Parte II

La aplicación como ente y gestora de su planificación

3

La aplicación como entidad

“Caminante no hay camino,
se hace camino al andar”

Antonio Machado

ABSTRACT: Una de las dificultades que tiene la sintonización del sistema es la de obtener el máximo rendimiento para todas las aplicaciones que componen su carga actual, debido a la creciente variedad de aplicaciones que encontramos hoy día. En concreto, en el ámbito de las aplicaciones paralelas, que es donde situamos nuestro trabajo.

Presentamos a continuación, de todo el abanico de trabajos paralelos que se da en la actualidad, aquel en el que se situan las aplicaciones a las que queremos dar soporte.

Partimos de la carencia de un objeto que se ofrezca, ya sea a nivel usuario, ya sea a nivel sistema, para definir y aunar los trabajos de una misma aplicación y la relación entre ellos.

Empezamos un esbozo de lo que va a ser nuestro entorno de trabajo a partir de la aplicación como unidad de planificación, tanto en cuanto a políticas, y parámetros de dichas políticas, como de recursos asignados.

Capítulo 3: La aplicación como entidad

3.1. Introducción

La gran variedad de aplicaciones que corre actualmente en los sistemas multiprogramados va en aumento. Ésto es cierto no sólo en cuanto a los tipos de trabajo que realizan y el paralelismo que expresan, sino también en cuanto a los requisitos que imponen al sistema para tener un rendimiento no ya óptimo, sino aceptable.

En la actualidad, los sistemas operativos de propósito general deben cubrir todos los casos planteables entre aplicaciones. Por este motivo, se ha llegado a una complejidad grande, excesiva, que hace difícil controlar de modo previsible la ejecución de las aplicaciones, complica la depuración de errores y, en definitiva, neutraliza las ventajas de su funcionalidad con la sobrecarga de cálculo y gestión que conlleva.

Esta complejidad en las acciones del sistema no conduce a un rendimiento notablemente mejor y, sin embargo, sí que dificulta o imposibilita al usuario el poder adaptar sus aplicaciones al rendimiento más adecuado para ellas.

Es necesaria una armonía y compenetración entre las acciones de una aplicación, que afecta al rendimiento final de toda ella. Por todo ello, hace falta un modelo que nos permita representar y manejar los objetos y los vínculos que hay entre estos objetos, que los llevan a la consecución de un único fin, el fin de la aplicación.

Poniendo en manos del usuario la posibilidad de sintonizar la planificación de sus aplicaciones, de un modo sencillo, hemos comprobado que el rendimiento del sistema se mantiene, no disminuye [GIL94].

A partir de mecanismos ya existentes hemos desarrollado un entorno de trabajo multiusuario basado en políticas de particionado para la planificación de aplicaciones. De esta manera, cada aplicación se aísla del resto del sistema sin influir ni verse influida por el comportamiento de otras aplicaciones. Es lo que se conocen como políticas de espacio compartido[CROV91].

Apoyándonos en una tecnología microkernel, con parte del sistema trabajando en modo usuario a partir de librerías y servidores, hemos extendido esta parte del sistema que funciona en modo usuario. Así, hemos desarrollado herramientas que permiten gestionar la planificación de flujos a nivel usuario -con un orden menos de magnitud en los tiempos de ejecución que sus homólogas en el kernel- y que pueden ser modificadas por usuarios más experimentados para explotar el rendimiento de sus programas.

No todas las aplicaciones pueden adaptar su concurrencia al paralelismo real que les ofrece el sistema, pero sí que pueden beneficiarse al eliminar un nivel de concurrencia. Así, la aplicación define el número de flujos que le convenga, y el sistema le ofrece realmente paralelismo físico a través de sus procesadores virtuales.

En los siguientes puntos de este capítulo, que consideramos como introductorio del trabajo realizado, exponemos nuestro concepto de aplicación. Este ha sido el fundamento de todas las decisiones tomadas y de la filosofía de trabajo empleada.

Desarrollamos el concepto de aplicación como entidad planificable, desde los programas más sencillos que se realizaron a las aplicaciones actuales más complejas, viendo las diferentes ideas que se tienen de aplicación como entidad en diferentes entornos de trabajo.

Por último, exponemos las aportaciones de nuestra realización, partiendo de la necesidad de contar con un objeto que permita aunar la idea de cooperación que existe en la ejecución de una aplicación paralela.

3.2. Perfil de las aplicaciones a gestionar en un multiprocesador de propósito general.

En nuestro trabajo, hemos partido de los multiprocesadores en cuanto que son utilizados como máquinas de próposito general, en los que pueden correr diferentes aplicaciones, posiblemente 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.

Hay una notable falta de verdaderas cargas paralelas -aplicaciones compuestas por múltiples flujos- de propósito general. Por ello, los bancos de pruebas para medir el rendimiento de las diferentes políticas de planificación, se han basado mayoritariamente en aplicaciones de mucho cálculo. Se han ignorado las cuestiones referentes al comportamiento de las E/S de las aplicaciones y el tiempo de respuesta, que no pueden ser olvidadas en una realización práctica. De hecho, una gestión eficiente en estos campos puede complicar la realización de algunas de las estrategias presentadas, como la planificación de grupos [GUPT91].

Tampoco se han considerado otros aspectos de las aplicaciones que, en trabajos paralelos, adquieren importancia crítica, como pueden ser los mecanismos de sincronización o la frecuencia de comunicación entre trabajos.

Hemos centrado nuestro estudio, trabajo y conclusiones en aplicaciones paralelas multiflujo de granularidad media.

Entendemos por granularidad el “peso” del estado que hay que mantener por cada flujo. Esta granularidad afecta tanto al coste de creación de los flujos como al coste de realizar un cambio de contexto entre ellos.

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).

En concreto, trabajamos con aplicaciones que determinan explícitamente el número de flujos que las forman (aunque sea un número variable en el tiempo) y que los crean y destruyen ellas mismas con primitivas de librerías multiflujo: lo que conocemos como threads de usuario[1].

Este marco de trabajo no significa que para otro tipo de aplicaciones nuestras conclusiones y filosofía de trabajo no puedan ser adoptadas y adaptadas para mejorar igualmente su rendimiento actual[2].

3.3. La aplicación como entidad planificable

En los primeros sistemas, un programa se ejecutaba como una única secuencia, a menudo la única posible, en una máquina. Todos los componentes, tanto lógicos como físicos, implicados en la obtención del objetivo deseado por la aplicación estaban integrados en una abstracción de modo monolítico, equivalente a los primeros sistemas operativos: no existía diferenciación en objetos, funciones o acciones, más allá de las que los lenguajes permitían, consistentes en subacciones

1. Cfr. Capítulo 2 “Concurrencia y Paralelismo”.
2. Ver Capítulo 8 “Conclusiones y Líneas abiertas”.

imbricadas, como mucho.

Con la multiprogramación, aparece la necesidad de caracterizar de alguna manera los trabajos para poder diferenciarlos entre ellos; darles una prioridad, un modo de ejecución, o un lapso de tiempo compartido.

Surgen los multiprocesadores para sistemas de propósito general, y nace la programación paralela. Al igual que en los sistemas operativos se van concretando y escindiendo funciones concretas, módulos definidos, una máquina virtual en la que los componentes son variados y diferenciados (procesadores propios, memoria propia, ...), empieza también a vislumbrarse que en un programa en ejecución pueden definirse acciones que vayan trabajando de un modo más independiente, concurrentemente y, si la máquina lo permite, en paralelo.

A medida que va aumentando esta capacidad, tanto lógica -por parte de los lenguajes y de los sistemas operativos- como física -con los sistemas multiprocesadores de propósito general-, de ejecutar más de una acción simultáneamente -concurrentemente- por parte de la aplicación, se hacen visibles las notables deficiencias en la tradicional planificación de estas acciones por parte del sistema.

Ya no existe una relación equivalente entre todas las acciones que reconoce el sistema como planificables, sino que desde un punto de vista más elevado, hay agrupaciones de ellas, quizá con un cierto orden, una cierta jerarquía, una cierta planificación intrínseca en la que participan y de la que, hasta ahora, el sistema se ha mantenido al margen.

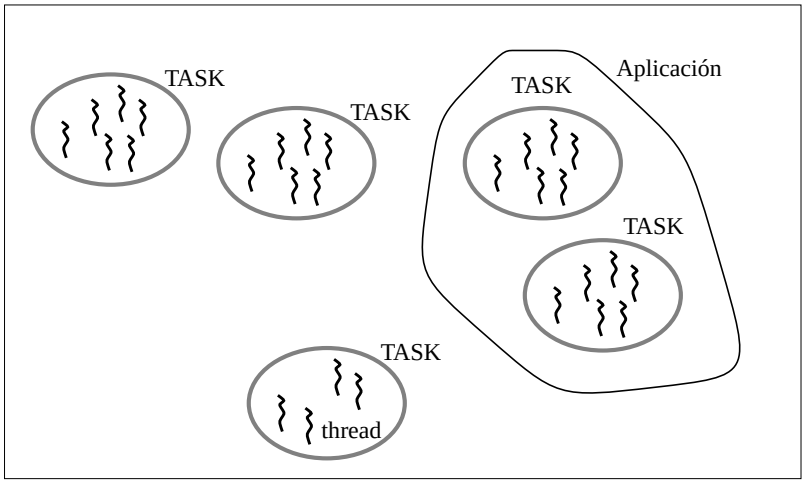

Figura 3-1 Diferentes vínculos de cooperación relacionan trabajos de la misma o diferentes aplicaciones.

Definimos aplicación como el conjunto de acciones realizadas para obtener un fin. En una aplicación paralela, además, varias de estas acciones pueden coincidir en el tiempo. Cada acción tiene su propio objetivo para acercar al total de la aplicación a su resultado. Existe, por así decirlo, un fin corporativo en todas las acciones de una misma aplicación. Por eso, entre las diferentes acciones de una misma aplicación, existen unos vínculos mucho más fuertes que entre acciones de diferentes aplicaciones (Figura 3-1): manipulan (no necesariamente todas, pero seguramente algunas de ellas) los mismos datos o recursos; y cooperan en la obtención de resultados: muchas veces, una acción partirá de la finalización de alguna otra.

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.

Es necesario un nuevo sistema de planificación, con una cooperación entre la aplicación y el kernel. El kernel ha de reconocer y diferenciar los objetos que planifica y que pertenecen a diversas aplicaciones. Para ello, ha de asignar un nuevo recurso a la aplicación, como hace con la memoria física del sistema: los procesadores. De esta manera:

Cada aplicación tiene una submáquina dedicada, en la que sólo pueden ser planificadas acciones suyas, en igualdad de condiciones.

