5

Cooperación entre la aplicación y el kernel para la planificación de flujos

"Convendría preguntarse qué quieren decir las cosas que pasan, porque algo quieren decirnos y son todo lo elocuentes que pueden."

Carlos Pujol ("El lugar del aire")

ABSTRACT: Las aplicaciones paralelas, con múltiples flujos a ejecutar concurrentemente, o en paralelo si el hardware lo permite, son las que pueden decidir de modo eficiente el orden de ejecución -planificación- de sus procesos para obtener el máximo rendimiento.

En sistemas experimentales o académicos, se está trabajando actualmente (ver capítulo 2) en propuestas, 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 ideas provenientes de estos mecanismos y de la librería de CThreads^{+}, hemos diseñado un entorno de ejecución para aplicaciones paralelas que permite a éstas controlar siempre el estado de sus flujos y poder tomar decisiones sobre el momento y lugar de su ejecución.

Este entorno de trabajo es totalmente compatible con el funcionamiento tradicional de los microkernels, sobre los que lo hemos desarrollado. Incluso, en el tiempo, una aplicación puede cambiar de política de planificación.

Explicamos a continuación los elementos del entorno que hemos diseñado y construido y la filosofía de diseño que nos ha ido guiando en su realización.

Capítulo 5: Cooperación entre la aplicación y el kernel en la planificación de flujos

5.1. Introducción

Como hemos perfilado en los primeros capítulos, el diseño de los sistemas operativos tradicionales, en cuanto a la gestión de los recursos, puede ser una carga, la mayoría de las veces inadmisible, para las aplicaciones paralelas actuales o que se prevean para un futuro próximo. Este hecho está provocando un distanciamiento cada vez mayor entre las aplicaciones y el software de sistema, hasta el punto de que se presentan como incompatibles la utilización de un SO y el tener un alto rendimiento en los sistemas multiprocesadores.

Las aplicaciones paralelas, con múltiples flujos a ejecutar concurrentemente, o en paralelo si el hardware lo permite, son las que pueden decidir de modo eficiente el orden de ejecución -planificación- de sus procesos para obtener el máximo rendimiento. Actualmente, ésto se consigue mediante extracción de paralelismo en las herramientas de compilación y traducción o, en tiempo de ejecución, con librerías de threads, como hemos visto. También hemos visto, sin embargo, que esta ayuda se puede volver contra la propia aplicación cuando el kernel ejerce la planificación de sus objetos, los procesadores virtuales de la aplicación. Esto es debido a que ambos niveles trabajan independientemente, sin ningún tipo de comunicación que pueda ayudarles a actuar en la misma dirección, y sí muchas veces en contra.

El motivo de nuestro estudio y consiguiente desarrollo ha partido de preguntarnos si es posible que el usuario pueda tomar decisiones en la gestión de los recursos que el sistema le asigna. Luego, ver si esta toma de decisiones puede afectar de modo positivo al rendimiento del sistema.

Pensamos que la gestión debe ser llevada en consorcio por el kernel y el propio usuario, pues tan malo sería que recayera toda en la aplicación como que la hiciera toda el kernel, como ocurre ahora. Para ello, hay que ofrecer un entorno que permita pasar información al nivel de usuario para que la aplicación pueda manejarla cuando sea necesario y como le convenga para lograr su fin eficientemente. A la vez, el kernel debe poder recuperar el control de la máquina siempre.

El kernel no deja de hacer su función de gestor de recursos; un papel importante, en cuanto que hace de árbitro entre las múltiples aplicaciones que pueden estar corriendo en el sistema. También porque nos interesa, desde el punto de vista del usuario, seguir teniendo esa "máquina virtual" que nos aísla de la complejidad del hardware y nos facilita y simplifica su gestión.

En sistemas experimentales o académicos, se está trabajando actualmente en la propuesta de mecanismos que permitan algún tipo de comunicación, bien de la aplicación hacia el kernel, bien del kernel a la aplicación (ver apartados 2.7 y 2.8 de la Parte I). A partir de ideas provenientes de estos mecanismos y de la librería de CThreads^{+}, hemos diseñado un entorno de ejecución para aplicaciones paralelas que permite a éstas controlar siempre el estado de sus flujos y de los recursos que le asigna el sistema para que puedan ejecutarse. Con ello, puede tomar decisiones sobre el momento y lugar de su ejecución, mejorando su rendimiento.

En este capítulo, vamos a establecer los requisitos necesarios para que el kernel pueda colaborar con la aplicación. Primero teniéndola informada de los recursos físicos que le proporciona y después, de los acontecimientos que le pueda interesar tratar a nivel usuario.

A continuación, expondremos el entorno que hemos desarrollado y la filosofía de diseño que nos ha ido guiando en su realización.

5.2. Colaboración del kernel con la aplicación

En los sistemas multiusuario de propósito general, uno de los objetivos principales de diseño y

realización es habitualmente el facilitar el trabajo y hacer amigable al usuario la programación y utilización de los servicios del sistema. Para ello, se realizan entornos de trabajo en los que el usuario queda al margen de los aspectos más engorrosos. Entre ellos, hay dos puntos básicos que impiden que la aplicación pueda saber con certeza y fiabilidad el estado de ejecución de sus flujos.

Por un lado, en cuanto a los recursos que el sistema le asigna para llevar a término su trabajo. En los sistemas de propósito general, ha sido habitual hasta ahora "esconder" los recursos físicos reales bajo capa de máquinas virtuales que se adaptan totalmente a los requerimientos de la aplicación. En la actualidad, cada vez se hace más necesario saber los elementos de que se dispone realmente: número de procesadores físicos, cantidad de memoria física,... Muchas veces, incluso es de utilidad saber dónde están ubicados sus recursos, en relación a todo el sistema.

Por otro lado, hay acontecimientos que afectan a la ejecución de los flujos de una aplicación mientras están trabajando con código en el interior del kernel. Cuando se pide un servicio al sistema es como si, momentáneamente, el flujo desapareciera de los ojos de la aplicación y volviera un poco más tarde con un nuevo estado, que puede afectar a una variable (como resultado de una función), a uno o varios flujos (una transferencia), o a toda la aplicación.

La aplicación confía en que su flujo está ejecutando código del kernel; pero la verdad es que ese flujo puede estar bloqueado a la espera de un evento y, con ello, bloqueado también su procesador virtual. Si ella fuera consciente de este hecho, podría seguramente adelantar trabajo eligiendo a otros flujos para ejecutar, como lo hace normalmente en las situaciones que ella controla; es el caso, por ejemplo, de las primitivas de sincronización que trabajan a nivel de usuario, ofrecidas por las librerías.

Veamos más despacio ambos casos y cómo el kernel puede pasar la información correspondiente a la aplicación, centrándonos en los recursos que afectan a la ejecución y planificación de flujos.

5.2.1. Relación entre procesadores físicos y procesadores virtuales: mapeo 1-1

El paralelismo le llega a la aplicación a través de los procesadores virtuales que el sistema le proporciona. Con herramientas de nivel de usuario, la aplicación sólo puede planificar sus flujos en procesadores virtuales: un flujo que ya tiene asociado un procesador virtual es, para la aplicación, un flujo en ejecución.

Es habitual que se proporcionen a la aplicación más procesadores virtuales que físicos, dándose concurrencia a nivel sistema. Ésto significa que hay procesadores virtuales que no están realmente ejecutándose, aunque la aplicación así lo vea desde su nivel. Es posible, incluso, que los flujos que más prioridad tienen para el usuario, porque su trabajo sea más urgente, no hayan sido planificados por el kernel, por desconocimiento: la política de asignación de prioridades del kernel se basa, generalmente, en parámetros diferentes, independientes de la aplicación, y ligados más a características del procesador virtual que al código que se está ejecutando en su contexto (tiempo de CPU consumido, por ejemplo). Esto hace que la planificación prevista por el propio programa pierda toda su eficacia.

Sólo cuando el número de procesadores virtuales coincide con el de procesadores físicos, la aplicación puede estar cierta de los flujos que están efectivamente corriendo:

Para que una aplicación conozca cuáles de sus flujos están efectivamente ejecutándose ha de disponer de tantos procesadores virtuales como procesadores físicos. No puede darse concurrencia a nivel de kernel.

A partir de esta situación, conociendo el paralelismo real de que dispone - número de procesadores físicos que le ha asignado el sistema-, lo más adecuado será que se ejecuten realmente los flujos que conviene a la aplicación por las razones que se hayan estimado oportunas: los que hagan más trabajo, los que acaben más rápido, los que vayan más lentos,... Podemos decir que la aplicación CONTROLA totalmente lo que hacen, esperan y necesitan cada uno de sus flujos; porque CONOCE su trabajo y el momento en que necesita de cada uno de ellos, respecto al estado global de toda la aplicación.

Pero no es suficiente con que el número de procesadores virtuales que se dan a la aplicación sea el mismo que el número de procesadores físicos: el sistema podría ir permutando los procesadores virtuales entre los físicos, siguiendo con sus políticas de tiempo compartido. De manera intuitiva parece que ésto no se vaya a dar si, al replantearse un cambio de contexto, no hay ningún objeto planificable. Por los desbloqueos y sincronizaciones que suceden entre los objetos de kernel, dependiendo de cómo el sistema lleva a término los cambios de contexto, puede ser que haya objetos que pasen por la cola de preparados para la replanificación y, entonces, podrían volver a ejecutarse en cualquiera de los procesadores que estén planteándose un cambio de procesador virtual[1]. Se pierde, entonces, la información acumulada en las memorias cache o local del procesador; incluso, en arquitecturas donde la localización de los datos sea crítica -acceso NUMA-, empieza a haber pérdida global del rendimiento de la aplicación, al aumentar los tiempos de acceso a datos [MARK92].

