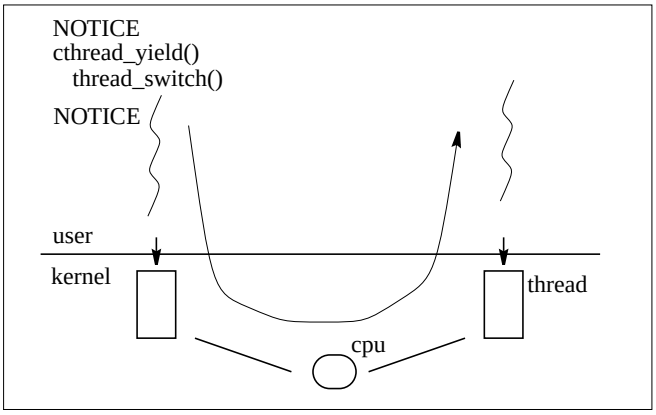

Figura 7-4 Cambio de contexto entre threads de kernel

Tenemos 2 threads de kernel que se pasan el procesador. Queremos evaluar los tres costes que puede tener en el kernel la función cthread_yield(): cuando no encuentra ningún otro thread de kernel más prioritario que el que llama y devuelve el control a él mismo, y cuando realmente cambia de contexto a otro thread de la misma task y, finalmente, entre threads de dos tasks diferentes (Tabla 7-4).

Descripción	Tics	Tiempo (μs)
cthread_yield (no mux)[a]	182.54	71.19
cthread_yield (si mux)[b]	345.80	134.86
cthread_yield (mux entre tasks)[c]	637.78	248.73

a. Llama al kernel que recalcula la prioridad, pero vuelve al mismo thread.
b. Cambio de contexto en el kernel.
c. Cambio de contexto en el kernel, entre tasks.

Tabla 7-4 Cambio de contexto llamando al kernel

Estas medidas confirman, al pie de la letra, que el cambio de contexto entre threads en el kernel es de un orden de magnitud mayor que entre flujos de la librería de usuario. Justifica ésto el esfuerzo de pasar la planificación a modo usuario, tanto si el programador utiliza las nuevas primitivas como si es la librería la que las usa por defecto.

7.7.1.1. Peso del mínimo código de cambio de contexto

En las medidas anteriores se incluye el tiempo del algoritmo de decisión de planificación y selección del nuevo thread. En el caso del kernel, el trap de entrada y las salida a modo usuario. Hemos medido también el código mínimo del cambio de contexto, aquel que salva y restaura el estado de los threads, para separar el peso del objeto de los algoritmos de planificación (Tabla 7-5).

Descripción	Tics	Tiempo (μs)
cambio de contexto en usuario	9.857	3.84
cambio de contexto en kernel	104.378	40.71

Tabla 7-5 Tiempo mínimo de un cambio de contexto en usuario y en kernel

Las medidas confirman que incluso el mínimo coste de cambio de contexto es muy inferior fuera del kernel que dentro de él. Se debe principalmente a que en el kernel hay que salvar y restaurar todos los registros del thread y actualizar más información que en usuario, donde se puede optimizar la cantidad de información a salvar dependiendo del lenguaje de programación y de los objetos gestionados por la librería.

7.7.1.2. Sobrecarga por recálculo de prioridades en el kernel.

El algoritmo de recálculo de prioridades del kernel de Mach es de propósito general, de forma que intenta repartir los procesadores de forma equitativa, recalculando las prioridades y ajustando los *quantums* a la relación de threads por procesador.

Nosotros, al considerar la planificación en el espacio y asignar procesadores a aplicaciones, consideramos que no tiene sentido un algoritmo tan costoso para esos procesadores y lo hemos reducido a una propuesta simple basada en las ideas del algoritmo ESCHED, de Straathof y Agrawala [STRA86] [STRA87].

Medimos el tiempo medio requerido para recalcular el *quantum* y la prioridad, y chequear si hay AST pendientes para el thread que corre actualmente el procesador. Es la rutina thread_quantum_update(), que se llama desde la interrupción de reloj cada 10 ms. El sistema se ha sometido a una carga media para tomar esta medida (Tabla 7-6).

Descripción	Tics	Tiempo (μs)
thread_quantum_update (Mach 3.0)	76.1	29.68
thread_quantum_update (nueva versión)	17.9	6.98

Tabla 7-6 Coste del recálculo de prioridades

En la nueva versión de la actualización de prioridades y *quantums* la aplicación puede actuar sobre el kernel para decidir el *quantum* asociado a cada prioridad dentro de su processor set. La prioridad se reduce en una unidad cada vez que se agota un *quantum* y aumenta cuando el thread ha estado bloqueado, según la duración del bloqueo y un factor que también puede modificar el usuario.

Esa sintonización puede quedar en manos del usuario porque, recordemos, queda todo localizado dentro de su pset.

El trabajo realizado nos ha llevado a reducir sobrecarga en las rutinas que ejecuta el propio sistema, a partir de algoritmos más sencillos para los recálculos de planificación necesarios. Esto lo comprobamos por los resultados obtenido en los microbenchmarks que miden la diferencia de tiempos entre las rutinas originales empleadas por Mach 3.0 (29.68 μs) y el tiempo empleado por las rutinas que ofrecemos en el sistema modificado (6.98 μs).

7.7.2. Creación de objetos planificables en usuario y en kernel

Primero presentamos los tiempos mínimos de creación de estructuras de usuario para dar soporte a los flujos por parte de la librería de CThreads, cuando no se llama al kernel para crear el procesador virtual (Tabla 7-7). Es el caso común en aplicaciones que crean múltiples cthreads. Sólo las primeras veces se llama al kernel, a partir de un punto se empiezan a reaprovechar los threads de kernel para nuevos flujos de usuario.

Hemos comprobado experimentalmente, que una aplicación de granularidad media que tenga una concurrencia de unos 200 flujos se ejecuta, en media, sobre unos 20 o máximo 30 threads de kernel, aunque el usuario no los haya limitado. El coste de creación de un objeto de kernel es superior al tiempo de ejecución de un flujo medio y, en cuanto éste acaba, se reaprovecha su procesador virtual para un nuevo cthread.

Descripción	Tics	Tiempo (μs)
cthread_fork (creación cthread)	52.42	20.44
cthread_alloc (estructura cthread)	34.30	13.38

Tabla 7-7 Tiempos mínimos de creación de flujo a nivel usuario

En segundo lugar, medimos el tiempo medio requerido para crear, desde usuario, un thread de kernel, dar un valor inicial a sus registros y activarlo (Tabla 7-8). La medida incluye la comprobación de errores de las llamadas al kernel.