Al no ser objetos sin relación ninguna, sino que todos van a la obtención de un fin, la política de planificación del sistema, general, es frecuentemente ineficiente.

La aplicación es la que sabe hacer avanzar a sus acciones del modo más conveniente. Es ella la que ha de gestionar la planificación de sus flujos y definir las políticas que más se adapten a su perfil de trabajo.

3.3.1. Adaptación de la aplicaciones a la política de planificación

Una de las dificultades que tiene la sintonización del sistema es la de obtener el máximo rendimiento para todas las aplicaciones que componen su carga actual. El hecho de que, además, esta carga se vaya modificando en el tiempo, hace que la sintonización de la planificación del sistema sea compleja y costosa.

Si se pudiera parametrizar la política de planificación del sistema, para ajustarse al perfil de trabajo de cada aplicación aumentaría el rendimiento y se simplificaría a la vez la gestión, demasiado generalista, de los flujos.

En Mach, la existencia de *processor-sets* hace que cada aplicación pueda tener una planificación aislada del resto. Incluso se pueden agrupar aplicaciones por perfiles e independizarlas del trabajo del resto del sistema ya que, asociados a un mismo pset puede haber diferentes tasks (Figura 3-2).

Sin embargo, los parámetros que definen a cada una de las políticas posibles (tiempo compartido y prioridades fijas, actualmente) son globales a todo el sistema.

Nosotros hemos aprovechado esta circunstancia para proponer una parametrización de la política de planificación del sistema, no sólo ajustable por el usuario, sino propia de cada aplicación. De esta manera:

Cada aplicación puede definir los valores de *quantum* asociados a cada prioridad y el intervalo de prioridades entre las que se mueven sus flujos.

Así se consigue una adaptabilidad mayor en la planificación de sus flujos al trabajo y utilización de los recursos que ha de llevar a cabo [GIL94].

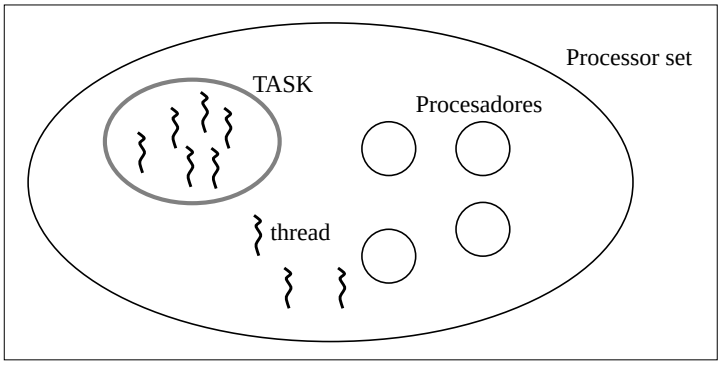

Figura 3-2 Mach permite agrupar threads bajo una misma unidad planificadora: el *processor-set* (pset). En concreto, pueden agruparse tasks con todos los threads que las formen.

3.4. Composición de la aplicación como objeto

Cabe ahora preguntarse si la aplicación es sólo la suma de diferentes acciones y recursos, o si tiene una entidad propia "agrupadora" de ciertos recursos y acciones sobre ellos.

Ciertamente, la aplicación lleva una gestión de distribución de recursos, más que de planificación entendida como a corto plazo: es la que determina la "velocidad" de avance de un grupo determinado de tareas, a medio plazo. Es la responsable del uso de los recursos del sistema, pero no los utiliza ella directamente, sino que los distribuye adecuadamente, entre las acciones que trabajan para ella. Podemos decir que, como entidad globalizante, es un objeto pasivo.

3.4.1. La task

Nos apoyamos en el modelo y objetos que ofrece Mach, por considerarlo suficientemente general dentro del tipo de aplicaciones que consideramos. Desde el punto de vista estructural, la aplicación es, en primer lugar, un conjunto de tasks. Cada task, es la que recibe y planifica los recursos para determinadas acciones a las que engloba y que poseen una comunicación entre ellas más frecuente, o más estrecha.

Es por esto que, entre diferentes tasks de una misma aplicación, puede haber diferentes vínculos, más o menos estrechos, o incluso nulos; y vínculos, en cualquier caso, más lejanos que entre las acciones de una misma task (lo que no implica que tenga necesariamente que haber una relación explícita de sincronización o comunicación entre cualesquiera dos acciones de una misma task: ejemplo de sincronizaciones tipo barrera).

Tampoco es, pues, una task, simplemente la agrupación de determinadas acciones.

3.4.2. Visibilidad de los datos y semántica de la abstracción de flujo

Más importante aún, es ver si todos los recursos de una acción le llegan a ésta a través de una task, o, al contrario, han de existir objetos propios de una acción, además de los que propiamente la definen, como son el contenido de sus registros y pila (estado hardware). Tradicionalmente, se ha ofrecido ejecución a las aplicaciones a través de la abstracción de proceso, que definía un flujo en ejecución más todo su entorno y sus recursos; entre ellos, la memoria con los datos a los que accedía.

Una aplicación paralela que defina sus flujos a través del modelo de proceso, se encuentra

con el impedimento de no poder acceder “cómodamente” a datos comunes: la protección impuesta alrededor del proceso, se vuelve en este caso en su contra. Sólo determinadas versiones de determinados sistemas operativos permiten zonas de datos comunes a los diferentes procesos, y no de manera automática[3].

Ahora, con el modelo de task y threads, se ha escindido totalmente esta semántica de visibilidad, yéndonos al polo opuesto: todos los recursos son visibles y accesibles a todos los flujos de ejecución pertenecientes a una misma task, por definición [ARAL89]. La diferencia en el acceso a memoria de los dos modelos expuestos puede verse en la Figura 3-3.

A veces, sin embargo, es necesario que un flujo tenga información propia suya. En las realizaciones actuales, ésto sólo es posible a través de su propia estructura de información (Thread Control Block) o su pila. Para no limitar el tamaño y tipo de estos datos privados, se accede la mayoría de veces a través de una doble indirección (puntero a una zona de memoria que hay que pedir) y no de manera automática. Ésto supone añadir complejidad a la programación y limitaciones, además de menor grado de transportabilidad de un tipo de procesador virtual a otro (por ejemplo de proceso UNIX a thread de kernel). Se puede destruir, en el peor de los casos, la propia semántica de la aplicación.

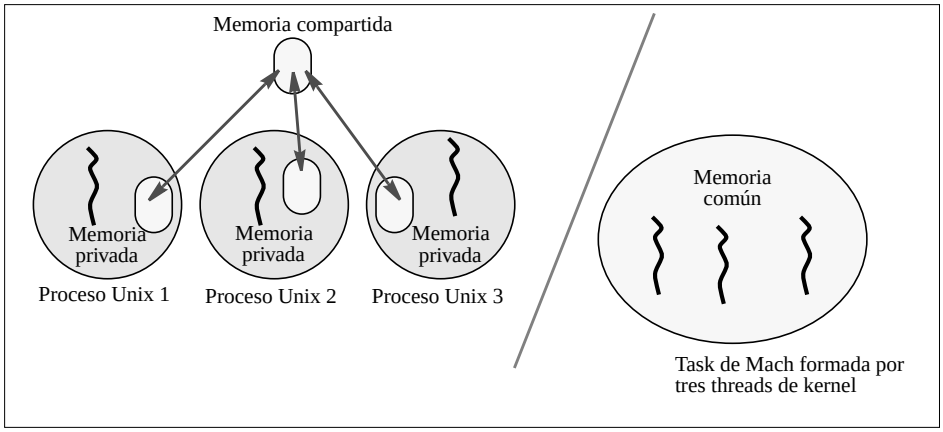

Figura 3-3 Cambio en la visibilidad de los datos al pasar de un modelo de proceso a un modelo de threads de kernel.

Por otra parte, de cara a la portabilidad de aplicaciones a diferentes entornos de trabajo, hay que tener en cuenta que eliminar el espacio de direcciones privado es incompatible con la semántica que establece UNIX [EDLE88].

3.5. Desarrollos actuales del concepto de aplicación

Existe hasta el momento una gran diversidad de opiniones y modelos para definir lo que es una aplicación; más, en concreto, una aplicación paralela. No obstante, de manera difusa e implícita, son muchos y buenos los trabajos realizados en los que el concepto de aplicación aparece como mucho más amplio y complejo que el de proceso o *job.*

Vamos a ver a continuación algunas aportaciones, en cuanto a funcionalidad y objeto, en este campo.

3. Por ejemplo, las versiones más actuales de UNIX System V, con sus llamadas al sistema para definir y trabajar con memoria compartida, mapeada en el espacio de direcciones de diferentes procesos.

3.5.1. La aplicación como conclusión de diferentes modelos de programación: Psyche

En la actualidad, existen ya variados entornos de ejecución, proporcionados por lenguajes y librerías, que se han desarrollado como soporte a la programación paralela. Cada entorno proporciona su propio modelo de paralelismo, básicamente, a través de la abstracción de proceso que define y del mecanismo de comunicación entre dichos procesos que utiliza [MARS92].

Abstracciones típicas de procesos con las que se trabaja hoy día son las corrutinas, los procesos ligeros que se ejecutan hasta acabar, los procesos ligeros que pueden bloquearse a nivel de usuario, los procesos ligeros definidos por el propio kernel en un mismo espacio de direcciones y los procesos, en el sentido tradicional de la palabra, entendidos como procesos pesados o al estilo de UNIX. Mecanismos de comunicación entre procesos conocidos y utilizados en estos entornos incluyen mensajes, síncronos o asíncronos, canales unidireccionales o bidireccionales, RPC, *buffers* globales, y espacios de direcciones compartidos con mecanismos como monitores, semáforos o esperas activas.

Cada modelo de programación incluye una determinada asunción sobre la granularidad y frecuencia de la comunicación que habrá entre sus procesos, la sincronización, el grado de concurrencia deseado y la necesidad o no de protección. Así, los diferentes entornos existen como intención de acomodarse a los requisitos particulares las aplicaciones de un determinado dominio, sin existir claramente un modelo que satisfaga todos.

La programación paralela con múltiples modelos consiste en la ejecución simultánea de múltiples modelos de programación dentro de un mismo programa o entre varios. Al seleccionar los algoritmos que forman parte de una aplicación, no existe la limitación de vincularse a un único modelo.

Aplicaciones de esta clase, como robots, que incluyen parte gráfica, parte de tiempo real, sistemas expertos, etc... pueden descomponerse por funciones independientes. Cada una de ellas se diseña eligiendo el modelo de programación que mejor se adapte. Cuando estos módulos se integren en una única aplicación, la comunicación entre ellos rompe igualmente cualquier frontera entre modelos de programación.