Ésto sin contar con que el único conocimiento que tiene la aplicación del procesador físico en el que está corriendo es a partir del procesador virtual que lo representa. Si un procesador virtual fuera rotando de un procesador físico a otro, la aplicación acabaría perdiendo el control real del procesador físico en que está corriendo cada uno de sus flujos y con ello, su localidad.

Se ha de eliminar todo tipo de replanificación en el interior del kernel, de modo que un procesador virtual no cambie de procesador físico.

Con ello eliminamos además la sobrecarga que introduce el sistema con los algoritmos de replanificación.

Opinamos que la gestión de los recursos que el sistema distribuye entre las aplicaciones debe ser llevada en consorcio por el kernel y el propio usuario, pues tan malo sería que recayera toda en la aplicación como que lo hiciera, como hasta ahora, toda el kernel. Para ello, hay que ofrecer un entorno que permita pasar información al nivel de usuario de modo que la aplicación pueda manejarla cuando sea necesario y como le convenga para lograr su fin eficientemente.

A la vez, hay que seguir manteniendo la gestión tradicional privilegiada del kernel, en la que los acontecimientos, determinados acontecimientos, pasaban ocultos al usuario. El kernel debe poder siempre recuperar el control de la máquina; y hay, por otra parte, operaciones privilegiadas que difícilmente podría dejar en manos del usuario sin suponer un peligro para él o para el resto de aplicaciones del sistema[2].

---

1. Puede ocurrir sobre todo en sistemas como Mach, que carecen de un módulo planificador centralizado, y la planificación es *self-scheduling* por procesador.
2. Se han incluído aquí, tradicionalmente y por protección, la gestión de los dispositivos físicos -periféricos- del sistema.

El sistema tendrá al usuario informado de:
a) el número de procesadores físicos de los que dispone la aplicación,
b) el procesador físico al que representa cada uno de los procesadores virtuales que le ha ofrecido y
c) cuando llegue un evento, del evento que ha llegado y en qué procesador ha llegado.

Así, la aplicación, sabe exactamente a cuál de sus flujos afecta dicho evento y puede tratarlo.

La aplicación puede disponer, entonces, de procesadores virtuales que son una representación exacta, cada uno, de los procesadores físicos que el sistema le ha asignado. Podemos decir que el kernel está ofreciendo a la aplicación, a través de estos procesadores virtuales, “contextos de ejecución” en los cuales ella decide el “contenido”.

PROPONEMOS que el kernel ofrezca a la aplicación contextos de ejecución (**eXecution context, o eXc** para abreviar) como **procesadores virtuales en los que la aplicación pueda planificar y gestionar sus flujos**.

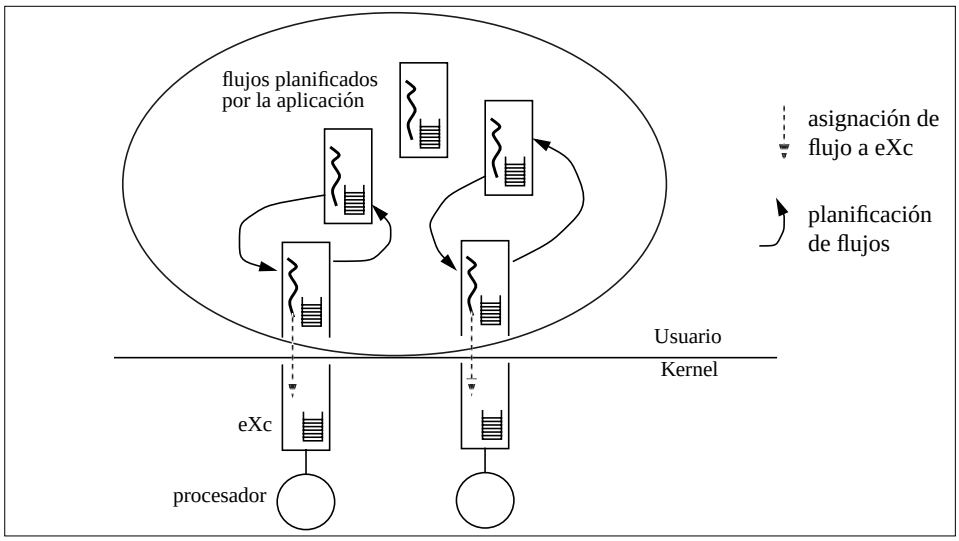

Figura 5-1 A través de los eXc, el sistema ofrece procesadores físicos a la aplicación y un contexto en el cual poder ejecutar sus flujos.

Cada eXc, representa a un procesador físico asignado a la aplicación, y sólo a uno. Además, está ligado a él, de manera que no se dan cambios de contexto dentro del kernel. Es un nuevo paradigma en la planificación de flujos, otro prisma desde el cual ver las decisiones que influyen en la elección de flujos a ejecutar en un momento dado.

A través de cada eXc, el sistema tiene al usuario informado del número de procesadores físicos de los que dispone la aplicación y del procesador físico al que representa cada uno de los

procesadores virtuales que le ha ofrecido. Hasta ahora, ésto era función del kernel; ahora, el kernel ofrece "una vasija a la aplicación en la que poder planificar flujos en procesadores físicos"[3]: es la aplicación la que toma las decisiones (Figura 5-1).

5.2.2. Mecanismos para la notificación de eventos

Cuando en la máquina se produce un evento, éste llega al kernel -si es un evento producido por el hardware- o bien se da, ya desde su inicio, en el kernel -si es un evento de tipo software, por ejemplo un bloqueo por sincronización o a la espera de un mensaje-. En cualquier caso, todo evento del sistema, finaliza en un conocimiento por parte del kernel de dicho suceso.

Queremos diferenciar ahora dos tipos de gestión para estos acontecimientos que recibe el kernel: una, que podríamos llamar transparente al usuario porque la gestionará enteramente el kernel, y otra que puede transmitirse al usuario para que sea este mismo el que decida qué hacer y cómo.

Si el acontecimiento que llega es de los que se ha decidido que gestione la aplicación, el kernel debe notificárselo. Puede hacerlo a través de memoria compartida o enviándole un mensaje; pero ello significa que no es un evento urgente, ya que la aplicación, para tratarlo -para darse cuenta de que ha ocurrido- debería consultarlo cada cierto tiempo, a través de un mecanismo de *polling* o encuesta.

Otra posibilidad para notificar un evento a una aplicación, de manera asíncrona, consiste básicamente en poner en marcha código de la propia aplicación que pueda tratarlo. En este sentido, se están desarrollando mecanismos de comunicación desde el kernel a la aplicación, como son interrupciones software [MARS91] o, más sofisticado y complejo, las *scheduler-activations* [ANDE90]. Podemos entenderlo como el mismo mecanismo utilizado en una llamada al sistema pero desde el kernel al usuario (upcall, de abajo hacia arriba). Se diferencia de una llamada al sistema por el hecho de que no vuelve al punto desde dónde se invocó, el kernel, sino que continuará ejecutando código de usuario.

PROPONEMOS que, a través de un interfaz adecuado, el kernel pueda avisar al usuario, de manera asíncrona, de todos los acontecimientos que puedan interesar a la aplicación para replantearse un cambio en la ejecución de sus flujos.

En la propia aplicación tiene que darse ahora, o poder darse para los usuarios que así lo requieran, los mecanismos y políticas de planificación de flujos que antes llevaba el kernel. La idea básica es que la aplicación indique al kernel la continuación de código donde quiere que siga ejecutando, ya en modo usuario, cuando se produzca una notificación y un tratamiento. Es equivalente al servicio que el sistema tiene asignado para cada una de las interrupciones hardware que le llegan.

A este código que va a ejecutarse en la aplicación, y siguiendo la similitud con una interrupción hardware, hay que proporcionarle un cierto entorno mínimo para que pueda correr. Por ejemplo, es imprescindible que se le asigne una zona de memoria para su propia pila, para los parámetros que reciba y sus variables locales. Esta pila pertenece al espacio lógico de usuario y por tanto, corresponde a memoria gestionada por la propia aplicación. Así, la aplicación también deberá informar al kernel de la reserva de un espacio de memoria para pilas, por si acontecieran varios eventos a tratar simultáneamente (Figura 5-2).

3. Cfr. [ANDE90]

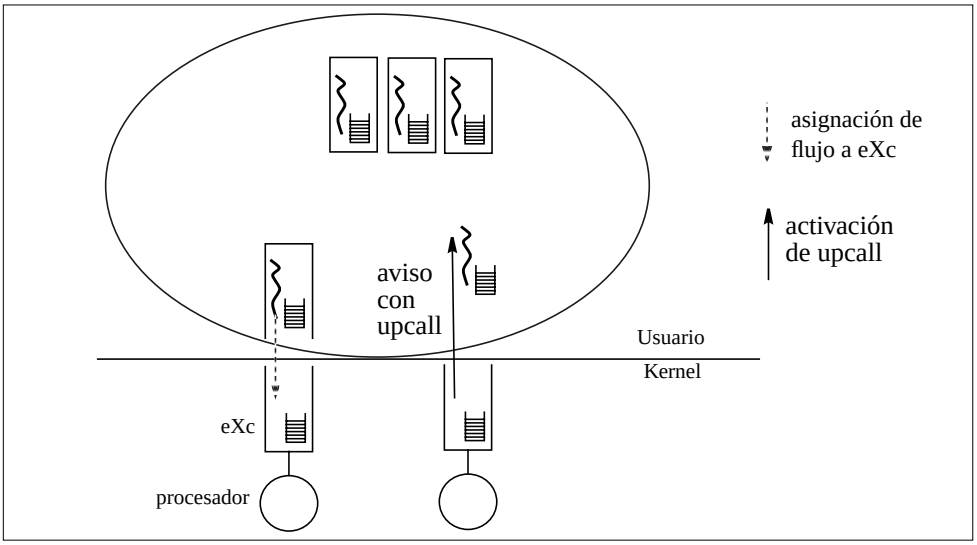

Figura 5-2 Notificación de un evento desde el kernel a la aplicación mediante una upcall.