Descripción	Tics	Tiempo (μs)
Tiempo total creación/activación	3199.319	1247.73
thread_create	1731.307	675.21
thread_set_state	913.771	356.37
thread_resume	554.242	216.15

Tabla 7-8 Creación y activación de un thread de kernel

Verificamos que la creación de threads de kernel, aún siendo objetos ligeros, es muy costosa si se compara con el poco esfuerzo que hay que hacer en la librería de usuario cuando ya han acabado los primeros flujos de la aplicación y se reusan sus descriptores y sus pilas.

En nuestro trabajo, esta vez hemos metido dentro del kernel el reciclaje de los threads ya usados dentro de una aplicación. Mach tenía un pool de memoria del cual cogía espacio para sus datos dinámicos, pero nosotros hacemos que el reuso sea local a la aplicación, intentando así evitar la migración de los datos de kernel entre procesadores y memorias cache de diferentes aplicaciones

Remarcamos aquí que con el reciclaje de estructuras thread que hemos introducido en el kernel cuando trabajamos en un entorno de eXc, el tiempo de thread_create se ha suprimido; sólo se crean threads de kernel en las primeras upcalls de bloqueo, luego se reciclan los antiguos eXc.

7.8. Tratamiento de las notificaciones mediante upcalls

Parte de nuestro trabajo es demostrar que, a partir de un entorno adecuado, el microkernel puede, por una parte mejorar el rendimiento de las aplicaciones adecuándose más al perfil de cada una de ellas; por otra parte, pasar parte de su gestión a la aplicación para que ésta, saque el máximo partido posible de los recursos que le ofrece el sistema.

Por ello, vamos ahora a evaluar el tiempo dedicado en la librería CThreads^{+} a tratar las notificaciones que suben con las upcalls y la planificación de un nuevo flujo de usuario mediante el scheduler o flujo nulo de cada procesador.

7.8.1. Tratamiento y planificación de flujos desde la upcall de temporización

Medimos el tiempo de tratamiento de la upcall timer, desde que llega la upcall hasta se vuelve a ejecutar el cthread interrumpido u otro cthread planificado sobre el nuevo eXc. Consta de dos partes: El tiempo de la upcall, hasta que se da control al scheduler de usuario, y el tiempo del scheduler, que replanifica el cthread (Tabla 7-9).

Descripción	Tics	Tiempo (μs)
Tratamiento del timer (total)	227.599	88.76
Upcall timer	92.87	36.22
Scheduler de usuario	130.94	51.06

Tabla 7-9 Tratamiento de la upcall de temporización

El tiempo de upcall timer incluye el salvar los registros del flujo desbancado por la temporización. El scheduler de usuario actualiza la cola de preparados de la librería y selecciona, por prioridades, un nuevo cthread.

7.8.2. Tratamiento y planificación de flujos desde la upcall de asignar procesador

Contamos el tiempo desde que se solicita un procesador (“processor_request”) al CPU *server* hasta que nuevamente nos encontramos en el programa del usuario. Es una situación compleja, pues se mezcla el hecho de enviar un mensaje -un bloqueo-, la llegada del procesador -otra upcall- y la finalización de la llamada al kernel -un desbloqueo-.

Medimos el tiempo que pasa hasta que llega la upcall de bloqueo (*blocked*) del mensaje enviado al *server*, la upcall de procesador añadido y la upcall de desbloqueo (*resumed*) del contexto de ejecución que hizo la petición (Tabla 7-10).

Descripción	Tics	Tiempo (μs)
Total de usuario a usuario	74313.23	28982.16
Llegada upcall bloqueo	5990.61	2336.34
Llegada upcall proc. added	21078.31	8220.54
Desde bloqueo a proc. added	15083.91	5882.72
Desde bloqueo a resumed	67417.61	26292.86
De resumed a thread inicial	861.61	336.03

Tabla 7-10 Tratamiento y planificación de flujos desde la upcall de asignación de procesador

La Figura 7-5 muestra la secuencia en el tiempo de la petición y la llegada de las tres notificaciones. En cada una de ellas la aplicación decide la nueva planificación de sus flujos. Es una muestra de la conversión de una llamada al kernel síncrona en un nuevo funcionamiento asíncrono, gracias a las upcalls, manteniendo la semántica del servicio solicitado.

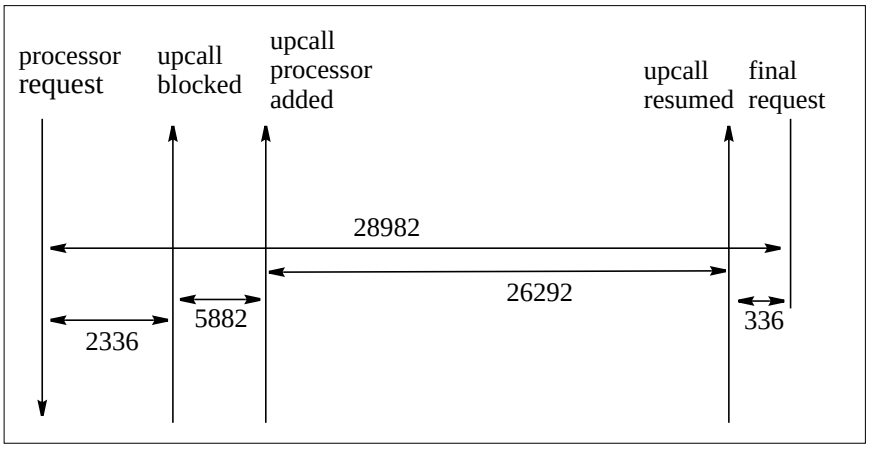

Figura 7-5 Secuencia en el tiempo de la petición y la llegada de las tres notificaciones (en microsegundos)

7.9. Planificación de la entrada / salida gestionada por la librería

Como evaluación del rendimiento obtenido en un entorno de eXc, hemos medido los tiempos de sobrecarga debidos a la preparación de los contextos de ejecución y notificaciones que el kernel hace llegar a la aplicación, comparando la ejecución de una entrada/salida en el entorno tradicional gestionado por el kernel, y en el entorno de nuestra aportación, en la que el usuario recibe las notificaciones de lo que ocurre en el kernel y decide el siguiente flujo a ejecutar.

Comparamos una llamada a UNIX de escritura, write(), pasándole unos pocos caracteres para que los escriba en un fichero. Hemos comprobado que en llamadas que transfieren grandes cantidades de datos entra el juego la gestión de la memoria virtual, y aquí queremos obviarla.