3.5.2. La aplicación como descomposición de un problema

Los entornos de programación paralela y las aplicaciones, deben mantener un delicado equilibrio entre el coste y los beneficios que acarrea el paralelismo. Este coste incluye la sobrecarga de la gestión de procesos, la sincronización y la comunicación. Cualquier cambio significante en alguno de estos factores, afecta a la decisión sobre la granularidad de paralelismo apropiada para una determinada aplicación.

En la primera mitad de la década de los 80, la sobrecarga en la gestión de procesos era un factor dominante para decidir cómo descomponer una aplicación. En un intento de dar soporte al paralelismo, Mach diferenció las abstracciones de kernel referentes al espacio de direcciones y a los flujos de control, de modo que un programa paralelo pudiera construirse utilizando múltiples flujos de control compartiendo un mismo y único espacio de direcciones [TEVA87].

Para evitar el coste de entrar en el kernel para las operaciones con flujos, aparecieron varios paquetes de threads que los manipulaban en el espacio de usuario. Estas librerías representan una extensión en la funcionalidad del propio sistema operativo en el espacio de usuario, por motivos de rendimiento, hasta el punto de haberse conseguido en la actualidad operaciones de flu-

jos por un tiempo de sólo una orden de magnitud más que una llamada a procedimiento [COOP88].

Otro tema concerniente a nivel de aplicación y en la realización de threads es la sincronización, dado que cuando la creación de flujos se abarata, permite programación paralela de granularidad más fina, obligando a un mayor número de puntos de comunicación. El desarrollo de primitivas de sincronización eficientes y escalables reduce dramáticamente la sobrecarga de este factor, que ha llegado a ser nula en la mayoría de aplicaciones.

Cuanto más fina es la granularidad de los flujos, más rápidamente finalizan el trabajo que se les ha asignado, aumentando por consiguiente la posibilidad de dejar al respectivo procesador sin trabajo. Es generalizado el uso de un única cola de repartición de trabajos para evitar el posible desbalanceo de la cargo de los procesadores, aún a costa de que se aumente la sobrecarga de sincronización para acceder a ella en exclusión mutua.

Como resultado de estas directrices en el software, muchas aplicaciones paralelas en multiprocesadores de memoria compartida se escriben utilizando un modelo de programación de memoria compartida.

Se utilizan procesos ligeros para representar una granularidad fina de paralelismo; se almacenan los datos comunes en un espacio de direcciones compartido; se utilizan primitivas eficientes de sincronización para el acceso a referencias no locales y se evita el desbalanceo en la carga de procesadores mediante una cola global de trabajo pendiente.

El único coste que no queda controlado es actualmente el de la comunicación -tanto entre procesadores, como entre procesadores y memoria-. Esto es especialmente delicado en arquitecturas NUMA, para las cuales se proponen algoritmos de planificación de flujos que tengan en cuenta la posición de los datos con los que va a trabajar el flujo: Memory Conscious Scheduling (MCS) [MARK93].

3.5.3. La aplicación como programación multiflujo: Mach

Con los cambios aparecidos en las arquitecturas multiprocesador y en la programación, aparece la necesidad de nuevas propuestas para la gestión de recursos. Técnicas como la programación paralela y multiflujo, modifican la utilización del recurso procesador al emplear más de una entidad de ejecución independiente (proceso, thread) en una misma aplicación. Esta relación de cooperación dentro de la aplicación choca con la noción tradicional de tiempo compartido, según la cual todas las entidades planificables compiten de modo independiente [BLAC90].

Los cambios en los requisitos de las aplicaciones motivan una mayor investigación en las áreas de planificación y gestión de los procesadores y de los accesos no uniformes a memoria, donde confluyen los problemas de mayor entidad por los avances en el diseño y utilización de multiprocesadores.

El conjunto de problemas involucrados en la gestión de procesadores, o planificación,

puede caracterizarse según el tipo de aplicaciones para el que se manifiesta dicho problema.

Algunas aplicaciones paralelas necesitan trabajar con procesadores dedicados para obtener de ellas un rendimiento aceptable; y no permiten un modo de trabajo en compartición con otros trabajos del sistema (políticas tradicionales de tiempo compartido).

Un segundo tipo de aplicaciones trabajan con más flujos planificables que el número de procesadores de los que disponen, y necesitan soporte por parte del sistema para que la sincronización y comunicación entre los flujos sea eficiente.

Seguir manteniendo las aplicaciones de tiempo compartido, que trabajan en los sistemas y mantienen un peso importante, en integración con estos nuevos tipos de aplicación, es otro punto a tener en cuenta para la gestión de los procesadores.

Los requerimientos de planificación de programas paralelos y concurrentes son una consecuencia derivada de su clasificación en modelos de programación. Esta clasificación se basa en la concurrencia que los diferentes modelos introducen (si es que la introducen) al paralelismo básico ofrecido por el hardware.

Un SO puede introducir concurrencia proporcionando entidades independientes para planificar que multiplexen en los procesadores físicos del sistema; es lo que conocemos como procesadores virtuales y que pueden ser procesos UNIX, thread de Mach, etc.

Un lenguaje o una librería de usuario introducen más concurrencia multiplexando flujos, que mantiene el propio nivel de usuario como entidades independientes y planificables, sobre estos procesadores virtuales.

Cuatro clases emergen cuando estos modelos se clasifican según el uso de concurrencia que hagan, pasando del paralelismo puro, en el que no existe concurrencia alguna tras el paralelismo ofrecido por el hardware a modelos de concurrencia dual en la que se aprovechan de la concurrencia a los dos niveles: usuario y sistema[4]. A partir de esta clasificación de programas concurrentes y paralelos, aparecen dos primeros requisitos de planificación:

- Los modelos que asumen una disposición de procesadores tal que puedan trabajar en paralelo, necesitan la garantía de esta disponibilidad por parte del sistema.
- Los modelos que trabajan concurrentemente (por ejemplo, más flujos de usuario que procesadores), necesitan que el sistema les proporcione soporte para gestionar esta concurrencia eficientemente (en concreto y especialmente para la sincronización y comunicación).

3.5.4. El futuro de las aplicaciones de alto rendimiento

Está abierto el debate sobre el rango de aplicaciones que se han de considerar para sistemas de programación masivamente paralelos (MPP). y de él han sido las ideas recogidas en el proyecto HPCC[5].

Se puede argüir que la programación paralela, hacia un alto rendimiento, necesita vínculos progresivamente mayores entre las aplicaciones y los sistemas (arquitecturas y sistemas operativos).

Esta utilización ventajosa de una máquina paralela requiere un mapeo (*matching*) de la

4. Cfr. Capítulo 2 “Concurrencia y Paralelismo”
5. High Performance Computing and Communications, USA

aplicación con la máquina y, por tanto, o bien el usuario “entiende” la máquina (por ejemplo, un modelo de Fortran con paso de mensajes), o bien el sistema entiende, conoce y se adapta a los requisitos de la aplicación.

En otro orden de discusión, situados en entornos de programación, las aplicaciones están inherentemente limitadas por las capacidades que les proporciona el entorno en el que se desarrollan. Así pues, una manera de analizar los requisitos de una aplicación para High Performance Computing es examinar los requisitos asociados a la infraestructura de los entornos software.

Contemplando los interfaces de usuario de sistemas paralelos, se observa una gran incertidumbre. La programación paralela y su representación a partir de un lenguaje de programación es complicada para gestionar eficientemente los recursos subyacientes más modestos. Los programadores van a soluciones “a mano”, muchas veces recurriendo al ensamblador. Se imposibilita así la portabilidad y se limita el desarrollo de software e incluso el tiempo de vida de las aplicaciones, demasiado corto para el esfuerzo que conllevan.

En el otro extremo, hay lenguajes de programación paralela que permiten expresar explícitamente el paralelismo pero no permiten la intervención del usuario en la gestión de recursos. Se distancia, con ello, al programador de los mecanismos de control que podrían ser esenciales en la optimización del rendimiento.

Prevalece una construcción conservadora por la inversión que supone realizar un nuevo entorno de programación y la baja probabilidad de que sea ampliamente aceptado por la comunidad de usuarios: a veces la curva de aprendizaje es un obstáculo insalvable.

La mayoría del trabajo en conseguir un mejor rendimiento para una aplicación se emplea en los detalles de decisión dependientes de la máquina. Queda abierta la cuestión sobre hasta qué punto la estructura de la máquina y el modelo de ejecución debe afectar al algoritmo en orden a obtener eficiencia.

Está claro que hace falta un mayor conocimiento sobre requisitos de las aplicaciones en un entorno masivamente paralelo. Sin él es francamente incierto que los desarrollos a corto plazo del software se dirijan adecuadamente a las necesidades más importantes. Es necesario saber cómo deberían evolucionar los sistemas para dar soporte a las necesidades de las aplicaciones.

3.6. Aportaciones de este trabajo

La dedicación de procesadores a aplicaciones ya ha sido estudiado y presentado en múltiples trabajos [CROV91] como la política de planificación más eficiente en sistemas multiprocesadores para aplicaciones paralelas. Incluso la tendencia va hacia el mapeo 1-1 entre procesadores virtuales y procesadores físicos, ya que los cambios de contexto entre procesos ligeros son más eficientes que entre procesos UNIX tradicionales. Pero aún serían más eficientes sin la intervención del kernel [MARK92], [GOTT92], [EDLE92], [WELC92], [ROSE92].

Dado el gap creciente que se da hoy en día entre las velocidades de los elementos de cálculo y las velocidades de los elementos de comunicación, es importante dar a la aplicación la oportunidad de gestionar su planificación basándose no sólo en el conocimiento de su trabajo (acciones que realiza), sino también en el conocimiento del hardware determinado (cuánto y cuál) que le otorga el sistema.

Nosotros nos hemos limitado únicamente al recurso procesador. Pero pensamos que sería interesante abrir la gestión y conocimiento hacia otros recursos.

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.

En un entorno multiusuario es posible obtener un mayor rendimiento de las aplicaciones paralelas multiflujo partiendo de políticas de planificación de tiempo compartido. Conseguimos, así, aislar unas aplicaciones de otras en cuanto a los recursos asignados por el sistema.

De este modo:

- Evitamos la influencia que pudiera haber en el mal comportamiento de una aplicación, en la ejecución del resto.
- Damos a las propias aplicaciones la posibilidad de aprovechar la localidad de los datos y su reutilización, según políticas intra-aplicación de afinidad a los procesadores.

A nivel de usuario, hemos trasladado al nivel de aplicación -en concreto, hemos diseñado nuevas librerías- políticas y mecanismos de planificación que hasta ahora residían en el kernel, eliminando de éste toda la planificación de flujos.