No todas las aplicaciones necesitan una planificación original y concreta; y por otra parte, sería una carga para el programador el tener que hacerla. Por eso, en el caso general, hay que contar con un interfaz de usuario -una librería en nuestro caso- que actúe como lo hacía antes el kernel con unas políticas sencillas. Ahora serán sensiblemente más rápidas porque se realizan en el propio entorno de la aplicación, en modo usuario.

Programadores más experimentados, sin embargo, han de poder diseñar sus propios mecanismos y políticas que enriquezcan el diseño y que aumenten el rendimiento y la eficacia de la aplicación en cuestión. Un ejemplo muy sencillo y gráfico sería que el kernel pudiera avisar de la terminación de un *quantum* definido por la propia aplicación para sus flujos; que pudieran ser los propios programas los que llevaran a cabo, por ejemplo, una política de *round-robin*, en lugar de hacerlo el sistema.

5.2.3. Eventos que se notifican a la aplicación

Es decisión del diseñador del sistema definir qué eventos y bajo qué circunstancias pueden dejarse en manos de la aplicación. Esos puntos en los que el sistema pasa la gestión a la propia aplicación los hemos denominado momentos de *scheduler-activation*, porque son momentos de posible replanificación en la aplicación. Notar que en [ANDE90] se conoce bajo la denominación de *scheduler-activation*, tanto el momento, como el procesador virtual que notifica el evento y se ofrece a la aplicación para planificar a sus flujos.

Ante un mismo acontecimiento, los eXc son capaces, en determinados momentos de activar una replanificación a nivel usuario y en otros, sin embargo, dejar que la gestión la siga llevando el kernel (tratamiento tradicional de threads de kernel) y su efecto pase oculto al usuario.

En la versión actual de nuestra realización, los acontecimientos que hemos decidido notificar al usuario son aquellos que afectan directamente a la ejecución de los flujos:

- bloqueo de un flujo en el interior del kernel,
- desbloqueo de dicho flujo,
- asignar un nuevo procesador físico a la aplicación y
- quitarle un procesador.

Además, hemos añadido una nueva funcionalidad al kernel de Mach, que provoca un quinto evento:

- un temporizador

para que, por ejemplo, el planificador de la aplicación pueda recuperar el control cada cierto tiempo, como ocurre en los sistemas operativos tradicionales

En cuanto a los momentos de *scheduler-activation*, hemos elegido aquellos que provienen de una llamada al sistema. Es comprensible que sean éstos ya que son los provocados por una petición de servicio de la propia aplicación, y no otros que derivan de los mecanismos internos de gestión del propio sistema. En este segundo grupo hemos incluido los fallos de página o las interrupciones de dispositivos, en general. Diferenciamos, claro está, aquellas interrupciones de dispositivos cuyo servicio acabe en una decisión de cambio en la planificación de flujos. Pero sólo en el momento en que ésto ocurra, se subirá la notificación al usuario, si él lo ha solicitado; y será la notificación del bloqueo, no la notificación de la interrupción.

Decidimos que los fallos de página, por ejemplo, no sean notificados y su servicio sea transparente a la aplicación: ante estos eventos, la aplicación no puede tomar decisiones que mejoren su rendimiento. Necesitaría conocer el estado de su espacio de direcciones lógico y físico, y tener la posibilidad gestionarlo[4].

Una vez se reconoce que el acontecimiento que ha llegado es una s-a, se activa el mecanismo de preparación y subida de una upcall. Dependiendo del evento, la información que proporciona el kernel al usuario varía. En nuestra realización, siempre se sube el tipo de evento acontecido, el procesador que está tratando el evento y el identificador de eXc que está subiendo la información. Además:

- Cuando se añade un procesador a la aplicación, se informa del procesador físico que se le ha asignado, para que la aplicación pueda actualizar su información respecto a los recursos que el sistema le confiere.
- Cuando un flujo se bloquea, se sube la identificación del eXc en que corría el flujo que se ha bloqueado. En el eXc que sube la notificación, la aplicación puede planificar a otro de sus flujos.
- Cuando un flujo se desbloquea, se sube el identificador del eXc que se ha desbloqueado, con todo su contexto modificado por la acción del kernel que provocó el bloqueo. Además, como ha habido que desbancar a un flujo asíncronamente, todo el contexto necesario del flujo desbancado, por si la aplicación no lo selecciona en ese instante para seguir ejecutándose.
- Cuando llega un temporizador, el contexto del flujo que corría en ese procesador, por si acaso la aplicación decidiera un cambio de contexto.
- Cuando se quita un procesador a la aplicación, el procesador que se le ha quitado, el eXc que estaba corriendo en dicho procesador para que la aplicación pueda identificar el flujo y también actualizar su información, y el contexto de dicho flujo. Como para subir la notificación se ha tenido que desbancar a otro procesador de la aplicación, también se sube la información correspondiente: el procesador desbancado, para identificar el eXc y el flujo, y el contexto.

5.3. Transferencia de la gestión del kernel a la aplicación

Como ya hemos anotado en el punto anterior, sólo un subconjunto de eventos de todos los que van

4. Véase más sobre el tema en el capítulo “Conclusiones y líneas abiertas”.

a llegar al kernel se van a notificar a la aplicación para que ella los gestione. La aplicación elige entre todos ellos los que quiere tratar y los que deja gestionar al sistema.

Una vez llegados a este punto, nos encontramos con el siguiente escenario de trabajo. El sistema otorga a la aplicación un conjunto de procesadores dedicados y un número igual de procesadores virtuales, con un mapeo 1-1. La aplicación trabaja con un conjunto de flujos que va asignando, según su propia política de planificación, a los procesadores virtuales -y, por tanto, físicos- que le ha dado el sistema.

Pasemos entonces a explicar el entorno en que hemos desarrollado nuestro trabajo y la realización que hemos llevado a cabo para trabajar con eXc. En este nuevo entorno, la aplicación conoce exactamente en qué procesador físico está corriendo cada uno de los flujos que tiene en ejecución y puede tomar decisiones de planificación basadas en este conocimiento.

5.3.1. Entorno de trabajo

El marco en que hemos trabajado ha facilitado, como veremos en más detalle, un diseño limpio y sencillo para ofrecer un interfaz de paralelismo real a las aplicaciones. Es aplicable a cualquier entorno que esté formado por flujos a nivel de usuario con la suficiente entidad como para ser planificados de manera automática (es decir, que conste de un contexto de ejecución, propio, no corrutinas); y, por otra parte, con un funcionamiento en modo sistema con mecanismo de AST (Asynchronous System Trap) o equivalente. Así se garantiza que ciertos acontecimientos a tratar por el kernel se retrasan al instante en que se inicia la salida a usuario. Es éste un punto seguro en el que el procesador virtual puede ser desbancado si fuera necesario, porque su contexto de ejecución en el kernel es nulo[5].

Hemos trabajado, a nivel usuario, con el paquete de CThreads^{+} en el que se han subido todas las políticas y mecanismos que hasta ahora se encontraban en el kernel y de los que no disponía el paquete. El kernel utilizado ha sido Mach, que trabaja con threads de kernel como procesadores virtuales y mecanismo de AST para tratar eventos asíncronos.

Para definir el entorno de una aplicación hemos utilizado, lo mismo que [BART92], el objeto *processor-set* (pset) ofrecido por Mach. Nos permite tratar a la aplicación como una abstracción por encima de la task, que da una unidad de finalidad al conjunto de tasks -una o más- que la forman. Además posibilita dedicar procesadores a aplicaciones, y aislar las políticas de planificación de una aplicación concreta de las del resto del sistema.

Otro elemento importante para poder dedicar procesadores a las aplicaciones es el gestor de los procesadores, o CPU *server*, para dar soporte a la posibilidad de una planificación de propósito general basada en políticas de espacio compartido. Necesitamos una utilidad para asignar procesadores a aplicaciones y permitir así el particionado del multiprocesador y la coexistencia de varios modelos de programación. Las aplicaciones paralelas que desean sacar un alto rendimiento a la máquina necesitan saber cuántos procesadores están disponibles, cuántos pueden pedir en este instante, qué procesadores son, etc.

Existen tres componentes para la gestión y asignación de procesadores:

- El kernel, que tiene los mecanismos de asignación;

5. Este funcionamiento en el interior del kernel como monitor es heredado de UNIX.

- un servidor privilegiado, con las políticas de asignación y soporte a diferentes arquitecturas;
- la aplicación, que pide procesadores al servidor y los utiliza, y puede controlar su uso

A pesar de poseer de esta facilidad para dedicar procesadores a aplicaciones y poder hacer políticas de planificación de grupos y de espacio compartido, no viene ningún servidor de CPUs en la distribución oficial de Mach (ni en la de Carnegie Mellon, ni en ninguna otra de las que utilizan el microkernel como plataforma base: OSF/1, por ejemplo). Debido a ello, hemos diseñado uno propio. Las políticas que ofrece son sencillas, pero no entraba en nuestro proyecto un estudio especial de este componente, sino utilizarlo como herramienta para poder obtener procesadores del sistema.