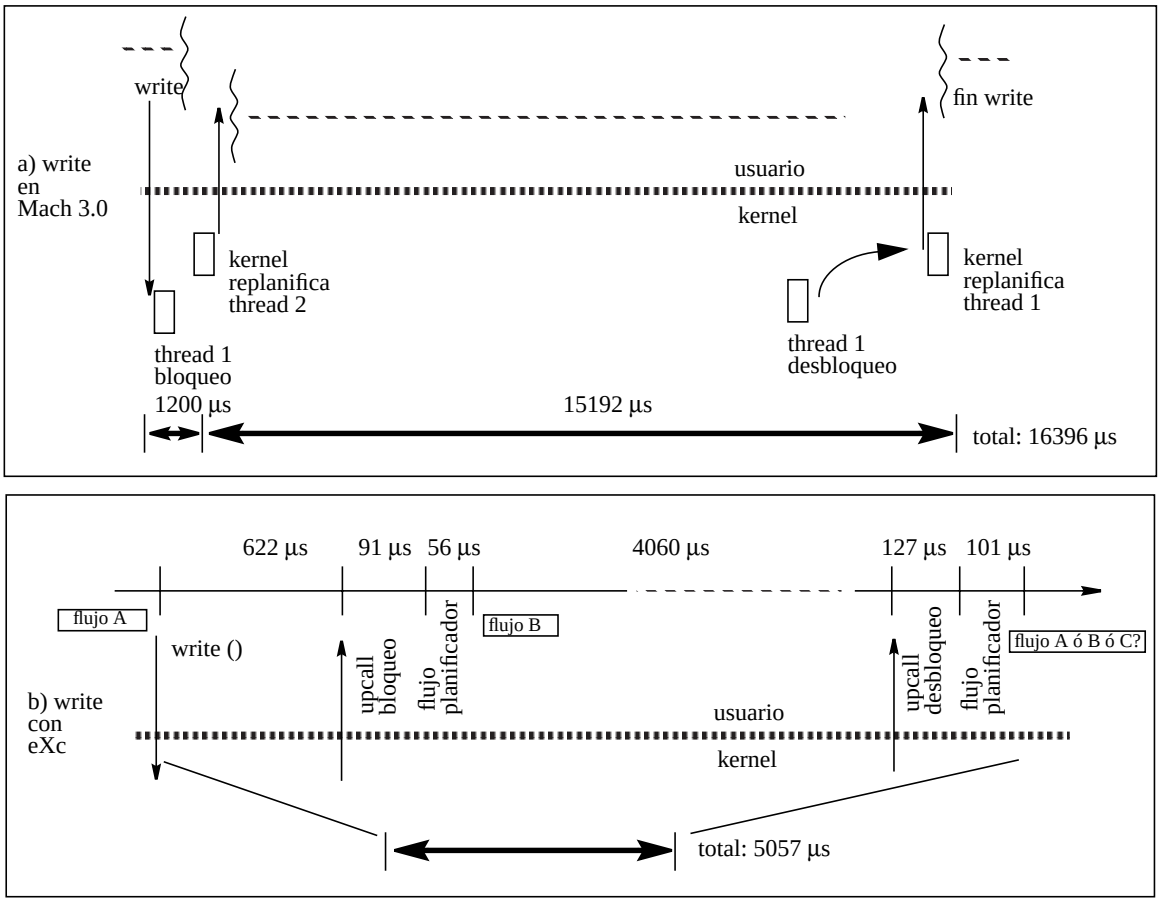

Figura 7-6 Planificación de la entrada / salida gestionada por el kernel o por la librería con eXc

Cuando el kernel de Mach gestiona la planificación de threads en un bloqueo, escoge el thread más prioritario de los que están preparados en el *processor-set* (Figura 7-6 a)). El coste de ese cambio de contexto es grande (1200 microsegundos, con el bloqueo del thread anterior). Además, cuando acaba la escritura y el thread pasa a preparados dentro del kernel, éste no replanifica hasta, como mínimo, la expiración de un *quantum*. Es por ello que se retrasa tanto el final de la entrada/salida. Hemos configurado el kernel para que lo hiciera con el mínimo *quantum*, pero éste es de 10 milisegundos, por eso el kernel no devuelve el control al thread que inició la escritura hasta 15000 microsegundos después.

En cambio, en el entorno de eXc, el kernel no actúa en lo que se refiere a la planificación. Da el control en cuanto puede al usuario, mediante las upcalls, avisándole de lo que ha ocurrido. Vemos en la Figura 7-6 b) cómo a los 600 microsegundos ya ha iniciado la transferencia y nos comunica el bloqueo del eXc. Entre la upcall de bloqueo y el planificador de usuario deciden en 91 y 56 microsegundos respectivamente cuál es el nuevo flujo. Incluso antes que los 1200 microsegundos que tardaba el kernel.

Lo mismo ocurre en el desbloqueo. Avisamos a la aplicación enseguida, desbancado al eXc en ejecución y reciclando uno de los antiguos para subir uno con identificador nuevo y los bancos de registros de los dos flujos involucrados. Ahora tanto la upcall como el planificador tardan un poco más porque han de gestionar dos contextos. Pero aún así, a los 5000 microsegundos ya podría volver a correr el flujo que hizo la llamada write(). Es decisión, una vez más, únicamente

del usuario.

Evidentemente, el tiempo que ha estado el primer flujo bloqueado en el kernel se ha aprovechado para hacer correr a otro sobre el mismo procesador.

Otro factor que sobrecarga la planificación normal de Mach es el recálculo de prioridades que realiza cada vez que ha de seleccionar un nuevo thread. En nuestro caso no hay que recalcular nada: la aplicación posee tantos procesadores físicos como virtuales (eXc). No hay concurrencia en el kernel y este hecho afecta a la medida anterior.

Podemos también probarlo de otra forma. Cuando la aplicación tenga activado el entorno de eXc pero sin rutina de tratamiento de upcall. Significa que no quiere tratar ese evento, que lo gestionará el kernel sin avisar al usuario, decimos, de forma transparente. El procesador queda reservado para el eXc, y por tanto el flujo de usuario, que hizo la llamada al kernel y, en cuanto finaliza el bloqueo, hace un *handoff* en el kernel sobre el thread que soporta al eXc, saltándose toda sobrecarga de replanificación que considere a otros threads.