Era importante a este nivel, para mantener la misma funcionalidad que ofrecía la planificación del sistema, contar con una mecanismo de preempción 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. Es lo que ocurre en alguna de las librerías actuales realizadas sobre UNIX, que recuperan el control para planificar a partir de los *signals* que ofrece; esto hace que, no sólo las aplicaciones no puedan utilizar este mecanismo, sino que las aplicaciones que lo hacen servir, no sean transportables. Para ello hemos realizado upcalls del kernel a la aplicación, permitiendo entornos de tiempo compartido totalmente gestionados a nivel de usuario.

Por último, hemos diseñado al completo un sistema de planificación basado en el conocimiento 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 “contextos de ejecución” 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.

Ello no implica que el usuario quede ahora desprovisto de todo el soporte que antes, de manera transparente, le proporcionaba el sistema.

> Siguiendo con la filosofía de los microkernels, hemos extraído del kernel el módulo de planificación de procesadores virtuales (las políticas), ofreciéndoselo al usuario en forma de librerías.

Estas librerías pueden ser utilizadas por los usuarios menos experimentados o que no necesiten de una adaptación concreta y exclusiva para el rendimiento de sus programas. Sin embargo, al tener la librería dentro de sí los nuevos mecanismos de planificación, esos usuarios los estarán utilizando de forma implícita y para ellos transparente, y por estar realizados a nivel de usuario, serán ahora más eficientes.

En cambio, el usuario más experimentado podrá sacar más partido de determinados aspectos del soporte a la planificación que se le ofrece en nuestro entorno, adaptándolos al perfil de su propio trabajo. Por el hecho de estar a nivel usuario, permite también que se pueda modificar con facilidad.

3.7. Referencias y Bibliografía

[ARAL89] "Variable Weight Processes with Flexible Shared Resources"
Ziya Aral, James Bloom, Thomas Doeppner, Ilya Gertner, Alan Langerman, Greg Schaffer
Proceedings on the USENIX Conference, S. Diego CA, pp 405-412, January 1989.

[BLAC90] "Scheduling and Resource Management Techniques for Multiprocessors"
David L. Black
PhD Thesis, Carnegie Mellon University, July 1990.

[COOP88] "C Threads"
Eric C. Cooper and Richard P. Draves
CMU-CS-88-154, School of Computer Science, Carnegie Mellon University, June 1988.

[CROV91] "Multiprogramming on Multiprocessors"
M. Crovella, P. Das, C. Dubnicki, T. LeBlanc and E. Markatos
Third IEEE Symposium on Parallel and Distributed Processing, December 1991
Technical Report 385, Computer Science Department, University of Rochester New York, February 1991.

[EDLE88] "Process Management for Highly Parallel UNIX Systems"
Jan Edler, Jim Lipkis and Edith Schonberg
Proceedings on the USENIX Supercomputer Workshop, pp 1-17, September 1988.

[EDLE92] Jan Edler
Comunicación personal, Junio 1992.

[GIL94] "Towards User-level Parallelism with Minimal Kernel Support on Mach"
Marisa Gil, Toni Cortés, Angel Toribio, Nacho Navarro
DAC/UPC Report RR-94/07, 1994.

[GOTT92] Allan Gottlieb
Comunicación personal, Junio 1992.

[MARK92] Evangelos Markatos
Comunicación personal, Junio 1992.

[MARK93] “Scheduling for Locality in Shared-Memory Multiprocessors”
Evangelos Markatos
Ph.D. Thesis
University of Rochester, New York, 1993.

[MARS92] “Multi-Model Parallel Programming”
Brian D. Marsh
Ph.D. Thesis
University of Rochester, New York, January 1992.

[ROSE92] Bryan Rosenburg
Comunicación personal, Junio 1992.

[TEVA87] “Mach Threads and the Unix Kernel: The Battle for Control”
Tevanian, Rashid, Golub, David L. Black, E.C.Cooper and Young
USENIX Association Summer Conference Proceedings, June 1987.

[WELC92] Brent Welch
Comunicación personal, Junio 1992.

4

Mecanismos y políticas fuera del kernel

“Some people see things as they are and they say: why?
I dream things that don't exist and I say: why not?"

John F. Kennedy

ABSTRACT: La creación y gestión de flujos a nivel de usuario, a través de librerías, disminuye dramáticamente los tiempos de manipulación con dichos objetos.
El hecho de estar a nivel de usuario permite que el código pueda ser modificado por usuarios expertos y adaptado a las necesidades de la aplicación en concreto que la esté utilizando; aunque esta facilidad no se ha explotado todo lo que se podría hasta el momento.
En este capítulo presentamos una nueva librería, basada en el paquete de CThreads, a la que se le han añadido todas las políticas y mecanismos que hasta hoy se han realizado en el kernel de Mach. Se hace primero una introducción de la librería original de CThreads, para pasar a continuación a explicar las modificaciones realizadas.
Se presenta un juego de pruebas simulando el comportamiento de un servidor y se evalúa su rendimiento con ambas librerías: CThreads y CThreads^{+}.

Capítulo 4: Mecanismos y políticas fuera del kernel.

4.1. Introducción

El kernel es un entorno excesivamente protegido para planificar tareas que persiguen un mismo objetivo y comparten muchos de los objetos y recursos. Sus políticas son excesivamente generales para los perfiles peculiares de las aplicaciones paralelas actuales.

No existen hoy en día paquetes de usuario con facilidades adecuadas para la gestión de flujos, como pueden ser mecanismos de cambio de contexto entre flujos y políticas de planificación *round-robin*, o por prioridades. Sí la intención de desarrollarlos y definir interfaces de usuario más potentes, entre ellos, Pthreads[6] [IEEE92].

Estas realizaciones, mayoritariamente en experimentación, están diseñadas para sistemas monoprocesadores, asumiendo un único procesador físico [POWE91], o bien con otro tipo de restricciones, demasiado fuertes para poder obtener un rendimiento equivalente al de esos mismos mecanismos ofrecidos por el kernel. Por ejemplo, en entornos UNIX, el caso más común es utilizar la programación del alarm() como reloj a nivel de usuario, provocando un fallo en la ejecución de la aplicación si el programador desconoce ese funcionamiento por parte de la librería [DOEP87].

No se trata, con nuestra realización, de despojar al usuario de la sofisticación alcanzada por los sistemas actuales en la planificación de flujos. A partir de un estudio detallado de los mecanismos que ofrece el kernel, y de los que ofrecen los paquetes más sofisticados de threads, hemos subido la gestión de flujos del kernel al nivel de usuario, con una mejora considerable en los tiempos de ejecución de la aplicación y construyendo un interfaz de usuario totalmente general, para cualquier sistema y cualquier aplicación, sobre cualquier subsistema.

Nuestros cambios se han centrado básicamente en tres frentes: ofrecer al usuario la posibilidad de ordenar sus trabajos de manera explícita, dotándoles de una prioridad relativa a la aplicación; ofrecer primitivas sencillas, con un trabajo concreto, que permitan políticas de planificación diversas a las que puedan acogerse las diferentes aplicaciones; por último, utilizar en lo posible las primitivas de planificación realizadas, dentro de la propia librería, para mejorar el rendimiento de la sincronización y gestión de los flujos, de una manera transparente al usuario.

En los siguientes apartados vamos a ver el diseño de algunas librerías en experimentación para mejorar el trabajo con flujos gestionados a nivel de usuario, centrándonos en la política de planificación. A continuación, daremos una descripción del paquete de CThreads, sobre el que hemos realizado nuestras modificaciones.

En los apartados 5, 6 y 7 explicamos nuestras modificaciones en el campo de las políticas y mecanismos de planificación, nuestro objetivo y nuestras decisiones de diseño. Estas mejoras se han aplicado también en la programación de las funcionalidades de comunicación y sincronización ya existentes en la librería.

Acabamos con algunas medidas tomadas con diferentes utilidades de planificación de flujos. Las diferencias de rendimiento obtenidas para un mismo trabajo, dependiendo del diseño utilizado, nos han permitido proponer ciertas recomendaciones en la construcción y trabajo con aplicaciones paralelas y concurrentes.

4.2. Algunas políticas ya existentes a nivel de usuario

Los flujos que se crean en la aplicación, a través de librerías, pueden trabajar sobre diversos tipos de procesador virtual: procesos o threads de kernel, principalmente. El suponer uno de los dos tipos influye en los objetivos de diseño de la librería y en su propia construcción.

6. Pthreads es un acrónimo de POSIX 1003.4a threads.

CThreads, por ejemplo, se planeó inicialmente para ayudar al usuario en la programación de threads de kernel[7] [DUCH91]. La relación thread de usuario-thread de kernel era 1-1 y por tanto, no se asumía concurrencia a nivel de usuario: cada vez que se creaba un thread de usuario, equivalía a crear un thread de kernel. En la actualidad, este diseño ha cambiado, como veremos en el siguiente apartado.

Otros paquetes de threads, ofrecen concurrencia en aplicaciones que, de otro modo, lo tendrían costoso, como ocurre en entornos en los que los procesadores virtuales son pesados, como pueden ser los procesos UNIX.

En el primer caso, las librerías no están preparadas con mecanismos ni funcionalidades demasiado ricos.

En el segundo, se conoce y asume que la máquina virtual sobre la que trabajan es monoprocesador y se "emula" un kernel a nivel usuario de tipo monolítico, que facilita la gestión de concurrencia. Incluso, la mayoría de las veces se asume que el entorno operativo de trabajo es UNIX y se utilizan, de manera transparente al programa, utilidades de usuario, que pueden llevar a comportamientos erróneos a las aplicaciones. Es el caso de la librería Threads desarrollada en la Brown University por Thomas Doeppner [DOEP87].

Últimamente también se están desarrollando algunas herramientas (*toolkits*) para que el usuario pueda definir sus propias rutinas de planificación. La potencia y versatilidad de las mismas depende de la arquitectura sobre la que se trabaje. En esta línea están los QuickThreads [KEPP93].

4.3. Entorno de desarrollo: CThreads

El paquete de *threads* de usuario en el que hemos realizado nuestras modificaciones es CThreads [COOP88], que permite la realización de programas en C con múltiples flujos de control, sobre el kernel de Mach [TEVA87]. Aunque existe la posibilidad de ligar de modo permanente un flujo a un thread de kernel[8], el comportamiento por defecto es que pueda haber cambios de contexto a nivel de usuario entre los diferentes procesadores virtuales y flujos definidos en la aplicación [GIL93].

Todo flujo tiene un identificador propio en la librería, que se devuelve al padre después de la llamada **cthread_fork()** y que puede ser consultada por el propio flujo con la función **cthread_self()**. Este identificador de *thread* es único en el tiempo pero reutilizable, por lo que el propio programador debe ser cuidadoso en el control de sus flujos, y asegurarse de no referenciar a uno que ya ha terminado.