Las peticiones al CPU *server* indican el número de procesadores y la operación que se desea: solicitar procesadores, devolverlos al pool general y pedir información del estado de la máquina y de los procesadores asignados. En la petición se permite indicar:

- si se necesitan todos los procesadores,
- si es suficiente con los que asigne, aunque sean menos,
- si el número de procesadores asignados ha de ser potencia de dos y
- si nos queremos esperar a que se nos asigne el número que hemos pedido.

El CPU *server* es un servidor multiflujo y puede, por lo tanto, atender varias peticiones en paralelo; también puede retrasar las respuestas cuando haga falta. De igual modo está atento a la finalización súbita por error de la aplicación o la destrucción de los psets, gracias al mecanismo de excepciones de Mach, recuperando siempre los recursos procesador y manteniendo estable el estado de la máquina. Su interfaz de trabajo se describe con detalle en el Apéndice A.

El funcionamiento de este entorno es totalmente compatible con el funcionamiento tradicional de los microkernels sobre los que lo hemos desarrollado: la mecánica de los threads de kernel no es más que un caso extremo en el cual la aplicación no quiere gestionar ningún acontecimiento. Pueden convivir simultáneamente aplicaciones que trabajen con los threads tradicionales (por ejemplo, los servidores y subsistemas) y aplicaciones especiales que funcionen con un conocimiento total de sus recursos físicos. Incluso, en el tiempo, una aplicación puede cambiar de política de planificación.

5.3.2. Inicialización de los eXc

Una aplicación empieza a trabajar con el modelo de eXc, notificándolo con una llamada al sistema en la que, además, indica el pool de memoria que tiene reservado para las pilas en modo usuario que necesitará cuando se active una upcall (Tabla 5-1). A partir de ese momento, los procesadores virtuales pasan a comportarse como eXc, que ofrecen a la aplicación un contexto en el que ella pueda ejecutar sus flujos. El kernel traspasa las decisiones de planificación de flujos a la propia aplicación.

A través de otra llamada al sistema la aplicación indica al kernel, sucesivamente, cada uno de los eventos que quiere gestionar, pasándole las rutinas correspondientes al tratamiento. Esta llamada es independiente de la anterior y se puede proporcionar en cualquier momento, antes y después de dar de alta el funcionamiento de eXc (Tabla 5-1).

Cada tipo de evento que el kernel reconoce puede tener su propia rutina de tratamiento en la zona de usuario. Esta decisión en el tratamiento de altas y bajas de rutinas está inspirada en el funcionamiento de los *signals* de UNIX. Nos permite una gestión dinámica del número de eventos que la aplicación quiere gestionar en un momento dado y en el tratamiento específico para cada uno de ellos. Hemos preferido que sea el usuario, si quiere, el que dé la misma gestión a todas las upcalls, sin dejarle limitado a esta única opción.

eXc_enable (on/off, stack_pool)
eXc_upcall_check_in (event, upcall_routine)

Tabla 5-1 Nuevas llamadas al sistema añadidas con sus parámetros correspondientes.

El interfaz de las rutinas que tiene que proveer el usuario para tratar las upcalls que hemos realizado es el especificado en la Tabla 5-2. Los parámetros que recibe la aplicación con cada upcall son: el evento que ha ocurrido, la identificación de la nueva eXc que el kernel ha creado para notificar el evento, el procesador físico donde ha ocurrido o el que se utiliza para notificar, otros eXc implicados y el estado de los eXc que han desaparecido al ocurrir este evento.

upcall_processor_added (event, new_eXc, processor)
upcall_blocked_thread (event, new_eXc, processor, old_eXc)
upcall_timer (event,new_eXc, processor, old_eXc, state)
upcall_processor_preempted (event,new_eXc, processor, state, old_eXc)
upcall_resumed_thread (event, new_eXc, processor, old_eXc, state, old_eXc)

Tabla 5-2 Upcalls del sistema al usuario, con sus parámetros correspondientes.

La especificación del interfaz de las nuevas llamadas al kernel y de la rutinas de upcall se detalla más en el Apéndice C.

Cada eXc de los que actualmente dispone una aplicación tiene una identificación única en la vida del sistema e informa, cuando se halla en ejecución, del procesador físico al que representa. Como ya hemos comentado en el punto anterior, para representar esta realidad, no basta con la idea del *processor-set* que tiene asignados tantos procesadores como procesadores virtuales dispone la aplicación. No evitamos de esta manera una replanificación entre ellos dentro del sistema[6] y el usuario pierde parte de la potencia al no tener seguridad en cada instante del procesador físico sobre el que realmente están corriendo cada uno de sus flujos, además de la sobrecarga que esta replanificación interna supone. Nuestra propuesta es mantener al eXc sobre el mismo procesador físico sobre el que está corriendo, eliminando las políticas de planificación del kernel.

Un eXc siempre se está ejecutando sobre el mismo procesador físico.

Esta no planificación de procesadores virtuales por parte del sistema entra en funcionamiento con la llamada eXc_enable(on), independientemente de que el usuario quiera gestionar o no los eventos que el sistema le ofrece.

Con la notificación de un evento, cada upcall recibirá un eXc nuevo, que no viene a sustituir a ningún otro del kernel o de la aplicación; sino que servirá para que la aplicación pueda poner uno más de sus flujos a trabajar en paralelo.

---

6. No olvidemos que coexisten los modelos de threads tradicionales y de eXc.

Gracias a los parámetros que recibe de las upcalls, la aplicación conoce perfectamente qué ocurre en cada procesador, qué eXc (uno o varios) se han visto afectados,... y puede mantener actualizada su información acerca de lo que está ejecutándose sobre cada uno de sus procesadores.

La aplicación gestiona completamente el mapeo entre flujo, eXc y procesador físico.

Pasamos a explicar el mecanismo de las cinco upcalls realizadas según el orden de complejidad que a nuestro parecer presentan, de menor a mayor.

5.3.3. Añadir un procesador a la aplicación

La aplicación solicita los procesadores que necesita al CPU *server*, en una o varias llamadas sucesivas. Este servidor arbitra entre las peticiones que recibe de las diferentes aplicaciones y decide otorgar a la aplicación todos o unos cuantos de los solicitados. A medida que se va recibiendo un procesador físico -se incluye en el pset de la aplicación-, el sistema avisa a la aplicación de que dispone de un procesador más para planificar sus flujos, por medio de un eXc. Así, cada vez que la aplicación pide un procesador al servidor y éste resuelva la petición de manera satisfactoria, el kernel enviará una upcall, y con ella un nuevo eXc, a la aplicación.

Además de un nuevo contexto para poder ejecutar sus flujos, en esta notificación el sistema ha de indicarle qué contexto (eXc) le ha dado y qué procesador físico está representando. De este modo, la aplicación sabe dónde están corriendo los flujos que planifique sobre él y a la inversa: cuando lleguen acontecimientos que afecten a dicho procesador sabrá qué flujo estaba corriendo en él.

La rutina de tratamiento de usuario que se pone en marcha en la notificación -upcall-, se replanteará qué flujo de usuario debe continuar su ejecución, según alguna política definida por la aplicación -o bien la que tenga la librería por defecto- y lo lanzará a ejecutar sobre su mismo, el nuevo, procesador virtual (Figura 5-3).

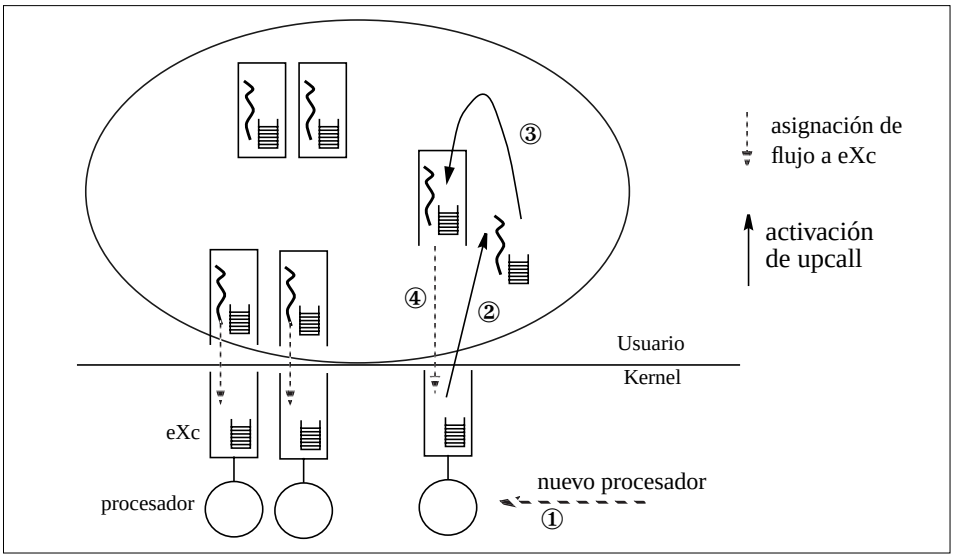

Figura 5-3 Esquema de notificación y replanificación de flujos, por parte de la aplicación, ante la llegada de un nuevo procesador.

Es un procesador nuevo para la aplicación y ésta será la que decide qué hacer con él, pero siempre recordando que **está dentro de un código de upcall** y que mientras no ceda el control a un flujo, no podrá gestionarlo como tal; es decir, no podrá utilizar funciones del interfaz de la librería CThreads$^+$ que necesiten la identidad del flujo.

¿Qué ocurre dentro del kernel? Mach dispone de un thread específico para realizar las operaciones sobre procesadores: el action thread (Capítulo 2).

Este thread del kernel -o threads, puesto que puede haber varios- es el encargado de asignar procesadores a psets (o desasignarlos, ya que el hecho de pasar a pertenecer a un pset hace que se deje de pertenecer, automáticamente, al anterior), de parar -*shutdown*- de manera lógica los procesadores, o de ponerlos en marcha.