Descripción	Tics	Tiempo (μs)
write transparente (5 bytes)	4237.61	1652.67

Tabla 7-11 Gestión del kernel ante un bloqueo transparente

De esta forma, cuando el usuario no quiere tratar un evento, por la gestión de dedicación de procesadores “transparentes” dentro del kernel, el desbloqueo de un thread de kernel ni siquiera ha de desbancar a otro como ocurre cuando hay que notificar. La pérdida de tiempo en el kernel es mínima. Un ejemplo es la llamada write(0 del ejemplo anterior tratada de forma transparente. Tarda 1652 microsegundos.

Se puede objetar que ese procesador no ha hecho nada durante ese tiempo, pero hay que recordar que es un procesador dedicado a la aplicación y ella decide cómo usarlo.

7.10. Evaluación del entorno realizado en cuanto a su transportabilidad

En las realizaciones llevadas a término en un entorno concreto es importante evaluar el coste de su transportabilidad a otros entornos. Esto hace que el modo de trabajo propuesto sea fácilmente aplicable a otros sistemas ya existentes.

Para evaluar ese transporte, hay que examinar por una parte los objetos utilizados, la semántica de trabajo del entorno y la cantidad de código modificado. Es en este último punto en el que nos vamos a centrar ahora. Respecto a los otros dos, la sencillez y la generalidad en que nos hemos basado, no limitan ni suponen un handicap para trabajar en otros microkernels. Quizá el objeto más específico del entorno sea el de *processor-set* representando a la aplicación como unidad planificable en el sistema.

El código modificado para realizar nuestro entorno de eXc podemos dividirlo en dos frentes:

- El código añadido a rutinas y funciones ya existentes.
- Funciones y rutinas añadidas al sistema.

No incluimos el código añadido a nivel usuario. El hecho de haber trabajado de modo totalmente independiente al resto de librerías de usuario y el no haber utilizado tampoco soporte alguno del sistema operativo, hace que sólo tenga que modificarse el tipo de objeto con que se trabaje y las estructuras de datos relacionadas con la planificación en el nivel de usuario.

Vamos a examinar las modificaciones en el interior del kernel.

7.10.1. Código modificado en el kernel

El código modificado de rutinas y funciones ya existentes en el sistema se ha limitado a puntos muy concretos y rutinas también muy concretas. Influye, claro está, el hecho de que nos hayamos limitado a eventos que afecten a la planificación de los flujos y no hayamos tenido otros en cuenta.

La modularidad del código de Mach y su propia estructura interna, ha contribuido también a esta sencillez.

Como ya hemos comentado anteriormente, al explicar la realización del entorno, las rutinas modificadas han sido:

- thread_block(): evaluando si el bloqueo se da en condiciones de *scheduler activation*. Si es el caso, se llama a la rutina correspondiente, que comentamos en el siguiente punto.
- thread_setrun(): evaluando si el desbloqueo se da en condiciones de *scheduler activation*. Si es así, se activan los avisos (ASTs) correspondientes.
- action thread: a la hora de reasignar procesadores, hay que comprobar que la aplicación a la que se añade un procesador o la aplicación a la que se le quita, trabajan con modelo de eXc. En uno y otro caso, se llama a la rutina correspondiente o se activan los avisos (ASTs), respectivamente.
- ast_taken(): se han añadido tres nuevos ASTs: AST_NOTIFY, AST_PREEMPT y AST_TIMER.
- alltraps: la entrada en el kernel a través de una llamada al sistema activa, si la aplicación trabaja con modelo de eXc, el mecanismo de *scheduler activation*.
- return_from_trap: la salida del kernel desactiva el flag de *scheduler activation*.

Además se han desactivado todas las rutinas de planificación de threads de kernel para estas aplicaciones.

7.10.2. Rutinas añadidas en el kernel

Las rutinas añadidas se han reducido a dos nuevas funciones en el interior del kernel y tres nuevas llamadas al sistema para poner en marcha el entorno y el temporizador.

Las rutinas añadidas han sido:

- sa_activate(): llamada en el momento de reconocer una *scheduler activation*, prepara la activación de las upcalls, recicla viejos eXc o crea nuevos si es necesario.
- exit_kernel_eXc(): creación del bloque de activación de la upcall y salida a usuario.

Las llamadas al sistema han sido:

- eXc_enable(): para empezar a trabajar con el modelo de eXc.
- eXc_upcall_check_in(): para dar de alta cada una de las rutinas de tratamiento de las upcalls que se quieran gestionar.
- eXc_timer_set(): para activar y desactivar el mecanismo de temporización.

La especificación de estas llamadas se encuentra en el Apéndice C.

7.11. Referencias y Bibliografía

[ANDE89] “The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors”
Thomas E. Anderson et al.
Performance Evaluation Review Vol.17 Num.1, May 1989.

[BURK89] “Performance-Measurement Tools in a Multiprocessor Environment”
Helmar Burkhart and Roland Millen
IEEE Transactions on Computers Vol.38 Num.5, May 1989.

[CARR86] “Profiling under ELXSI UNIX”
D.A. Carrington
Software Practice and Experience, Vol.16 Num.9, September 1986

[CHAB88] “Using sar to Zero in on Performance Bottlenecks”
E. Chaban
UNIX WORLD, July 1988, pp. 117-121.

[GIL91] “Caracterización de la ejecución del kernel de UNIX para su paralelización en entornos multiprocesador”
Marisa Gil, Nacho Navarro
UPC/DAC Report N. RR-91/06, Enero 1991.

[HINN88] “Accurate Unix Benchmarking: Art, Science, or Black Magic?”
David F. Hinnant
IEEE Micro, October 1988.

[LARR91] “Implementació d'un monitor hardware per a mesurar els events d'un sistema operatiu”
J.L. Larriba, Josep Cols, Rubén Hidalgo, Toni Perera, Jesús Labarta, Nacho Navarro, Marisa Gil
UPC/DAC Report N. RR-91/05, Gener 1991.

[STRA86] “UNIX scheduling for large systems”
J.H. Straathof, A.K. Thareja, A.K. Agrawala
Proceedings of the Denver USENIX Conference, January 1986.

[STRA87] “Methodology and Results of Performance Measurements for a New UNIX Scheduler”
Jeffrey H. Straathof, Ashok K. Thareja and Ashok K. Agrawala
Proceedings of the Washington D.C. USENIX Conference, Winter 1987, pp. 165-180

8

Conclusiones y líneas abiertas

"Habla Sancho: «Las obras que se hacen apriesa nunca se acaban con la perfección que requieren»."