Todos los *threads* de un entorno pueden trabajar concurrentemente, existiendo la posibilidad de sincronizarse con la terminación de un *thread* para recoger su resultado (**cthread_join()**), o bien que un *thread* se desligue totalmente del resto (**cthread_detach()**). La planificación de flujos que hace el paquete es FCFS (First Come, First Served), y no se desbanca a un flujo hasta que éste se bloquee voluntariamente, o cede el procesador virtual explícitamente con la llamada a **cthread_yield()**. En ambos casos, el flujo se halla en un punto seguro y adecuado para un cambio de contexto.

Todas las variables globales y las declaradas como estáticas son compartidas entre todos los flujos. Las variables propias de cada *thread* se crean y acceden a través de **cthread_set_data()** y **cthread_data()**.

Las primitivas ofrecidas para sincronización y exclusión mutua se basan en la existencia de variables de tipo *mutex* y *condition* y, básicamente, implementan la funcionalidad de los moni-

7. En Mach 2.5, versión monolítica con el código de 4.3BSD incluido en el kernel.
8. Existe también la operación contraria para deshacer el vínculo entre ambos.

tores.

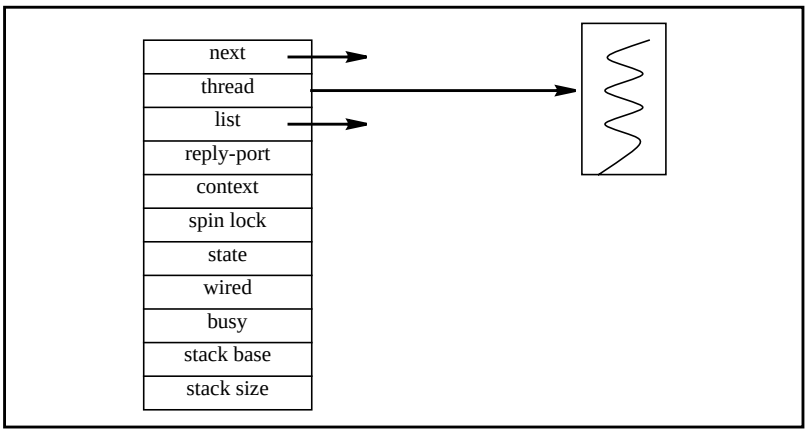

Figura 4-1 Esquema de la estructura de datos que se mantiene por cada proceso ligero.

El mecanismo de proceso ligero (*lightweight process* o *thread*) es realizado por la librería a partir de unos objetos no visibles al programador, los cprocs (Figura 4-1). Para cada flujo de ejecución del usuario, define un cproc al que luego le asignará dinámicamente, cada vez que se pueda ejecutar, un procesador virtual (*thread* de kernel).

Cada cproc tiene asignada su propia pila, y es precisamente a través de ella como puede identificarse el flujo de usuario que se está ejecutando en ese momento[9] (Figura 4-2).

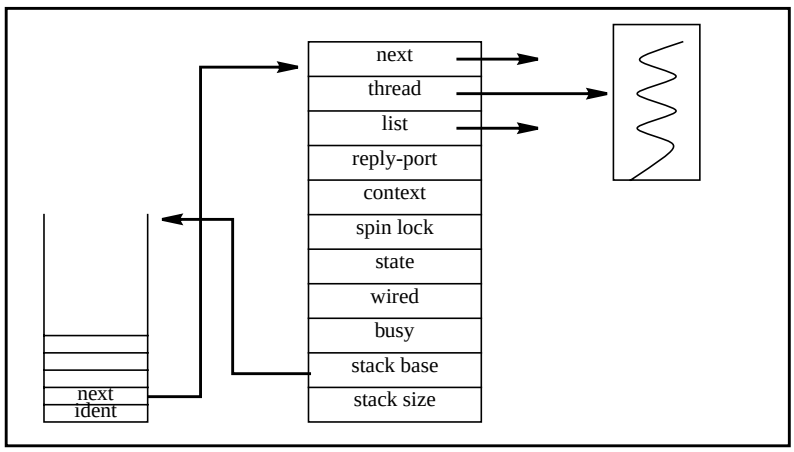

Figura 4-2 Relación entre la estructura de datos y la pila de un proceso para su propia identificación.

Aunque en principio no está limitado el número de flujos ni de procesadores virtuales que pueden estar activos en un momento dado, puede controlarse el nivel de paralelismo y de concurrencia definiendo unos máximos a partir de las llamadas **cthread_set_limit()** y **cthread_set_kernel_limit()**, respectivamente.

Esta limitación es totalmente transparente al funcionamiento de los flujos de usuario[10], ya que se ha separado la declaración e identificación del flujo que se pide ejecutar del proceso ligero

9. Todos los *threads* de usuario están viendo la misma memoria y las mismas estructuras de datos. Lo único que es propio de cada uno es precisamente su puntero a la pila, ya que es exclusiva de cada uno. Es usual en los paquetes de *threads* de usuario el poder identificarlos a partir de su *stack*.
10. Aunque no a su rendimiento, claro.

que lo ejecuta, diferenciando entre la estructura cthread y la estructura cproc, respectivamente (Figura 4-3). Un proceso ligero lo forma la unión de un cthread con un cproc. El número de cthreads es ilimitado y una vez alcanzado el máximo número de cprocs permitido, los cthreads que se vayan creando, quedarán encolados hasta que quede algún cproc disponible. Es un modelo de trabajo de *task queues*.

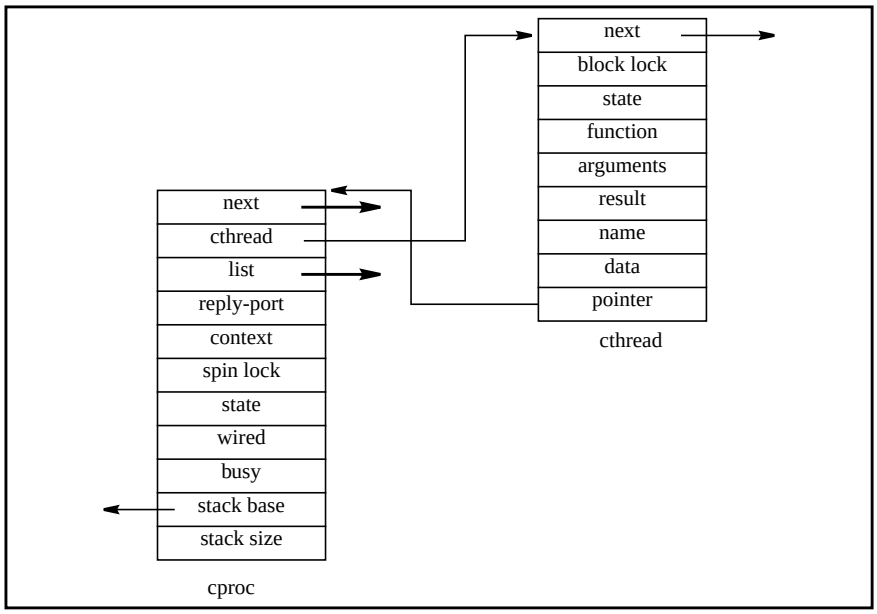

Figura 4-3 Separación de las estructuras de datos del proceso ligero en contexto (cproc) y programa (cthread).

La separación del proceso ligero en dos estructuras permite, además de la limitación de recursos, la reutilización de elementos. Por ejemplo, en un bucle para operar en un *array*, donde para cada iteración se crea un *thread* que ejecute las instrucciones del interior se observó que, para 100 iteraciones, se creaban una media de 15 a 26 cprocs, debido a la corta duración de los flujos (granularidad muy fina).

Este funcionamiento es muy interesante, ya que son precisamente los flujos de granularidad más fina (habitualmente, iteraciones de bucles) los que con más facilidad se expanden (se puede trabajar con bucles de miles de iteraciones en cálculo de estructuras, en dinámica de fluídos,...).

4.3.1. Establecimiento del vínculo proceso ligero-procesador virtual

Para que un flujo se ejecute realmente, su código tiene que ser seleccionable por un procesador físico. Para ello, la librería realiza dos pasos: primero, formar un proceso ligero uniendo un cproc y un cthread; después, asignar el *thread* a un procesador virtual.

Para que un procesador virtual ejecute el flujo de usuario específico, se mapean las estructuras de datos del flujo en el procesador virtual: se actualiza el contenido de los registros para indicarle el código y la pila con que ha de trabajar. Esto se lleva a cabo mediante las llamadas al kernel **thread_get_register()** y **thread_set_register()**, para coger los registros y actualizarlos. A partir de este momento, cuando el procesador seleccione a este procesador virtual, pondrá al *thread* en ejecución.

La asociación flujo-procesador virtual puede ir cambiando durante la vida del flujo, bien a través de cambios voluntarios con la rutina **cthread_yield()** o bien por bloqueos, ya sea en la

recepción de mensajes o por sincronización a nivel de usuario. La asociación cproc-cthread, sin embargo, persiste hasta que finalice la ejecución del flujo.

4.3.2. Tratamiento de bloqueos a nivel usuario

Para un *thread* que se encuentra en un punto seguro y quiere hacer cesión voluntaria de su procesador virtual, existe la llamada a la librería **cthread_yield()**. Si no existe un *thread* para seleccionar, llama al kernel para que se plantee la posibilidad de planificar a otro procesador virtual en el físico.

Cuando el bloqueo viene provocado por la espera de un servicio de la propia librería (por ejemplo, en el envío de un mensaje o en la espera de una exclusión mutua), el camino de cesión del procesador virtual que se sigue es diferente. Si no existe un flujo a quien cederle el procesador virtual, el *thread* entra en estado de espera. Esto consiste en coger un *thread* degenerado (*waiter*) y quedarse en un bucle de espera[11], hasta que aparezca algún *thread* disponible para ejecutar, posiblemente el mismo que pidió el servicio. La espera la realiza con soporte del sistema operativo, forzando un bloqueo en el kernel, a través de mensajes. En este caso, no se cede el procesador físico a otro procesador virtual.

4.3.3. Robustez de la librería y opciones de compilación

Existen muy pocas comprobaciones en el código sobre la validez o corrección de los parámetros que se pasan a las rutinas: en ningún momento se verifica que realmente “exista” el *thread* sobre el cual hay que operar. Esto puede ser un problema importante si se tiene en cuenta que la identificación de un elemento es la dirección de su estructura de datos, y ésta es reutilizada una vez que acaba la ejecución del flujo.

La librería ofrece determinadas opciones de compilación para optimizar el rendimiento y la robustez de la librería, así como para permitir un cierto seguimiento de la ejecución, que pasamos a comentar brevemente.