En el caso de añadir un procesador a un pset, que es el que nos ocupa, cuando acaba dicha acción, el action thread examina si el nuevo pset -es decir, la aplicación a la que se le ha añadido el procesador- trabaja en un entorno eXc y si, además, quiere que ese evento le sea notificado. Si se cumplen ambas condiciones, se acaba de producir una *scheduler-activation* de adición de procesador.

5.3.4. Aviso por temporizador

Es habitual encontrarse en la literatura sobre planificación concurrente de flujos, múltiples comentarios y loes sobre las ventajas y el barato coste del cambio de contexto a nivel de usuario, en contraposición al cambio de contexto a nivel de kernel[7]. Sin embargo no hay mucho escrito sobre la posibilidad de desbanque de un flujo a nivel usuario, ni tampoco realizado. ¿No es planteable un entorno de tiempo compartido realizado enteramente a nivel usuario?

Subir mecanismos y políticas que hasta ahora siempre hemos visto en el kernel y que, realmente, han llegado a un alto nivel de sofisticación, ya es mucho. Pero si no ofrecemos la posibilidad para poder replantearnos un cambio de ejecución de flujo asíncronamente, poco hemos mejorado.

Deberíamos ser capaces de poder construir en el entorno del usuario, políticas igualitarias de tipo *round-robin*, para lo que es necesario algún mecanismo que permita recuperar el control a la aplicación cada cierto intervalo de tiempo: necesitamos un reloj de usuario.

Para capacitar a la aplicación de reloj hemos construido una upcall de temporización. Puede programarse y desprogramarse, actuar repetidamente cada cierto intervalo de tiempo, o bien avisar de manera aislada a la aplicación.

Se requiere una nueva llamada al sistema para que el usuario active un temporizador, indicando si sólo quiere un aviso o si, por el contrario, quiere que se vaya repitiendo el aviso periódicamente. El resultado de esta llamada es la programación de un *timeout* en el tiempo especificado, a un procesador de la aplicación[8], que acabará en una upcall de temporización en cuanto se pueda disparar en dicho procesador.

A diferencia del caso anterior (añadir un procesador), ahora tenemos que subir un evento y no se dispone de procesador físico en el que hacerlo: en todos hay ya un flujo ejecutándose.

Cuando llegue la *scheduler-activation* de fin de temporización, el eXc que estuviera corriendo en el procesador afectado, subirá la notificación a la aplicación, junto con el estado del flujo que está ejecutando en ese momento.

---

7. Para valores reales medidos, véase apartado correspondiente en el Capítulo 7 “Evaluación del rendimiento en la cooperación kernel-aplicación”.
8. Es una decisión de diseño el que sea un procesador concreto de la aplicación o uno cualquiera.

eXc_timer_set (time, flavor)

Tabla 5-3 Llamada al sistema para programar un temporizador a la aplicación. El parámetro flavor indica si se quiere una llamada esporádica o continuada.

La aplicación puede decidir en ese momento un cambio de contexto entre el flujo que se ejecuta actualmente en dicho procesador y cualquier otro de los preparados para ejecutar (Figura 5-4).

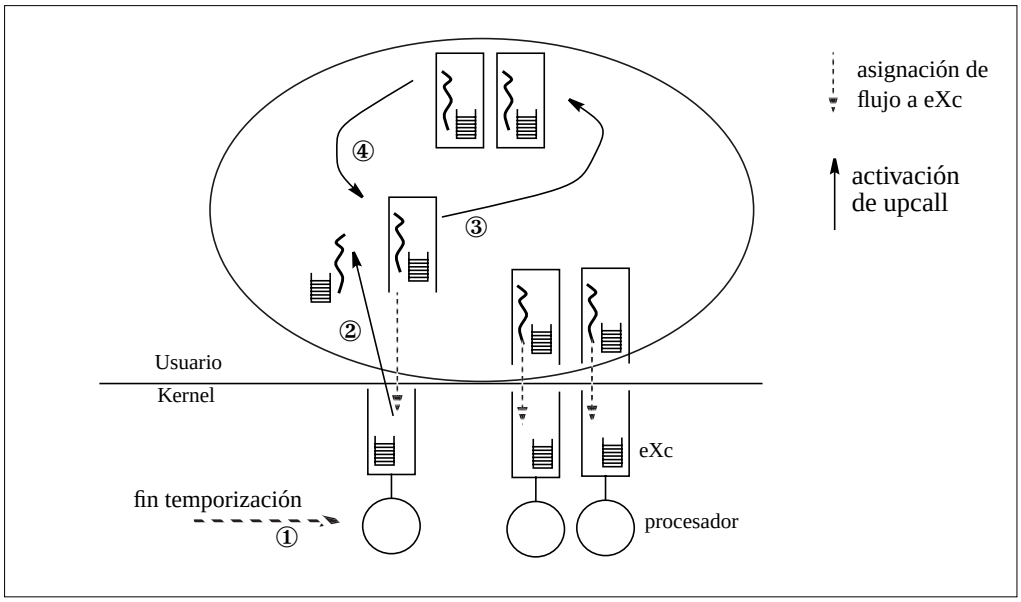

Figura 5-4 Esquema de un aviso a la aplicación por finalización de un temporizador.

Nótese que la planificación que lleva la aplicación es *self-scheduling* respecto al procesador virtual que está efectuándola: no hay ningún control ni aviso en el sentido de forzar un cambio de contexto en otros procesadores virtuales de la aplicación[9].

No hay en este caso adición de un nuevo procesador virtual para el nuevo eXc: puede considerarse como puramente una interrupción software que aprovecha el contexto de kernel del flujo que se estaba ejecutando.

Hemos utilizado el mecanismo de *timeouts*, hasta ahora para uso interno del kernel, de manera que cuando la aplicación hace una llamada al sistema pidiendo una temporización, ésta se anota en el kernel como un *timeout* más. Con ello se permite tener varias temporizaciones pendientes en curso, llamando repetidamente a eXc_timer_set(). Más detalles de su realización, se verán en el siguiente capítulo “Realización de los contextos de ejecución”.

5.3.5. Bloqueo y desbloqueo de un flujo

Cuando un flujo entra en el kernel para pedir un servicio al sistema, es posible que se requiera un cierto tiempo para satisfacerlo; por ejemplo, en el caso de las entradas/salidas, cuando se pide una transferencia de datos desde/a un dispositivo. En estas situaciones, tradicionalmente, se ha suspen-

9. Cfr. “Conclusiones y líneas abiertas”.

dido la ejecución de dicho flujo, mediante algún mecanismo de bloqueo, y otorgado el procesador a cualquier otro flujo preparado, a expensas de la aplicación, a la que no se le notificaba este hecho.

Si se avisara a la aplicación del bloqueo de uno de sus flujos, ésta dispondría inmediatamente de nuevo de su procesador y seguramente sabría planificar al mejor flujo para seguir en ejecución. Para ello, necesita la colaboración del kernel para poder disponer de un nuevo eXc en cuanto uno se le bloquee dentro del kernel. El mecanismo para proporcionar este nuevo contexto será la upcall de notificación de bloqueo.

Por otro lado, actualmente, incluso cuando se trabaja con E/S asíncrona, en que no se para la ejecución del flujo, la aplicación no sabe si está habiendo una espera real dentro del kernel o, si por el contrario, enseguida va a tener disponible el servicio que ha requerido. Se le devuelve el control para que pueda continuar trabajando, pero en el momento en que realmente se haya producido dicha E/S no le va a llegar ningún aviso; y si la aplicación necesita de su finalización, tarde o temprano deberá sincronizarse con el sistema a través de alguna llamada de espera. Es decir, la petición del servicio es asíncrona, pero la finalización es, por parte de la aplicación, síncrona. Sería mucho más potente disponer de un mecanismo que avisara a la aplicación en el momento en el que, efectivamente, se ha servido su petición. En un entorno con eXc ésto se consigue mediante la upcall de desbloqueo.

5.3.5.1. Tratamiento del bloqueo de un flujo

En el momento en que un flujo que se está bloqueando, tenemos un entorno de ejecución en un estado incompleto, ya que todavía no ha finalizado el servicio que ha pedido al kernel. No podemos, pues utilizarlo como eXc para subir la notificación del evento que le atañe: hay que crear una nueva estructura, un nuevo eXc, que suba al usuario el aviso de que se ha producido un bloqueo, y sobre el cual pueda la aplicación planificar un nuevo flujo.

Hay en esta notificación dos eXc implicados: el nuevo, que acogerá ahora un nuevo trabajo, y el que ha sufrido el bloqueo que se está notificando. Ambos identificadores han de subirse a la aplicación para posteriores acciones.

Como muestra la Figura 5-5, la rutina de tratamiento tendrá que elegir un flujo de los que están disponibles para correr sobre el nuevo eXc. A la vez, deberá apartar momentáneamente de toda planificación por parte del usuario al flujo que ha sufrido el bloqueo, hasta que le llegue la notificación de su desbloqueo.

Mediante este mecanismo el usuario sigue viendo y teniendo disponibles en ejecución el mismo número de flujos, sobre el mismo número de procesadores físicos.

Mach, como la mayoría de microkernels actuales, ofrece gran parte de sus servicios a través del mecanismo de paso de mensajes; se puede decir, de hecho, que las únicas llamadas al sistema que ofrece son las de envío (*send*) y recepción (*receive*) de mensajes. Envuelto en un entorno de RPC para hacerlo más cómodo al usuario, todas las llamadas a servicios de Mach o cualquier servidor externo, acaban en el trap de mensajes mach_msg_trap(), excluyendo cuatro o cinco, para pedir información local al *host*, o algunas en fase de experimentación.