Miguel de Cervantes ("D. Quijote de La Mancha", II Cap. IV)

ABSTRACT: En este último capítulo hacemos un resumen de las aportaciones realizadas en el marco de este trabajo. Damos un balance de las metas obtenidas y obtenibles, a partir de los objetivos que fueron inicio de la investigación realizada y de los resultados alcanzados. Dejamos también líneas de trabajo abiertas que pensamos pueden ser una buena aportación en el campo de los sistemas operativos como soporte a las aplicaciones paralelas que se prevén en un futuro próximo.

Capítulo 8: Conclusiones y líneas abiertas

8.1. Introducción

El objetivo de nuestro trabajo ha sido defender la tesis de que, para los entornos de trabajos actuales, multiprocesador, una opción válida para obtener el mejor rendimiento de las aplicaciones paralelas de propósito general es conseguir que el kernel y la propia aplicación cooperen en la gestión de los recursos.

El entorno en que hemos centrado nuestro estudio ha sido arquitecturas multiprocesador de memoria compartida, con tecnología microkernel como software de sistema. Los trabajos a los que dan soporte son aplicaciones multiflujo de granularidad media -lo que conocemos como librerías de threads- de propósito general y en un entorno multiusuario.

Justificamos por un lado, el trasvase a nivel de usuario de funcionalidades de planificación de trabajos que hasta ahora residían en el kernel, por su coste. En modo usuario se consigue trabajar con tiempos de ejecución de un orden de magnitud menor.

Por otro lado, la situación actual de estos entornos es el trabajo independiente de diferentes niveles de planificación: flujos por un lado y procesadores virtuales por otro. Ésto lleva a resultados negativos en la gestión de flujos que necesita la aplicación, más concreta y adaptada al trabajo específico que realiza, al “chocar” con la planificación de procesadores virtuales que hace el kernel, más general y partiendo de parámetros diferentes.

El fin de la planificación de un trabajo es que éste se realice de la manera más eficiente y rápida posible a partir de los recursos que el sistema le asigne. La planificación del sistema, sin embargo, busca la igualdad y repartición justa de recursos entre todos los trabajos del sistema.

En sistemas experimentales o académicos, se está trabajando actualmente en mecanismos que permiten algún tipo de comunicación, bien de la aplicación hacia el kernel a través de llamadas al sistema, bien del kernel a la aplicación, a través de upcalls, o zonas de memoria con acceso de lectura para ambos.

A partir de estos mecanismos, hemos estudiado qué herramientas y políticas de comunicación se podían utilizar para ofrecer la máxima versatilidad y eficiencia al sistema global y, a la vez, de qué abstracciones disponíamos como apoyo y base para dicha comunicación.

Al mismo tiempo, había que profundizar en objetos y abstracciones adecuadas para realizar con idoneidad la gestión de avance de vida de la aplicación. Ni tan cerradas como el concepto de proceso en UNIX, con espacios de direcciones totalmente disjuntos y privados, ni tan relajadas como los threads de kernel que carecen de privacidad, incluso de localidad en cuanto a que fueran propios, fuera de su misma pila.

En el banco de pruebas de trabajos paralelos, hemos desarrollado políticas y mecanismos de comunicación y acceso en exclusión mutua entre la aplicación y el kernel, que trabajan ahora en cooperación respecto a determinados acontecimientos que afectan a la gestión de flujos.

El resultado ha sido la realización de un nuevo entorno de trabajo, compatible con los entornos actuales de trabajo sobre microkernels. Probado con microbenchmarks y pequeñas aplicaciones multiflujo reales, hemos extraído las conclusiones que presentamos a continuación.

También comentaremos mejoras que pueden completar el trabajo y líneas que quedan abiertas para futuros trabajos de investigación.

8.2. Aportaciones realizadas en este trabajo

Presentamos a continuación las aportaciones de este trabajo, en el campo de la realización y en la filosofía y diseño de soluciones. Hemos partido de los multiprocesadores en cuanto que son utilizados como máquinas de próposito general, en los que pueden correr diferentes aplicaciones, posi-

blemente cada una de ellas siguiendo un modelo de paralelismo distinto.

Nuestro estudio abarca a la aplicación en su conjunto, con todas sus partes, en cuya ejecución el sistema operativo toma una parte activa y decisoria. Y nos hemos centrado en aplicaciones paralelas multiflujo de granularidad media.

Al referirnos a una granularidad media, descartamos a los flujos de granularidad fina, o sin peso, como son los generados con herramientas automáticas (compiladores que paralelizan, etc.). Tampoco hablamos de flujos de granularidad gruesa como los que llevan asociado su propio espacio de direcciones (por ejemplo, los procesos UNIX).

Hemos diseñado al completo un sistema de planificación basado en el conocimiento por parte de la aplicación del número de procesadores físicos que posee. A partir de la abstracción de procesador virtual, hemos dotado a la aplicación de la capacidad de gestionar la planificación de los procesadores físicos que el sistema le proporciona. Para ello, hemos redefinido el concepto de procesador virtual como contexto de ejecución (eXc) que el kernel ofrece a la aplicación para que pueda ejecutar un flujo en cada uno de los procesadores físicos que le ha asignado. Y hemos ofrecido un entorno en el que el kernel pueda comunicar de manera asíncrona con la aplicación, transmitiéndole los eventos que puedan afectar a ésta para decidir una replanificación en sus flujos.

Vemos a continuación las aportaciones concretas en cuanto a:

- La planificación de trabajos paralelos partiendo del objeto aplicación como unidad de planificación.
- Las realizaciones hechas a nivel de usuario en el entorno ofrecido.
- Las realizaciones hechas en el kernel.
- Las realizaciones hechas en el campo de comunicación y cooperación entre la aplicación y el kernel.
- Herramientas desarrolladas para poder evaluar mejor los acontecimientos del sistema.

8.2.1. Concepto de aplicación

Para el entorno que hemos desarrollado y las soluciones realizadas, era básico el concepto de aplicación paralela del cual partíamos. Era éste nuestro cliente y a quien de manera eficiente y sencilla queríamos dar soporte.

Definimos aplicación como el conjunto de acciones realizadas para obtener un fin. Existe una sincronización y una comunicación fuerte y estrecha entre ellas, no sólo de modo directo, sino indirecto: una aplicación acaba cuando lo haya hecho la última de sus acciones. Éso involucra a todas en el avance de la aplicación desde su comienzo hasta su final. De nada sirve que una acción haya finalizado hace tiempo, si hay otras que no pueden continuar.

Esta relación entre los flujos de una aplicación puede ser más o menos alta, en función del tipo de problema y la granularidad ofrecida, pero es un factor que no puede quedar al margen a la hora de definir abstracciones y políticas de trabajo.