El tamaño de la pila es el mismo para todos los *threads* de la aplicación y se hereda del primer *thread* del sistema[12]. Si se quiere proteger a la aplicación del *overflow* de una pila, se ha de compilar con la opción RED ZONE; de esta manera, se añade a cada pila una página protegida contra escritura, lo que permite al hardware avisar cuando se llega al límite. En caso contrario, no se hace ninguna comprobación. Esta unidad de protección de la pila es excesivamente grande (una página en los sistemas actuales suele ser del orden de 4 kbytes); pero puede medirse y ajustarse el tamaño de la pila mientras se depura el programa, y después suprimir esta comprobación.

Para saber qué rutina de la librería se está ejecutando en cada momento, la opción TRACE, en la entrada y salida de las rutinas, envía los correspondientes mensajes por el canal de salida estándar. Sólo si se activa CHECK STATUS comprueba los mensajes de error de las llamadas al kernel.

4.3.4. Conclusiones acerca de la librería CThreads

Pensada para trabajar sobre Mach, la librería de CThreads tiene mecanismos que aprovechan la filosofía subyaciente. Sin embargo, como en la mayoría de paquetes de usuario, no se ha explotado suficientemente toda su potencia.

A nivel de planificación, que es donde hemos centrado principalmente nuestro estudio, no

11. Consiste, básicamente, en quedar bloqueado en el envío de un mensaje al kernel y, al volver, comprobar cómo está la cola de *ready*.
12. La librería actual permite definir el tamaño de la pila en tiempo de compilación.

existe un *scheduling* completo, ni siquiera se ofrece la posibilidad al usuario de redefinir de manera sencilla un orden de ejecución en sus flujos, si no es modificando la librería.

4.4. Algunos desarrollos experimentales con CThreads

En entornos universitarios se están realizando modificaciones y nuevas propuestas de diseño a paquetes multiflujo como Cthreads. Todos ellos van encaminados a potenciar más el nivel de usuario, ofreciéndole funcionalidades que ya se han desarrollado a nivel de kernel, para la gestión de procesadores virtuales. Las más destacadas de entre ellas son las realizadas en Carnegie Mellon por Randall Dean [DEAN93] y en Georgia por Bodhisattwa Mukherjee {MUKH91].

La versión de Randall Dean está orientada a mejorar los tiempos de cambio de contexto y de bloqueo de los flujos de usuario. Para conseguirlo, ha realizado continuaciones explícitas en los flujos de la librería CThreads. Además ha modificado las primitivas de exclusión mutua para que trabajen también a dos niveles (dar unas cuantas vueltas de espera antes de bloquearse). De hecho, utilizando el mismo interfaz de CThreads, se puede decir que la han reconstruido de nuevo.

En [MUKH91] las diferencias esenciales con la librería original de CThreads han sido la arquitectura sobre la que se han apoyado (NUMA) y la introducción de la vuelta con error en las llamadas a primitivas de la librería para que el usuario pueda obtener más información en la depuración de sus programas.

El hecho de que la memoria compartida esté físicamente distribuida, les ha llevado a optimizar la planificación de los flujos, pudiendo lanzarlos a ejecución en procesadores concretos para no perder la localidad.

De esta última existe una versión monoprocesador sobre SunOS, que utiliza procesos Unix como procesadores virtuales.

4.5. Nueva funcionalidad: prioridades a nivel de usuario

A pesar de la múltiple literatura sobre gran variedad de políticas de planificación de procesos en sistemas multiprogramados de tiempo compartido, las más extendidas y utilizadas son las que se basan en una selección de flujos según su prioridad.

Esta prioridad, puede definirla el usuario o el sistema, según un criterio establecido (tiempo de ejecución que lleva, tipo de trabajo que realiza, tipo de usuario que lo ha lanzado,...). Si las prioridades las define el usuario, puede determinar cuáles son los flujos más prioritarios -más urgentes de ejecutar- y cuáles menos -por ejemplo, que sólo se ejecuten si no hay más trabajos disponibles en ese momento-. De este modo, ya se le dota de una cierta capacidad para gestionar la ejecución de sus flujos, por el orden en que se introducen en las colas. La aplicación tiene así, con el mecanismo de prioridades, una herramienta más sencilla y eficiente que el proporcionado por las primitivas de sincronización, y con menos riesgos de mala utilización.

Existen paquetes de threads que ofrecen, entre sus funciones, la posibilidad de trabajar con prioridades. Uno de ellos es la versión de Pthreads sobre OSF/1. Las llamadas son pthread_setprio() y pthread_getprio() que, sin embargo, todavía no están soportadas en las versiones multiprocesador actuales. Lo mismo ocurre con las funciones que permiten trabajar con las diferentes políticas de planificación, establecidas en la definición de Pthreads (*round-robin* por prioridades, FIFO por prioridades, o tiempo compartido sin tener en cuenta las prioridades): pthread_setscheduler() y pthread_getscheduler().

4.5.1. Prioridades en CThreads$^+$

Para permitir el uso de prioridades en la librería original de CThreads, hemos añadido nuevas llamadas, en las que la aplicación expresa la prioridad mayor, la menor y la inicial con la que van a trabajar sus flujos. Son prioridades estáticas, que se adaptan mucho mejor al objetivo de servicio para las que las queremos: que la aplicación controle la jerarquía de ejecución de sus flujos.

Inicialmente, la librería da por defecto un valor a las tres prioridades de la aplicación: si el usuario no explicita unos valores diferentes, sus flujos se crean todos con la misma prioridad, que no se modifica, y se comportan de manera FIFO respecto a su inserción en las colas de preparados. Es el funcionamiento tradicional de CThreads.

La llamada que permite inicializar estos valores es cthread_init_prio(min_prio, max_prio, default_prio). Estos valores pueden modificarse cuantas veces la aplicación disponga, repitiendo la llamada[13]. A partir del momento en que se hace esta llamada, todos los flujos nuevos de la aplicación se crearán con prioridad default_prio[14].

Se puede modificar la prioridad de un flujo determinado -subirla o bajarla- con la llamada cthread_set_prio(thread, prio) y preguntar por su valor con la llamada cthread_get_prio(thread). La única condición impuesta es que la nueva prioridad esté dentro del rango min_prio-max_prio dado a la aplicación.

Respecto a las estructuras de datos, se ha añadido un nuevo campo a las estructuras del cthread y del cproc -prio en ambos casos- para indicar su prioridad actual; un cambio en la prioridad afecta al cthread siempre, y éste se la transmite a su cproc. Esto es así para evitar posibles confusiones y dependencias de que el cthread estuviera ya asignado a un cproc, o no.

Las rutinas de inserción en una cola, internas a la librería, también han sido modificadas, para hacerlo ordenadamente por prioridades. El funcionamiento de las de extracción se ha conservado FIFO, con el primer elemento el más prioritario de la cola.

4.5.2. Prioridades estáticas versus prioridades dinámicas

A nivel usuario, sin ningún mecanismo de temporización regular, es difícil trabajar con prioridades dinámicas. Y las prioridades estáticas se pueden ver como una política muy pobre y poco útil si el sistema no permite preempción. Sin embargo, las políticas de planificación que se definen a nivel de usuario son internas a una misma aplicación.

El objetivo final que se busca dando diferentes prioridades a los flujos de una misma aplicación es inherentemente diferente al de un entorno general multiflujo. En él los flujos que comparten el procesador y se van alternando en la ejecución son independientes entre ellos y trabajan en un entorno abierto: hay entrada y salida continua de flujos.

De cara a una misma aplicación, aunque es cierto que existen dificultades técnicas a la hora de realizar políticas de tiempo compartido preemptivas, es mejor que se ejecute un determinado flujo hasta que no pueda continuar, que ir alternando el uso del procesador, con la consiguiente sobrecarga en cambios de contexto. A la larga, la aplicación acabará sólo cuando lo hayan hecho todos sus flujos en ejecución.

---

13. La modificación de estos valores no afecta a los que tuvieran ya los threads con anterioridad, sino a las modificaciones posteriores a la llamada.
14. Todas las funciones añadidas, o modificadas, en la nueva versión de CThreads, se encuentran especificadas en el Apéndice B de este trabajo.

> Vemos las prioridades a nivel de usuario como un peso asociado al flujo para indicar en qué momento, o en qué orden, respecto al resto de flujos de la aplicación, debe ser planificado.

4.6. Selección directa de flujos en la planificación a nivel de usuario

Los microkernels actuales han añadido nuevos mecanismos de planificación que permiten mejorar los tiempos de planificación. Detectando y reconociendo determinadas situaciones, se puede cortocircuitar el mecanismo utilizado como base, o incluso sobre las políticas empleadas. Es el caso de las pistas facilitadas por el usuario sobre la conveniencia de replantearse un cambio de contexto en un momento concreto (*hints*), o de pasar directamente el propio procesador a otro flujo, al que seguramente no le correspondería siguiendo la política del sistema (*handoff*) [BLAC91].

Con la nueva llamada cthread_hint(), un flujo replantea un cambio de contexto a la aplicación; este cambio será realmente efectivo si existe un flujo preparado para ejecutar con una prioridad mayor o igual a la del actual. La diferencia básica con cthread_yield(), ya existente en la librería, es que nunca se intenta un cambio de contexto a nivel de kernel, sino sólo con flujos de usuario; si no es posible, se le devuelve el control al primer flujo.

Nuestro objetivo principal de diseño, en todo el entorno de ejecución realizado, ha sido siempre facilitar al usuario todas las herramientas necesarias de planificación, pero sin proporcionar nunca más semántica a cada operación de la que el propio usuario pide:

> Si se pide un cambio de flujo a nivel usuario y éste no es posible, se devuelve el resultado al usuario para que sea él quien decida la nueva acción a seguir.

A través de la llamada cthread_handoff(thread), permitimos que un flujo de usuario dé paso directamente a otro, saltándose la política de planificación de la aplicación -en nuestro caso, sin comprobar la prioridad que tiene el flujo al que le pasamos control-.

4.7. Optimizaciones en las primitivas del propio paquete

Es importante que las secciones críticas, puesto que son zonas de ejecución inherentemente secuencial, estén el mínimo tiempo posible en posesión de un proceso. En un sistema dotado de planificación por prioridades, ésto puede conseguirse forzando la bajada de prioridades en las esperas activas, y la subida de prioridades cuando se está en posesión de una exclusión mutua. Así, cuando un flujo pide entrar en una exclusión mutua, si no lo consigue, baja su prioridad al mínimo definido por la aplicación. Cuando el flujo que tiene cogido el *lock* lo libere y el primer flujo consiga pasar, recuperará su prioridad inicial.