Las esperas se realizan encolando al thread de kernel en el puerto involucrado en el paso de mensajes. La finalización del evento se detecta por el envío al mismo puerto de un mensaje de respuesta. Esto ha facilitado mucho la concreción de los puntos donde se puede producir un bloqueo por la espera de un evento, y por tanto, la opción de notificárselo al usuario.

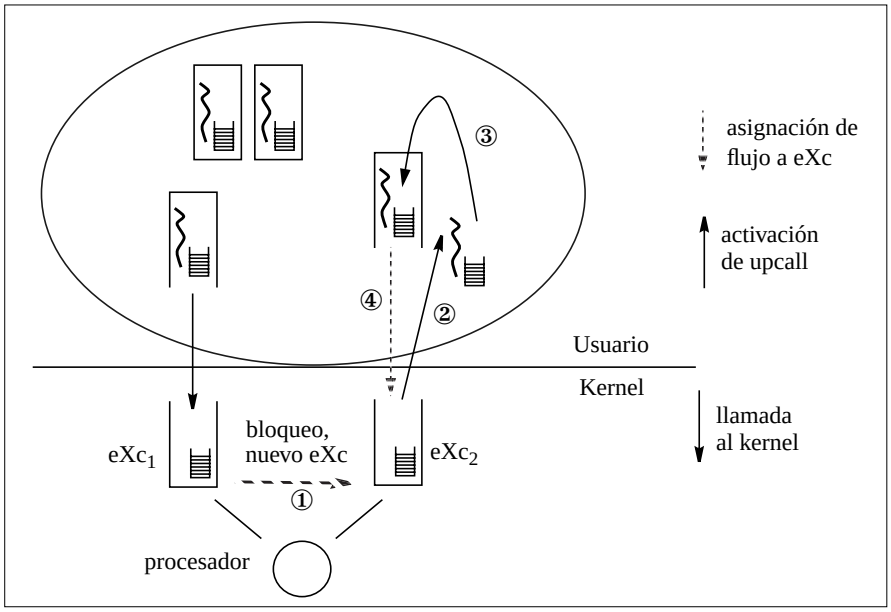

Figura 5-5 Esquema de bloqueo y creación de un nuevo eXc para notificar el evento al usuario.

5.3.5.2. Esquema general de bloqueo de threads en Mach

Cuando un thread ha de bloquearse, parando su ejecución, siempre sigue en Mach un mismo patrón. Primero, se encola él mismo en la cola de bloqueo apropiada, para llamar a continuación a thread_block(). Esta es la función encargada de seleccionar un nuevo thread para ejecutar en dicho procesador y realizar el cambio de contexto entre ambos threads.

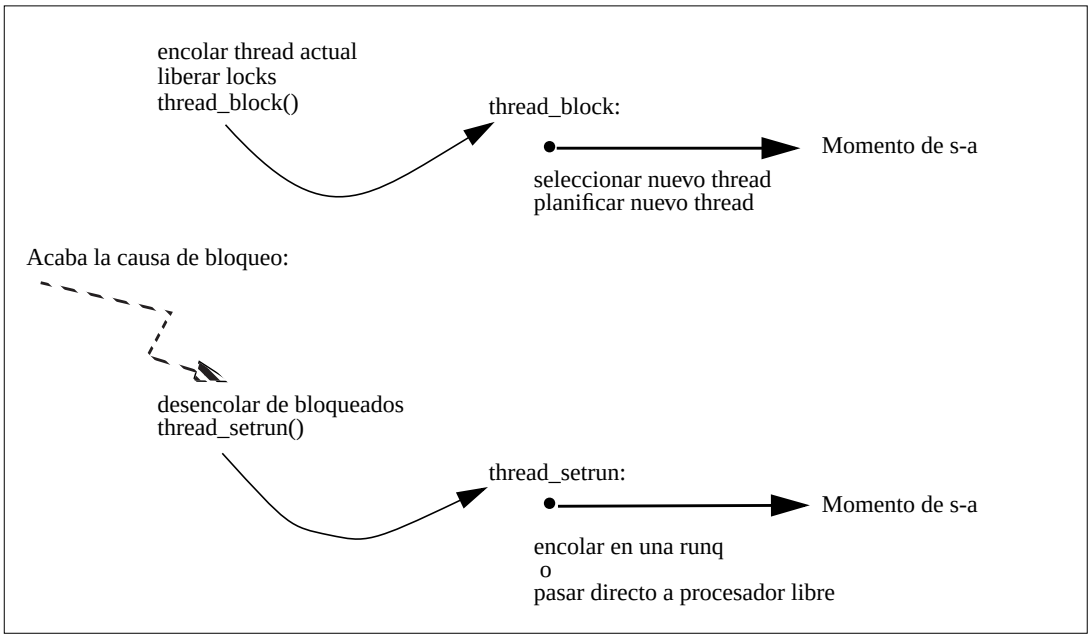

Figura 5-6 Puntos de bloqueo y desbloqueo para Mach 3.0

Después del periodo de bloqueo, el thread vuelve a un estado ejecutable entrando en alguna de las correspondientes colas de preparados, o bien asignándose directamente al campo

next de un procesador. La función thread_setrun() es la que lleva a cabo este paso. Para el caso de monoprocesadores, la función simpler_thread_setrun() es una versión más simple y optimizada.

5.3.5.3. Esquemas de bloqueo con optimización: *handoff* en las comunicaciones

El paso de mensajes es una función importante y frecuentemente utilizada en Mach 3.0. Por ello, se han llevado a cabo algunas optimizaciones para mejorar el *throughput* del sistema [DRAV91].

Una de estas mejoras es la introducción del mecanismo de handoff en el IPC de un thread, como hemos visto con detalle en el capítulo 3. La idea básica es pasar la CPU de un emisor al receptor correspondiente, sin utilizar la política de planificación general. Este especial cambio de contexto, genera un nuevo esquema de bloqueo: un thread puede llamar a la función thread_handoff() y ceder la CPU a otro thread. Este último thread es el responsable de encolar en el lugar apropiado al primero.

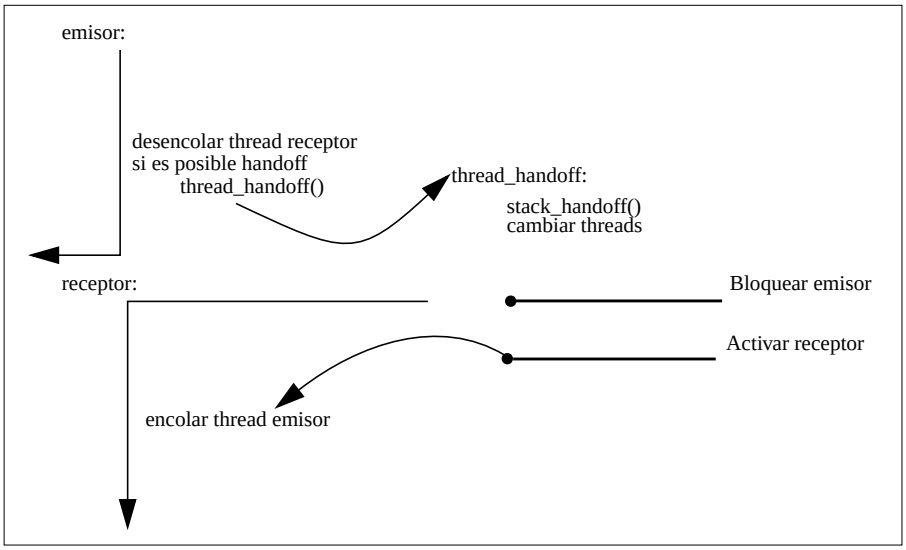

Figura 5-7 Puntos de bloqueo y desbloqueo por mensaje, cuando se produce un handoff

5.3.5.4. Tratamiento del desbloqueo de un flujo

Un flujo bloqueado en el kernel, que ha producido una upcall en el bloqueo, vuelve a ser ejecutable cuando llega el evento por el cual esperaba. En la planificación normal de Mach, sería un nuevo thread de kernel que se incorpora a los preparados para ejecutar y va alternando con todos ellos el uso del procesador.

Ahora no podemos permitir que éso ocurra: romperíamos la condición de tener tantos procesadores virtuales como procesadores físicos y, además, la aplicación dejaría de controlar su paralelismo[10]: hay que notificar lo ocurrido y que la aplicación decida cuál de sus flujos sigue en ejecución.

10. En realidad, como ya se ha eliminado toda planificación en el kernel, este nuevo thread nunca sería seleccionado.

> E**l kernel nunca decide la planificación de un flujo**, tampoco cuando uno se desbloquea: es una decisión que ha de tomar la propia aplicación.

Una vez desbloqueado un flujo, hay que comunicar al usuario este hecho y, como nos encontramos sin ningún procesador disponible para ello, hay que desbancar a alguno de los threads en ejecución. Luego, que decida el usuario cuál de ellos dos, o quizá incluso un tercero, continúa su ejecución.

Para la aplicación, una entrada en el kernel para un servicio, ya no le devuelve el control hasta que dicho servicio haya terminado. Esta es la semántica propia de una llamada al sistema, tanto si conlleva bloqueo como si no[11].

Si nada más acabar un bloqueo, se subiera la notificación al usuario, éste ya asumiría como preparado para ejecutar al flujo correspondiente. Sin embargo, es muy posible que todavía quede pendiente parte del servicio en el sistema. El estado del flujo en el momento del desbloqueo todavía no es un estado seguro: hay que esperar a que termine todo el trabajo en el kernel para que el flujo retenido en usuario pueda continuar su ejecución.