A partir de políticas de planificación de particionado de la máquina, hemos aislado a las aplicaciones, unas de otras, y hemos habilitado que en cada partición el usuario pueda decidir qué política de planificación y qué quantums y recálculo de prioridades quiere. Con ello conseguimos, por un lado, que cada aplicación pueda ajustar la planificación que mejor rendimiento le suponga. Por otro, que no queden afectadas unas por el funcionamiento de otras.

Además, al contemplar el entorno realizado como una submáquina con todos sus recursos dedicados a una única aplicación, los tratamientos de cada situación son mucho más sencillos.

Las políticas y mecanismos de planificación que se ofrecen también son más sencillos: al estar todas las acciones enfocadas a un mismo fin -el fin de la aplicación- lo más eficiente es acabar los trabajos cuanto antes, no forzar cambios de contexto que solo abocan a una sobrecarga en el tiempo de ejecución. En este entorno:

- Las prioridades estáticas son las más adecuadas. Se ven como una manera de ordenar la urgencia de los diferentes trabajos de la aplicación, más que como una búsqueda de repartición igualitaria del recurso procesador.
- Los acontecimientos asíncronos afectan siempre, de manera directa o indirecta a cualquiera que sea el flujo interrumpido. Por ello, es más eficiente tratarlos secuencialmente hasta acabar que definir una jerarquía y un desbanque entre ellos. Con ello sólo se conseguiría un aumento en los cambios de contexto realizados: todos los trabajos son igualmente importantes puesto que todos han de acabar.

8.2.2. Realizaciones a nivel usuario

A nivel usuario hemos seguido la filosofía actual de las tecnologías microkernel. Hemos definido unos *templates* de trabajo que se ofrecen al usuario en forma de librerías para que disponga de las mismas funcionalidades que hasta ahora le ofrecía el kernel, pero en modo usuario. De este modo son más eficientes.

Por estar ahora a nivel usuario, son más fáciles de modificar y se permite que cada aplicación pueda desarrollar sus propios métodos de trabajo. Los usuarios más experimentados pueden adaptar las partes que vean más convenientes al perfil concreto de sus trabajos

Como primer paso hemos desarrollado una librería de flujos basada en CThreads, a la que hemos añadido políticas de planificación más potentes y poco explotadas actualmente a nivel usuario:

- Política de prioridades estáticas.
- Mecanismos de planificación directa a flujos concretos, a través de pistas (*hints*) o conocimiento del trabajo en cuestión (*handoff*).
- Mejoras en las primitivas de sincronización de la propia librería utilizando estos mecanismos y políticas, como opción de compilación (primitivas de sincronización a dos niveles).
- Retorno de mayor información a la aplicación de lo que realmente ha ocurrido dentro de la librería, para que sea el usuario, si quiere, quien tome las decisiones.

Estas librerías que, como parte del sistema, se ofrecen a las aplicaciones se han hecho totalmente independientes de otras librerías o herramientas que pueda utilizar el usuario.

Esta independencia se manifiesta:

- A nivel de utilidades de usuario, como ha sido en nuestro caso concreto, no utilizamos primitivas de la librería CThreads, y actuamos sin alterar el paso de los flujos de aplicación a servicios de UNIX a través del emulator (librería de emulación).
- A nivel de sistema, eliminamos la utilización de mecanismos de bloqueo o desbanque de flujos, como ocurre en el caso de los "waiters", que hemos sustituido por nuestros propios flujos nulos. También nos mantenemos al margen de utilidades concretas del sistema o subsistema subyacientes: otras implementaciones experimentales o comerciales hacen uso intensivo de llamadas a UNIX como son los signals.

Conseguimos así la transportabilidad del entorno sobre otros sistemas, con un número mínimo de modificaciones. De igual manera, hacemos transportable cualquier aplicación a nuestro entorno, sin cambiar la semántica de su funcionamiento.

Para mantener la misma funcionalidad que ofrecía la planificación del sistema, era importante contar con un mecanismo de desbanque de flujos sin servirnos de ninguna utilidad del lenguaje o del sistema que pudiera utilizar la propia aplicación, y de la que por tanto, la estaríamos privando. Para ello:

Hemos realizado upcalls de temporización del kernel a la aplicación, permitiendo entornos de tiempo compartido totalmente gestionados a nivel de usuario y desbanque de flujos.

En cuanto a la gestión de las exclusiones mutuas, hemos pasado a trabajar en un entorno desbancable. Podríamos caer en abrazos mortales si se utilizan *spin locks* u otras primitivas de sincronización de manera indiscriminada por la aplicación. Somos partidarios de dejar a la aplicación el control de sus propias exclusiones mutuas. No obstante:

- Hemos desarrollado una nueva primitiva de *spin lock* que deja marcado al posesor del lock. Ésto permite pasarle el control directamente a él, cuando haga falta, para que pueda acabar su trabajo, en caso de ser desbancado, o no encontrarse en ejecución. Nosotros la hemos utilizado internamente en la librería para la protección de la cola de flujos preparados de la aplicación.
- Hemos diseñado un nuevo mecanismo de inserción en la cola de preparados de la aplicación de modo que no hace falta ningún protocolo de exclusión mutua. Así conseguimos reducir los cuellos de botella en el acceso a esta cola global.
- Permitimos también la inhibición momentánea de eventos asíncronos (upcalls) de manera local a cada procesador, convirtiendo trozos de código en no interrumpibles. El kernel retrasa entonces los avisos hasta la próxima evaluación (como muy tarde a la salida de la siguiente interrupción de reloj).

Es importante resaltar que la retención de eventos en el kernel por parte de la aplicación no puede ser indefinida. El kernel puede determinar acabar con la ejecución de una aplicación ante el mal uso de estos mecanismos.

8.2.3. Realizaciones a nivel kernel

El papel del kernel, en nuestro entorno de trabajo, pasa a ser el de gestor de los recursos hardware de planificación y árbitro entre las diferentes aplicaciones.

La política de planificación ofrecida por el kernel a las aplicaciones es de espacio compartido. La dedicación de procesadores a aplicaciones ya ha sido estudiado y presentado en múltiples trabajos como la política de planificación más eficiente en sistemas multiprocesadores para aplicaciones paralelas.

La novedad de nuestro enfoque es la de ofrecer un entorno multiusuario de propósito general basado en espacio compartido, en lugar del clásico tiempo compartido. De esta manera puede ofrecer una submáquina dedicada a cada aplicación.

Hemos desarrollado un CPU server que es el que mantiene las políticas de asignación y repartición de procesadores entre las aplicaciones. No era nuestro propósito entrar en este campo y las políticas actuales son sencillas.