Como nuestro sistema no es preemptivo, no basta con bajar la prioridad de un flujo, sino que hay que forzar una replanificación; en nuestro caso, puede hacerse con cthread_hint() y, si existe un flujo preparado de mayor prioridad, se producirá un cambio de contexto. Todavía sería mejor tener un apuntador en la estructura del *lock* que se quiere conseguir, que indicara qué flujo que tiene cogido el *lock* para, en caso que no estuviera ejecutándose, hacer un *handoff* hacia él:

ganaríamos tiempo en liberar la exclusión mutua. Veremos una realización de este mecanismo en el Capítulo 6, mediante una nueva primitiva spin_lock().

De igual manera, al liberar el *lock*, podríamos planificar directamente a alguno de los flujos que esperan por él, para que recuperara su prioridad inicial, o incluso que la aumentara al máximo para no ser desbancado (en caso de permitir desbanque).

A nivel de kernel, existen realizaciones prácticas de estas optimizaciones en algunos sistemas experimentales (*smart scheduler* en Symunix [EDLE88]).

Nosotros hemos realizado una versión similar, haciendo primitivas de sincronización a dos niveles: realizamos una espera activa durante un cierto tiempo y, si en ese intervalo no hemos conseguido el *lock*, bajamos la prioridad y hacemos una llamada a cthread_hint() para forzar una replanificación. Existe también la opción de forzar una replanificación sin bajar la prioridad, debido a que, como no sabemos qué flujos están esperando por un *lock*, no utilizamos *handoff* y un thread en espera activa podría no llegar a ejecutarse nunca por estar a la prioridad más baja.

Estas optimizaciones en la planificación cuando hay esperas activas son elección del usuario en tiempo de compilación de la librería.

4.8. Evaluación del rendimiento de la nueva librería de CThreads+

Para evaluar el rendimiento de la nueva librería CThreads+, hemos realizado una serie de microbenchmarks. Cada uno de ellos simula el comportamiento de un pequeño servidor y utiliza de modo más o menos intensivo la herramienta que queremos medir.

El servidor consta de dos tipos diferenciados de flujos, según el trabajo que realizan: un grupo se dedica a recoger datos de un servidor externo a él y se los pasa a otro grupo que realiza un trabajo intensivo en CPU con ellos.

Para aislar al programa de cualquier otro agente externo del sistema, hemos trabajado dedicándole procesadores, en un pset.

Existe una sincronización para pasar trabajo y esperar trabajo, que permite variar las políticas de planificación de los trabajos con las nuevas primitivas de la librería. El esquema de funcionamiento es el que se presenta en el código esquemático de la Figura 4-4 y en la representación de la Figura 4-5: cuando un flujo productor tiene ya datos para procesar elige a un consumidor sin trabajo y le pasa control.

Utilizamos variables de tipo *mutex* y *condition* para asegurar la exclusión mutua al acceder a datos compartidos, siguiendo el siguiente esquema:

```
type_t                  data;
mutex_t                 data_lock;
condition_t             data_cond;
```

Para conseguir el acceso a un dato:

```
mutex_lock (data_lock);
while (!freeData ())
  condition_wait (data_cond, data_lock);
m = getData ();
mutex_unlock (data_lock);
```

Para liberar un dato:

```
mutex_lock (data_lock);
release_data ();
condition_signal (data_cond);
mutex_unlock (data_lock);
```

La sincronización entre el productor y el consumidor sigue el patrón:

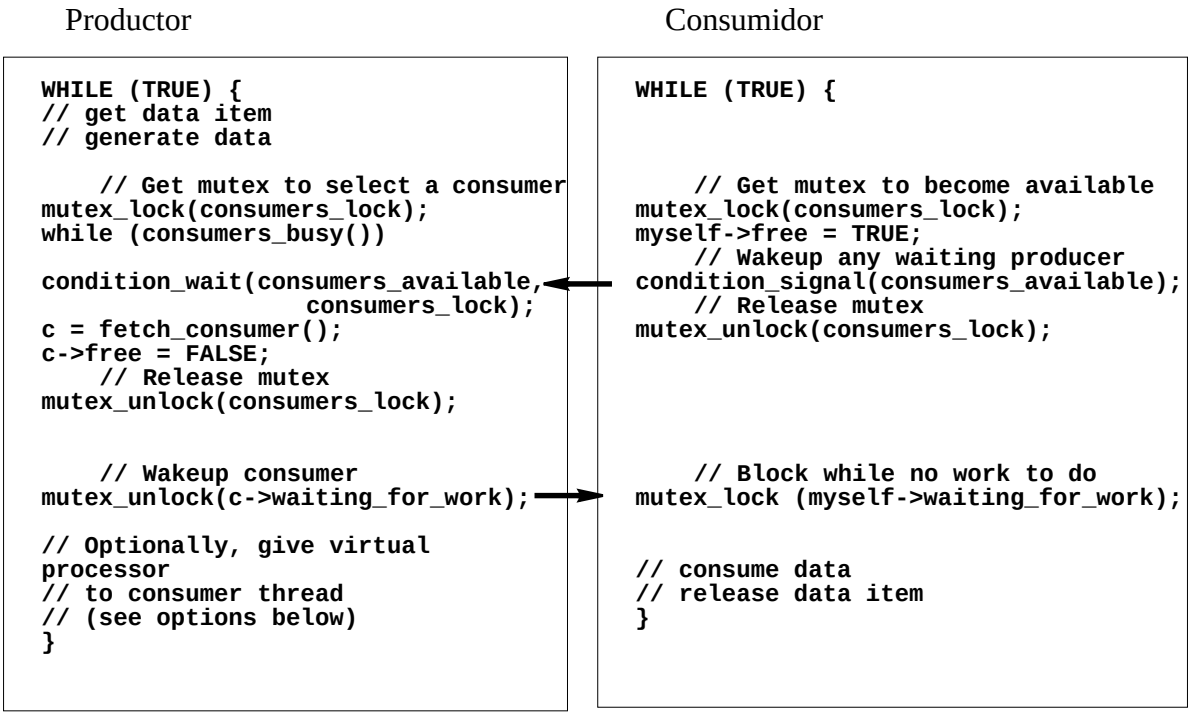

Figura 4-4 Sincronización entre el productor y el consumidor

Una vez que un productor sabe a qué flujo consumidor puede pasarle trabajo, tenemos diferentes posibilidades de planificar, que han dado lugar a diferentes versiones del servidor. Dos versiones corresponden a diferentes políticas de planificación con la versión CThreads^{+} y otras dos, con las políticas que permite la librería en su versión original.

Con la versión de CThreads:

- NOP: Como primera opción, no hacer nada. Generalmente, esta es la opción que tomarán los usuarios y, en este caso la planificación llegará más tarde; bien cuando el sistema seleccione al procesador virtual en que corre el consumidor o bien cuando el flujo productor se bloquee.
- YIELD: La segunda opción es llamar a cthread_yield(), que intentará un cambio de contexto a nivel usuario. Si el intento falla, hará una llamada al kernel para que selecciones otro procesador virtual.

Con la versión de CThreads^{+}:

- HANDOFF: El productor hace cthread_handoff(consumidor) para pasarle el procesador virtual. Si falla, vuelve al flujo productor que, como en el caso inicial, se bloqueará más tarde.
- HANDOFF+HINT: La última versión intenta igualmente un *handoff* y, si éste falla, intenta pasar el procesador virtual a cualquier otro flujo preparado llamando a cthread_hint(). Si este segundo intento falla, vuelve al flujo original.

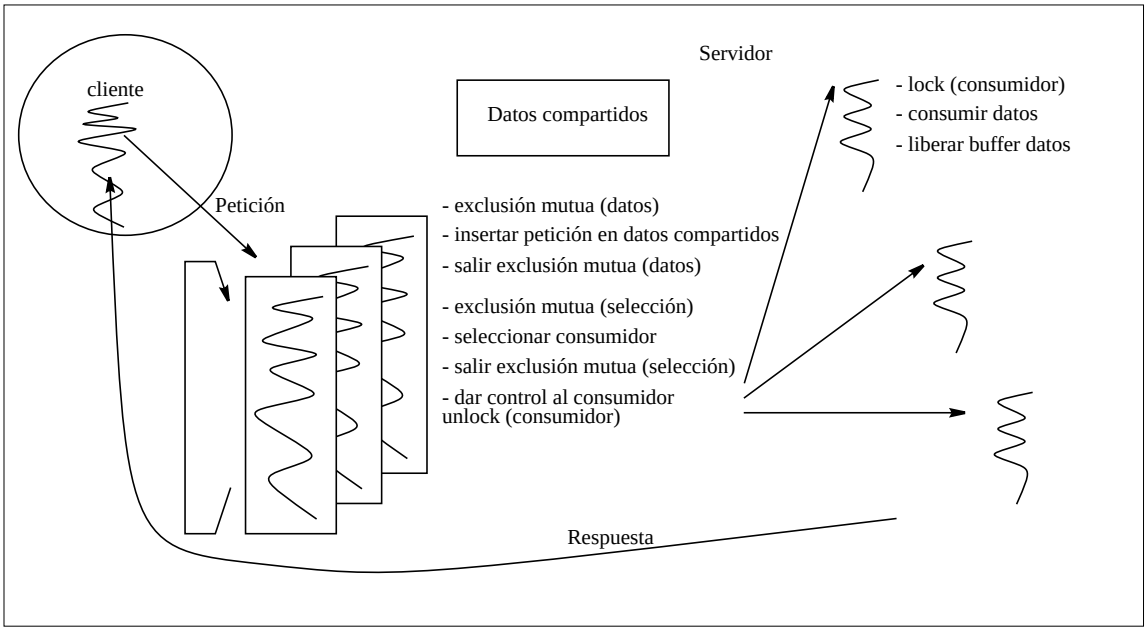

Figura 4-5 Esquema del trabajo realizado por el servidor: un grupo de flujo se dedica a recoger la información y otro a trabajar con ella.

Para ver el comportamiento de la aplicación frente al número de recursos que le daba el sistema, hemos lanzado también diferentes ejecuciones con 1, 2, 3 y 4 procesadores dedicados. Tanto en la versión de la librería tradicional, como en la modificada, vemos que, conforme el número de procesadores físicos del sistema aumenta, el impacto de añadir más procesadores virtuales va disminuyendo, y las curvas que se obtienen son cada vez más suaves, casi planas, para cuatro procesadores (Figura 4-6). De igual modo, con cuatro procesadores, las dos librerías tienden a comportarse igual (Figura 4-6 c)). Esto muestra la poca influencia de la planificación, tanto a nivel usuario, como a nivel kernel, conforme las unidades de procesamiento aumentan [STAL92].

Vamos a estudiar con un poco más de detenimiento los resultados comparativos de las dos librerías. Un primer comentario a hacer es que la forma de la curva, depende de nuestra arquitectura y del diseño y herramientas utilizadas para realizar la aplicación.