> No se avisará del desbloqueo de un flujo hasta que acabe todo el trabajo en sistema y se halle en el punto de vuelta a usuario. Es éste un punto seguro donde la aplicación puede replantearse el que vuelva a ser ejecutable.

A diferencia de la llegada de un temporizador, en este caso hay que desbancar todo un contexto de ejecución; y habrá que notificar a la aplicación, junto con el acontecimiento de desbloqueo, cuál de sus flujos ha sido desbancado para activar la upcall y subir el contexto del flujo que ha finalizado el servicio.

Hemos utilizado el mecanismo de ASTs de Mach: al detectar el desbloqueo de un flujo que trabaja con eXc's, se programa una AST a un procesador de la aplicación para avisarle que, en cuanto pueda, ceda su procesador físico para subir una notificación. Es además un mecanismo que se adapta perfectamente a nuestra filosofía de diseño, ya que los ASTs son evaluados siempre a la salida del sistema: el desbanque tendrá lugar cuando se vaya a salir a usuario -punto seguro-, después de un servicio del sistema o, como máximo, a la siguiente interrupción de reloj.

Nótese que el flujo que está provocando el desbloqueo no tiene por qué ser de la misma aplicación, y por tanto estaría trabajando en diferente *processor-set* (por ejemplo, un servidor que nos contesta a un mensaje). En el momento del desbloqueo se detecta que el despertado funciona por eXc y hay que notificarlo a su aplicación.

Para seleccionar el procesador al que va a programársele el AST, nos hemos basado en políticas de afinidad al procesador físico. Así se lo facilitamos también a la aplicación, que recibirá la vuelta de un flujo en el mismo procesador en que ese flujo estuvo corriendo por última vez.

11. Exceptuamos claramente el caso de tratar con primitivas de E/S asíncrona.

Es una decisión de diseño que el contexto de los flujos siempre vuelva a ejecución en el procesador físico en el que había corrido anteriormente.

A ese procesador es al que se le programa el AST para que suba la notificación. En caso de que dicho procesador no pertenezca ya a la aplicación, se elige cualquier otro de los restantes.

El thread despertado tiene que finalizar todo su trabajo pendiente en el sistema (recordar que proviene de una llamada al sistema), y completar así la información que luego se subirá a la aplicación, en la pila de usuario que se ha asignado a la upcall. En esta pila hay que poner también el contexto del flujo desbancado que estaba corriendo hasta ese momento en dicho procesador.

Ya en la aplicación, se restaurarán ambos contextos encolándolos en la cola de preparados de modo usuario y se decidirá qué flujo sigue en ejecución sobre ese eXc (Figura 5-8).

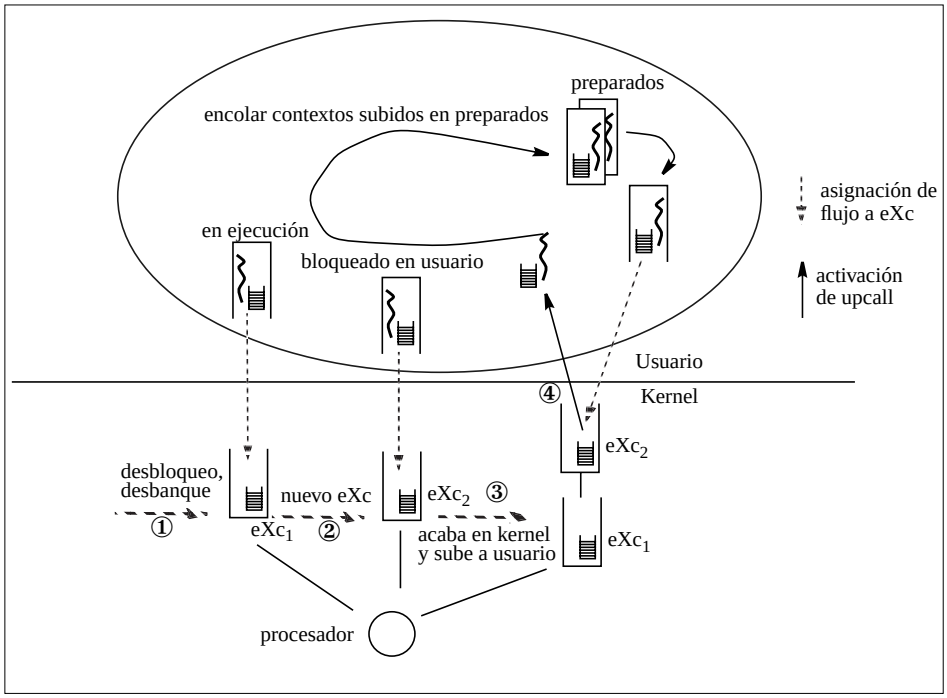

Figura 5-8 Esquema del desbloqueo de un flujo.

Durante el tratamiento de una upcall de desbloqueo, sólo puede haber un desbancado cada vez y, además, siempre hay uno: el que estaba corriendo antes en dicho procesador.

Esta afirmación es una consecuencia de nuestra propuesta de diseño que no se cumple en otras realizaciones [BART92].

Volvemos a la similitud con el tratamiento de una interrupción del hardware, que inhibe la llegada de otras. Nosotros hemos considerado en nuestra realización todas las upcalls al mismo nivel. Ésto nos ha facilitado el diseño, tanto del tratamiento a nivel de aplicación como del tratamiento a nivel de sistema. La gestión de cada upcall ha quedado sencilla y también más rápida.

Como ocurría con los puntos de bloqueo, es un lugar muy concreto aquel en el que los threads vuelven a reincorporarse a las colas de preparados para ser elegidos para ejecución. Es éste, pues, el lugar idóneo para incorporar las *scheduler activations*.

5.3.6. Quitar un procesador a la aplicación

En un sistema multiprocesador el procesador es un recurso más que el sistema asigna a una aplicación y que, por algún motivo, por voluntad incluso de la misma aplicación, le puede quitar en otro momento posterior.

Nuestra política en la asignación de procesadores ha sido no quitar nunca un procesador a una aplicación si ella no lo solicita. Pero en un caso general, con varias aplicaciones corriendo, es fácil que el CPU *server* vaya asignando procesadores a aplicaciones más prioritarias que vayan llegando al sistema y se vea en la necesidad de quitar recursos a una aplicación menos prioritaria [TUCK89].

Cuando la aplicación pierde un procesador, hay que notificarle que, a partir de ahora, dispone de un procesador menos en el que poder planificar sus flujos. También para que decida qué hace con el flujo que hasta ahora corría en dicho procesador: si lo planifica en otro de sus procesadores disponibles, desbancando a un segundo flujo, o si lo encola en preparados hasta que se decida un nuevo cambio de contexto, más adelante.

Como el lector habrá imaginado, el lugar y momento de esta *scheduler-activation* son los mismos que los de asignar un procesador nuevo a una aplicación. Pero esta vez, preguntando por el comportamiento de la aplicación a la que dejamos sin recurso.

Como cada subida de un acontecimiento al usuario, ésta tiene su propia originalidad. Nos han desbancado un flujo por la pérdida de un procesador físico pero, a la vez, nosotros hemos de desbancar a un segundo flujo, de los que está en ejecución, para subir la notificación. Al igual que en el caso de desbloqueo de un flujo, utilizamos el mecanismo de AST's de Mach para llevar a cabo esta operación.

Como en el desbloqueo de un flujo, al usuario le van a llegar dos contextos: el del flujo que estaba ejecutándose en el procesador que ha perdido la aplicación y el de aquel flujo cuyo procesador se ha utilizado para avisar del acontecimiento.

El tratamiento a seguir es simétrico al anterior, puesto que los dos flujos que llegan son seleccionables de seguir corriendo, además de los que tuviera la aplicación preparados para ejecutar. Esta vez, sin embargo, la aplicación tendrá que dar de baja un procesador: disminuye el grado de paralelismo. La upcall le informa del procesador físico en concreto que se le ha sustraído.

Podría darse el caso de que estuviéramos perdiendo el último procesador de la aplicación. Cuando vuelva a asignarse un procesador al pset, subirá una upcall de processor-preempted, pero no subirá ningún otro flujo desbancado: el desbancado era él mismo. Así la aplicación, detecta que le están devolviendo “su” procesador desbancado con anterioridad. Es posible, sin embargo, que sea un procesador físico diferente: tendrá que actualizar la información que se le suba como en el caso de añadir procesador, pero salvando además el contexto del flujo que le viene[12].

12. Obviamente, es una decisión de diseño. Podría haberse hecho este evento transparente a la aplicación. Hemos pensado que era interesante mantenerla informada, de cara a depuraciones y medidas de rendimiento, del número de procesadores que ha ido teniendo durante toda su ejecución.

5.4. Un ejemplo de gestión de flujos en un entorno de eXc: bloqueo y desbloqueo por una petición de servicio

En la Figura 5-9 mostramos la evolución en el tiempo de los eventos de planificación y los momentos de reconocimiento y disparo del mecanismo de *scheduler-activations*/eXc para una llamada al kernel que bloquea al flujo A, una escritura (write()).

5.4.1. Momento de bloqueo: llamada write() al kernel

Para el tratamiento de un evento que puede gestionar la aplicación, diferenciamos dos momentos dentro del kernel:

- el momento de detección de la s-a, y

- el momento de activación de la upcall correspondiente.

El momento de detección de la s-a (①,④), es cuando se ha de examinar si la aplicación quiere o no gestionar dicho evento. Si no quiere hacerlo, el kernel seguirá el camino normal de un bloqueo, en nuestro ejemplo, de manera “transparente” al usuario. Volveremos más adelante sobre este caso. Si la aplicación ha dado una rutina para tratarlo, en ese momento, se prepara y recoge toda la información que ha de recibir el usuario, a la vez que la petición de servicio dentro del kernel -el write- sigue en curso.