A partir del objeto processor-set de Mach, hemos podido aislar e individuar aplicaciones.

Pueden convivir así aplicaciones que trabajen con el entorno tradicional de planificación de threads de kernel y aplicaciones que trabajen en entorno eXc. Los recursos y políticas de cada una de ellas queda totalmente aislado del resto.

A este nivel de planificación, además:

- Hemos parametrizado los quantums y el rango de prioridades de cada aplicación, cuando trabajan en el entorno de threads.
- Hemos eliminado toda la planificación de procesadores virtuales en el kernel cuando se trabaja con entorno eXc. El paralelismo que se ofrece a este nivel es real: el número de procesadores virtuales coincide con el número de procesadores físicos. Además el mapeo entre ellos es de 1-1.

Para las aplicaciones que trabajan en entorno de eXc, el kernel (su diseñador) ofrece unos eventos que ella puede gestionar ante determinadas circunstancias. Son los llamados momentos de *scheduler activation*.

Nosotros hemos definido como momentos de *scheduler activation* los que puede provocar la propia aplicación con su petición de servicio al sistema y la reasignación de los procesadores. El resto de acontecimientos que ella no controla (por ejemplo, las excepciones) será tratado de modo transparente por el kernel.

Tomamos como decisión de diseño que los acontecimientos asíncronos lleguen al procesador en el que ocurrieron. Así se permite trabajar a nivel usuario con políticas de planificación de afinidad al procesador.

Este tipo de políticas mejora notablemente el rendimiento de las aplicaciones cuando su comportamiento es cerrado respecto a la recepción de datos (no recibe cantidades grandes de datos que puedan modificar el contenido de sus caches)[1] y cuentan con un conjunto de recursos dedica-

dos.

Hemos trabajado sólo con la parte de planificación de flujos. Los acontecimientos que el kernel deja en manos de la aplicación son, por tanto, los que pueden afectar al orden en la ejecución de sus flujos. En concreto, bloqueo/desbloqueo de flujos, reasignación de procesadores y un temporizador, comentado en el punto anterior, que hemos añadido para que el módulo planificador de la aplicación pueda recuperar el control.

El tratar o no todos los eventos o sólo parte de ellos es decisión de la aplicación y puede cambiar durante su ejecución.

La aplicación puede cambiar de modo de trabajo (entorno eXc o thread) durante su ejecución. También puede modificar las rutinas de tratamiento de los diferentes eventos y su decisión de tratarlos o no.

8.2.4. Comunicación entre la aplicación y el kernel

Para la comunicación entre la aplicación y el kernel hemos trabajado con avisos asíncronos que suben todo el estado del evento ocurrido a la aplicación y con datos compartidos entre el usuario y el kernel, consultables y modificables por ambos.

A partir de aquí, la aplicación tiene ya acceso a los procesadores físicos y estamos en condiciones de que el kernel pueda comunicarle los acontecimientos que lleguen al sistema y puedan influir en la aplicación, para que ella misma gestione sus flujos.

8.2.5. Evaluación de los acontecimientos del sistema y del mecanismo de upcalls

A parti de la herramienta JEWEL y la utilización de un High Resolution Clock hemos incluido sensores en la librería y el kernel y medido el tiempo dedicado a la creación de objetos de kernel y de usuario y a la sobrecarga introducida por la planificación en cada nivel.

Hemos medido también cada una de las upcalls, y vemos que se gana al realizar la planificación en modo usuario, no solo por seleccionar al flujo más conveniente, sino también en rapidez en la toma de decisiones.

8.3. Estado actual de los microkernels

El estudio profundo que hemos realizado del sistema en el que hemos trabajado para llevar a cabo el transporte de nuestro diseño, nos ha llevado también a comprobar de manera crítica algunos de los mecanismos de planificación, y políticas, que se ofrecen. De igual modo, hemos comprobado la potencia de algunas abstracciones y objetos poco utilizados por los usuarios y que les permitirían obtener mejores resultados en la ejecución de sus trabajos.

El trabajar a nivel de *processor set*, dedicando procesadores físicos a aplicaciones, no está lo suficientemente explotado, como lo muestra el hecho de no ofrecer ningún servidor de procesadores en las distribuciones comerciales de Mach. Como hemos comentado durante todo nuestro trabajo, esta política de dedicar recursos a aplicaciones es una de las que mejores resultados se

---

1. Se consideran aplicaciones abiertas los servidores, por el número de datos que reciben y tratan, independientes en cada servicio.

obtiene en entornos paralelos. Nosotros nos hemos apoyado en este objeto para definir la aplicación como unidad planificable en el sistema.

Es una limitación el no poder parametrizar aún más la planificación a este nivel. Nosotros lo hemos conseguido a partir de nuevas llamadas al sistema, ya que, en realidad, existen los campos locales a cada *processor set* para tener sus propios valores, independientes del resto.

Una opinión general en los entornos de desarrollo de microkernels es que la madurez de las abstracciones a la que se ha llegado exige, no obstante, un reinicio desde cero en la realización de los diseños. Mach microkernel, desarrollado a partir de Mach monolítico, con UNIX BSD en su interior, es una prueba evidente de ello. Sería bueno partir de un nuevo diseño en las políticas y mecanismos, totalmente diferenciado de la herencia de UNIX, ya obsoleto para multiprocesadores y, en general, para las tecnologías actuales.

Hemos observado que los mecanismos y políticas más avanzados que se ofrecen en entornos de experimentación son muchas veces incompatibles entre ellos. Como ejemplo comprobado por nosotros mismos, está el mecanismo de *handoff* en el IPC en Mach, que no funciona entre threads situados en diferentes *psets*. Ésto hace que, en la práctica no sean alcanzables las metas propuestas en la teoría.

8.4. Líneas abiertas

El entorno realizado puede considerarse la primera parte dentro de la línea de trabajo de diseño de sistemas operativos que vemos necesarios para el futuro. Dejamos camino abierto a una gestión mayor confiada a nivel de usuario en forma de librerías u otras herramientas.

Conforme las propias arquitecturas y microkernels evolucionen, podremos valorar el alcance de las aportaciones realizadas y de aquellas que se prevén posibles y alentadoras en cuanto a resultados.

Mostramos a continuación unas cuantas, las que más interesantes nos han parecido.

8.4.1. Mecanismos de desbanque “conscious”

A la hora de hacer llegar una notificación (activar una upcall), hemos basado nuestro diseño en políticas de afinidad del procesador. Así, la notificación de un flujo que se desbloquea llega al mismo procesador en el que se bloqueó. Si dicho procesador ya no pertenece a la aplicación se elige el primero que tenga la aplicación.