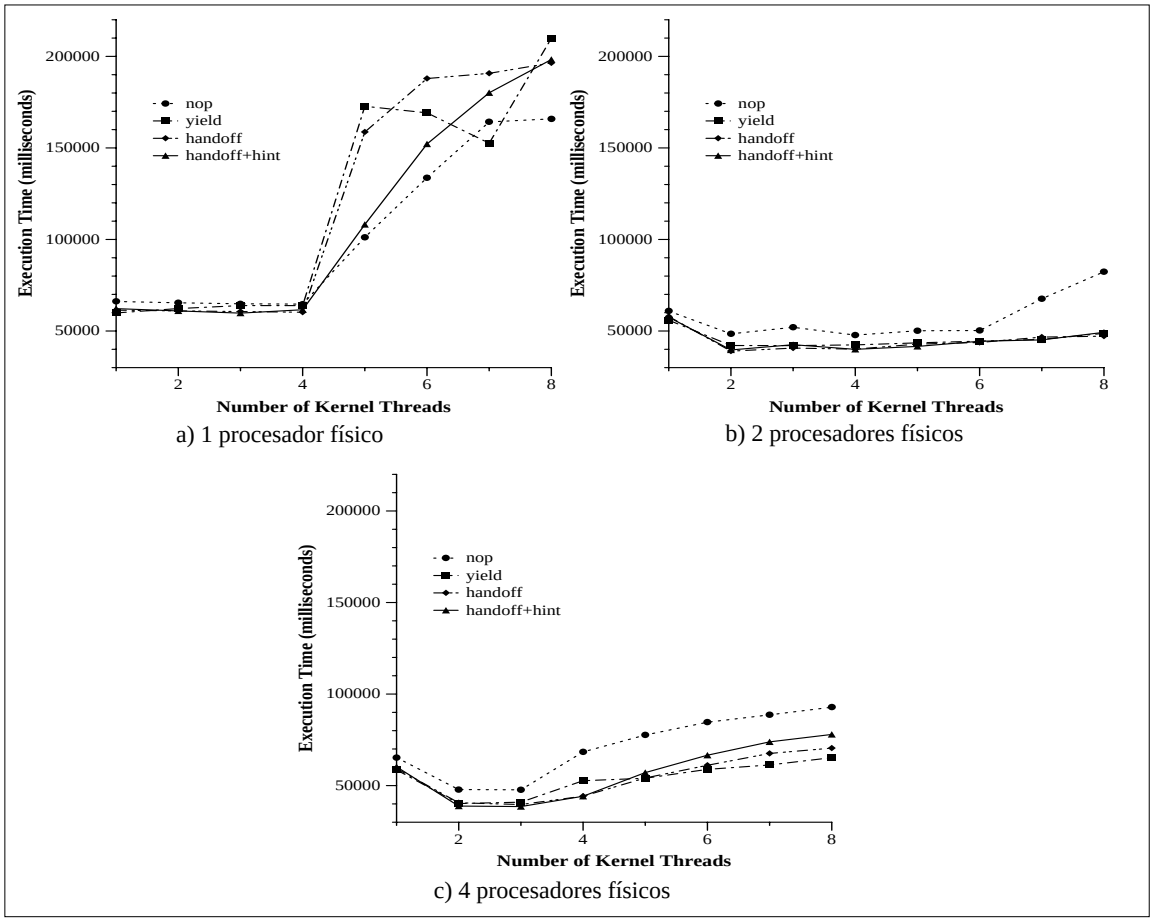

Figura 4-6 Tiempo empleado en realizar el servicio con 1, 2 y 4 procesadores físicos respectivamente, variando el número de procesadores virtuales y las políticas de cesión del procesador virtual.

Observamos que, cuando hay pocos procesadores virtuales, es decir, existe concurrencia a nivel de usuario, el tiempo de ejecución de la librería modificada es mejor que la de la tradicional. En cambio, conforme va creciendo el paralelismo a nivel usuario y la concurrencia a nivel de kernel (crece el número de procesadores virtuales, manteniendo fijo el número de procesadores físicos), el comportamiento de la antigua librería es mejor, hasta llegar al paralelismo total a nivel usuario en que ambos valores son comparables. Esto es debido a el kernel sigue haciendo su propia gestión, que puede ser totalmente contraria a los intereses de la aplicación.

Si existe un número alto de flujos de usuario para planificar (no tienen procesador virtual asociado), el conocimiento que tiene la aplicación de sus propios threads, la hace capaz de adelantarse y hacer un cambio de contexto eficaz (*handoff*). Conforme el número de procesadores virtuales aumenta, la aplicación se encuentra con que, a nivel de usuario, el flujo que le interesa ya está planificado para correr, no le puede pasar el control; aunque, a nivel de kernel, ese procesador virtual puede no estar planificado. La planificación de nivel usuario va teniendo, entonces, cada vez menos posibilidad de decisión, hasta depender totalmente de la planificación que hace el kernel (Figura 4-7). El punto de inversión está situado alrededor del cuatro por la estructura concreta de nuestra aplicación, que tiene cuatro flujos de cada tipo.

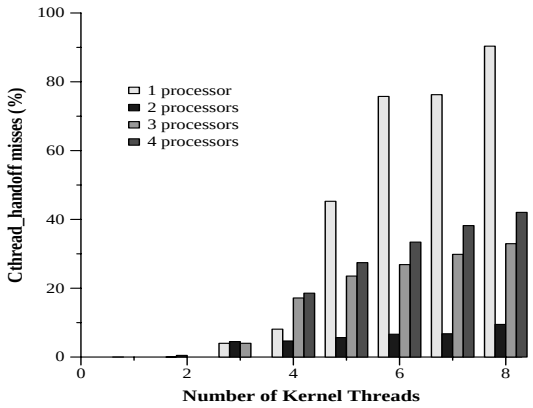

Figura 4-7 Relación del número de thread_handoff(0 fallados variando el número de procesadores físicos y virtuales de la aplicación.

4.9. Conclusiones

Las medidas que hemos realizado muestran cómo cuando el número de flujos de una aplicación es mayor que el de procesadores físicos que le ofrece sistema, las funciones empleadas en el diseño de la aplicación afectan a su rendimiento de manera visible.

También es significativa la variación de tiempos que se obtiene dependiendo de la relación entre el número de procesadores virtuales y físicos que el sistema le otorga: el usuario tiene más potencia de planificación cuanto menos diferencia hay. El caso óptimo se da cuando este mapeo es 1-1, independientemente del número de flujos de la aplicación. De aquí deducimos que:

Es más importante para la aplicación el conocimiento de los recursos físicos de que dispone que no esconderle esta realidad con un número variable de recursos virtuales.

Por otra parte, cuantas más políticas se permitan a nivel usuario, mejor podrá éste adaptar la comunicación y sincronización de sus trabajos para obtener un rendimiento óptimo, o cercano al óptimo.

Podríamos haber contado con algún mecanismo que nos permitiera preempción de flujos a nivel de usuario, apoyándonos en los *signals* de UNIX. Hemos desechado esta idea porque pensamos que es importante trabajar de manera totalmente independientes al subsistema UNIX, para sacar más rendimiento al microkernel por un lado, y para no entorpecer la transportabilidad de las aplicaciones.

4.10. Referencias y Bibliografía

[BLAC91] “Processors, Priority, and Policy: Mach Scheduling for New Environments”
David L. Black
USENIX Winter'91, Dallas TX.

[COOP88] “CThreads”
E.C. Cooper and Richard P. Draves
Technical Report CMU-CS-88-154, School of Computer Science, Carnegie Mellon University, February 1988.

[DEAN93] “Using Continuations to Build a User-Level Threads Library”
Randall W. Dean
School of Computer Science, CMU, March 1993.

[DOEP87] “Threads. A System for the Support of Concurrent Programming”
Thomas W. Doeppner Jr.
Brown University, Technical Report CS-87-11, June 1987.

[DUCH91] “Experience with Threads and RPC in Mach”
Dan Duchamp
Distributed&Multiprocessor Systems (SEDMS II), USENIX Conference, 1991.

[EDLE88] “Process Management for Highly Parallel UNIX Systems”
Jan Edler, Jim Lipkis, and Edith Schonberg,
NYU Ultracomputer Research Laboratory
Workshop on UNIX and Supercomputers, USENIX, September, 1988.

[GIL93] "CThreads por dentro”
Marisa Gil y Nacho Navarro
UPC/DAC Report N. RR-93/06, Enero 1993.

[IEEE92] “Threads Extension for Portable Operating Systems (Draft 6)”
IEEE, P1003.4a/D6, February 1992.

[JONE91] “Bringing the C Libraries With Us into a Multi-Threaded Future”
Michael B. Jones
USENIX, Winter 91, Dallas TX, pp. 81-91, 1991.

[KEPP93] “Tools and Techniques for Building Fast Portable Threads Packages”
David Keppel
Technical Report UWCSE 93-05-06, University of Washington 1993.

[KUCE89] “Making libc Suitable for Use by Parallel Programs”
Julie Kucera
Workshop on Distributed & Multiprocessor Systems, USENIX, October, 1989.

[MACH92] “Mach 3 Server Writer's Guide”
Open Software Foundation and Carnegie Mellon University
Keith Loepere Editor, January 1992.

[MUEL93] “A Library Implementation of POSIX Threads under UNIX”
Frank Mueller
USENIX, Winter'93, San Diego, CA, pp. 29-41.

[MUKH91] “A Portable and Reconfigurable Threads Package”
Bodhisattwa Mukherjee
Proceedings of Sun User Group Technical Conference, pp. 101-112, June 1991

also as Technical Report GIT-ICS-91/02, Georgia Institute of Technology.

[OSF92] “OSF/1 Applications Programmers Guide”
Open Software Foundation
11 Cambridge Center, May 1992.

[POWE91] “SunOS Multi-thread Architecture”
M.L. Powell, S.R. Kleiman, S. Barton, D. Shah, D. Stein and M. Weeks.
USENIX, Winter'91, Dallas, TX, pp. 65-80.

[STAL92] “Operating Systems”
William Stallings
Chap.6 “Scheduling”, MacMillan I. Edition, 1992.

[TEVA87] “Mach Threads and the Unix Kernel: The Battle for Control”
Tevanian, Rashid, Golub, David L. Black, E.C.Cooper and Young
USENIX Association Summer Conference Proceedings, June 1987.

[THAC88] “Firefly: A Multiprocessor Workstation”
C.P. Thacker et al.
IEEE Transactions on Computers, August 1988.

[WEIS89] “The Portable Common Runtime Approach to Interoperability”
Mark Weiser et al.
ACM OSR, Vol 23 Num 5, December 1989.