Sería interesante poder desbancar al procesador que esté realizando trabajo menos útil para la aplicación. Ésto podría conseguirse fácilmente si la aplicación, por ejemplo, diera esta información al sistema a través de la zona de datos compartidos.

Quizá la propia aplicación debiera ser su propio gestor de procesadores, si logramos definir una barrera de derechos y de gestión lo suficientemente flexible como para que cada aplicación pueda gestionar sus recursos.

8.4.2. Gestión de memoria a nivel usuario

Sería también interesante poder contar con la participación de la aplicación en otro tipo de decisiones. Por ejemplo, en la gestión de la memoria, parámetro también importante a la hora de decidir qué flujo poner en ejecución.

En este caso, el kernel debería dar soporte a un particionado de la memoria física, tal como en este trabajo hacemos con los procesadores. Creemos que esta gestión conjunta de la memoria se planteará principalmente en máquinas NUMA o NORMA, y lo dejamos como línea

abierta a proseguir después de este trabajo.

8.4.3. Estado de las aplicaciones de cálculo: registros de coma flotante

Las aplicaciones de cálculo intensivo utilizan los coprocesadores de coma flotante. Generalmente, el kernel, ayudado por el hardware de control, detecta que el flujo actual lo está utilizando y, cuando haya un cambio de contexto en el kernel, deberá salvar los registros de coma flotante. El kernel no usa el coprocesador, por lo tanto sólo hay que hacerlo cuando se interrumpe un proceso en modo usuario.

Ahora que las upcalls pueden desbancar un flujo de la aplicación y sube el estado como parámetros de la upcall, deberá subir también, cuando detecte que se estaban usando, los registros de coma flotante, para que se restauren en la planificación en usuario.

8.4.4. Potencia de planificación a nivel usuario

El kernel nos notifica, mediante una upcall, los flujos que se nos han bloqueado. El hecho de segregar a estos threads en una cola de bloqueados dentro del kernel, sin poder continuar su ejecución es, claramente, una decisión de diseño que hemos elegido por sencillez y por seguir el comportamiento que tenía el entorno que hemos elegido para modificar. Así, podemos establecer una evaluación del rendimiento y poder comparar ambos. Qué duda cabe que podríamos haber conseguido un entorno mucho más rico y complejo, permitiendo E/S asíncrona a nivel de usuario y, por tanto, consintiendo en la ejecución de un flujo con varios servicios pendientes dentro del sistema.

No era nuestro objetivo en este trabajo adentrarnos en la semántica de las operaciones a nivel usuario, ni en un entorno de usuario más o menos avanzado. Nuestra investigación se centra en el sistema operativo y en su modo de colaborar con la aplicación. Dejamos a los diseñadores e investigadores de software cómo la aplicación explote de un modo mayor o menor las ayudas que el sistema le ofrece.

8.5. Conclusiones del trabajo realizado

El mapeo entre los flujos de la aplicación, los procesadores virtuales del sistema y los procesadores físicos, es una de las decisiones más influyentes y determinantes en la eficiencia de una aplicación paralela.

El sistema es el dueño y gestor de los procesadores físicos y punto de comunicación de éstos con la aplicación.

Aunque desde el punto de vista teórico es deseable que la programación de una aplicación sea totalmente independiente del número de procesadores físicos de los que vaya a disponer, no siempre es posible de cara a obtener la mayor eficiencia real. Esto lo observamos en la mejora de tiempos de ejecución que obtenemos cuando el número de procesadores virtuales coincide con el de físicos (Capítulo 4).

Es importante, pues, saber el grado de confiabilidad que el kernel nos da en cuanto a recursos ofrecidos; es decir, si nos está ofreciendo concurrencia o paralelismo a nivel físico. Y ha de ser capaz de adaptarse a las necesidades de las aplicaciones, según necesiten concurrencia o paralelismo[2].

Por otro lado, las aplicaciones actuales saben mucho más de la planificación que necesitan que el sistema. El mejor servicio que puede hacerles éste es ser lo más transparente posible, de

2. Hay aplicaciones diseñadas para un determinado número de flujos, que no pueden adaptarse fácilmente en ejecución a otra cantidad, aún a costa de perder eficiencia porque estén trabajando sin paralelismo real.

manera que la aplicación sepa, por una parte, el estado real de los recursos que tiene asignados en cada momento; por otra parte, que pueda modificar ese estado según lo necesite.

En los sistemas de hoy en día ésto se traduce:

- En el aspecto lógico, cediendo de manera voluntaria el máximo número posible de decisiones a la aplicación y, para ello, “perdiendo peso”.
- En el físico, a través de un mapeo 1-1 entre los procesadores virtuales y los físicos, para que la aplicación disponga en todo momento de los recursos reales que tiene.

Éste es el punto de compromiso de dejar la gestión al usuario pero la protección al sistema operativo.

Con nuestra realización de paso de gestión de flujos del kernel a la aplicación, no sólo ganamos tiempo de ejecución para otros flujos de la aplicación, sino que la sobrecarga de tiempo en sistema es mínima, mejorando incluso el tiempo que emplea el kernel cuando realiza el mismo trabajo.

Por último, pero no lateral, ha de quedar claro que todos los flujos de una aplicación cooperan en la consecución de un mismo objetivo -el objetivo de la aplicación- y ello implica una dependencia entre ellos, ya sea física (acceso a los mismos recursos) o lógica (sincronización y comunicación). Esta relación entre los flujos de una aplicación puede ser más o menos alta, en función del tipo de problema y la granularidad ofrecida, pero es un factor que no puede quedar al margen a la hora de definir abstracciones y políticas de trabajo.

Como ocurre en todas las realizaciones de mecanismos similares de comunicación y gestión conjunta entre el kernel y el usuario que conocemos, nos falta la experiencia de ejecutar aplicaciones reales y suficientemente grandes que validen estos mecanismos y políticas. Tampoco son por ahora comparables los microbenchmarks de evaluación, pues cada realización tiene sus peculiaridades y funciona sobre máquinas y microkernels diferentes. Es la asignatura pendiente que nos queda a los diseñadores, convencer a los usuarios de que estas herramientas les van a ser útiles y sus aplicaciones mejorarán. De todas formas, al ofrecérselas en formato de librerías de programación multiflujo, el diseñador de sistemas aplica los nuevos conceptos en su realización interna, y el usuario las están ya usando implícitamente en sus programas.