Como el eXc de que disponía la aplicación se ha de quedar recogiendo el resultado, se ha de proveer a la aplicación con una nueva “vasija” en la que pueda planificar un nuevo flujo, mientras el que ha realizado el write() permanece bloqueado en el sistema. Es con este nuevo eXc (eXc 2 en la figura) con el que se subirá la información a la upcall.

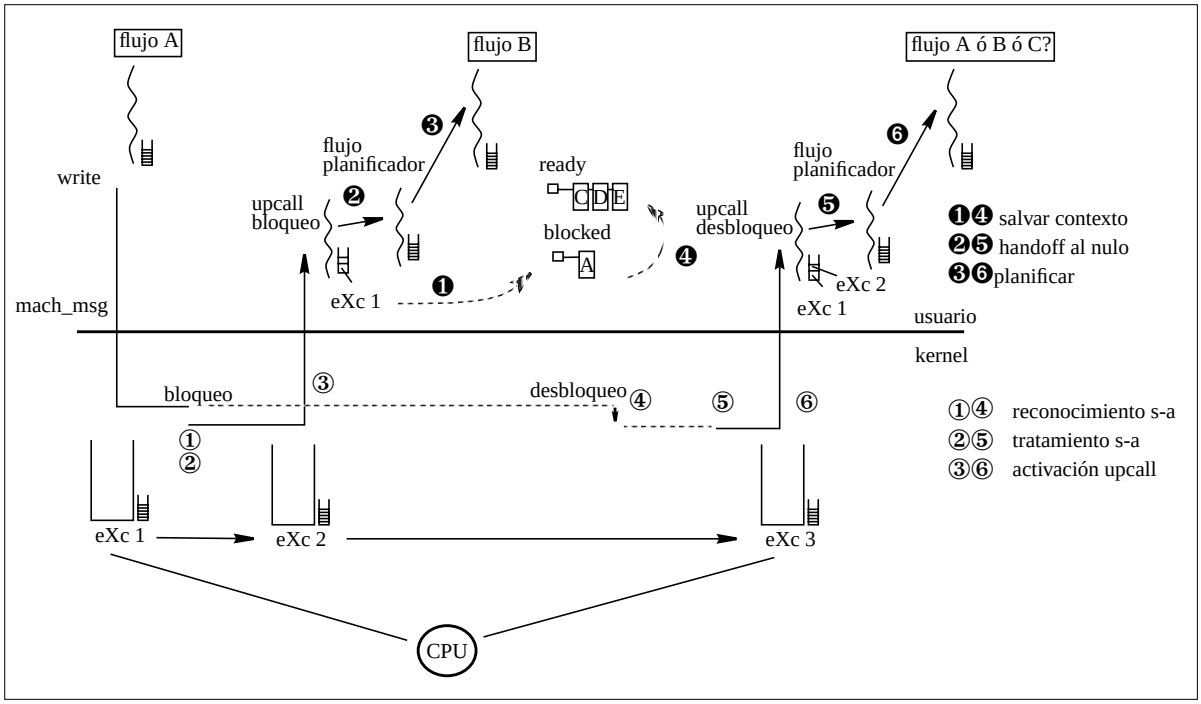

Figura 5-9 Gestión de los eXc y las upcalls de bloqueo y desbloqueo en un write().

Con toda la información del evento recogida en el nuevo eXc, llega el momento de activación de la upcall, a partir de la información que ha proporcionado el usuario al kernel: rutina de

tratamiento de la upcall a la que hay que retornar y dirección de la pila que se le asigna para pasarle los parámetros.

Una vez en la propia aplicación, también podemos diferenciar dos momentos en el tratamiento de una notificación que llega desde el kernel:

- la ejecución de la propia upcall (❶ y ❷), y
- la ejecución del flujo planificador correspondiente (❸).

Cuando se pone en marcha la upcall (❶), el usuario pasa a ser ininterrumpible por el kernel, de modo que, si en ese momento se quisiera avisar de otro acontecimiento en ese mismo procesador, el kernel tendría que posponerlo hasta que la upcall de bloqueo hubiera finalizado. El código de la upcall básicamente salva en el lugar correspondiente los contextos que le sube el sistema (❶) y pasa el control al flujo planificador (❷). En el caso del bloqueo, encolará al flujo de usuario (flujo A, en la figura) en bloqueados.

Una vez en marcha el flujo planificador, elegirá según la política que se le haya indicado (FIFO por prioridades, en nuestro caso) al siguiente flujo planificable de la cola de preparados (❸).

5.4.2. Momento de desbloqueo: finalización del write y vuelta a usuario

En el momento que finaliza el write() en el kernel, se detecta otro momento de s-a. En este caso, sin embargo, la detección se hace en dos pasos separados en el tiempo: la detección de que es una s-a (④) y el tratamiento de la s-a (⑤), cuando se prepara y recoge toda la información.

El hecho de esta separación está provocado por no disponer de un procesador propio para subir el eXc correspondiente. Por ello, hay que dejar un aviso pendiente (④) a alguno de los procesadores de la aplicación (nosotros lo hemos realizado con el mecanismo de AST) para que, cuando se halle en un punto seguro, desbanque el eXc que estaba corriendo y deje subir al nuevo. En ese momento, es cuando se hará el tratamiento de la s-a (⑤). En la elección del procesador, hemos utilizado el *footprint* de Mach para que el flujo vuelva a ejecutarse en el procesador que corrió por última vez, aprovechando el contenido de la memoria cache.

La activación de esta upcall (⑥) es algo más larga, al tener que subir más información: el contexto completo del flujo que ha finalizado el servicio, con el resultado de éste, y el contexto completo del flujo que estaba corriendo en ese procesador y ha sido desbancado para notificar el evento.

La rutina de servicio de la upcall tendrá que extraer de bloqueados al flujo que ha finalizado el write() e insertarlo en la cola de preparados (❹), lo mismo que al flujo que se ha desbancado. Después pasará control al flujo planificador (❺) para que seleccione al que corresponda continuar entre todos los de preparados (❻).

5.5. Gestión llevada íntegramente por el kernel: eventos “transparentes”

Además de los eventos que, por diseño, son responsabilidad del kernel, la aplicación no tiene por qué gestionar todos los eventos que el kernel le ofrece para hacerlo. Si por ejemplo, la aplicación dejara en manos del kernel la gestión de los bloqueos, en el momento de detección de la s-a correspondiente, se seguiría el camino tradicional de un servicio con bloqueo dentro del sistema.

Sin embargo, la aplicación está trabajando en un entorno de eXc y eso significa, por una parte, que hemos eliminado para ella la planificación de procesadores virtuales dentro del kernel; por otra parte, que hay tantos eXc como procesadores físicos, y no más.

Un bloqueo a la espera de un servicio del sistema supone un procesador virtual que no puede ser planificado. El procesador físico permanecería en este caso inactivo en el kernel espe-

rando a la finalización de bloqueo de su flujo.

Para la aplicación, es como si dicho evento no hubiera ocurrido, puesto que no se modifica la información sobre el estado de ejecución de sus flujos. De este modo, se sigue cumpliendo la tesis de que toda planificación de un flujo en un procesador físico la decide siempre la aplicación.

Este tipo de comportamiento -tener procesadores dedicados totalmente a un procesador virtual específico- puede ser adecuado para determinado tipo de servicios. Volveremos sobre él en los siguientes capítulos.

5.6. Nuevas funcionalidades en la gestión de flujos a nivel usuario

Ahora que ya hemos visto qué acontecimientos llegan al usuario y cómo el kernel le hace llegar estos acontecimientos para que sea la aplicación la que ejerza sus propias políticas, veamos cómo trabaja la aplicación ante la llegada de cada uno de estos eventos.

Para cada evento que la aplicación quiera gestionar, se ha de proporcionar al kernel la rutina correspondiente. Esta rutina puede ser la misma para todos los eventos, o propia para cada uno de ellos. Por defecto, además, la librería ofrece una rutina que, sencillamente, hace una replanificación de flujos, guardando el contexto que se le sube, en el caso de que se le suba alguno, y siguiendo la política que esté utilizando la aplicación a nivel usuario -FCFS o por prioridades[13]-. Son básicamente estos tratamientos por defecto los que vamos a comentar aquí, observando algunas características a tener en cuenta para cualquier otro método que se quiera utilizar.

5.6.1. Información compartida por la aplicación y el kernel: objetos de primera clase

Siguiendo con la filosofía de diseño de gestionar una upcall como una interrupción del hardware, la aplicación asigna por cada procesador que le da el sistema una pila para que pueda ejecutarse el código de las upcalls.

En el momento en que se salta a ejecutar el código de una upcall, la aplicación está trabajando con un código “extraño”, puesto que no tiene propiamente una identidad de flujo que lo represente, ni una estructura de datos donde conservar su estado.

No sólo no puede trabajar con primitivas de la librería que supongan una identificación como flujo o un autobloqueo, sino que hay que evitar que sea desbancada su secuencia de ejecución, ya que no volvería a ser planificada para finalizar. Su gestión ha de finalizar lo antes posible y ser ejecutado de manera atómica.

Como el código de usuario no es preemptivo desde la propia aplicación, el único peligro a evitar es la llegada de una upcall durante la ejecución de otra anterior en ese mismo procesador. Para ello, el usuario marcará un flag que haga retrasar la subida, por parte del kernel, de nuevos avisos para ese procesador.

El código de una upcall es ininterrumpible: se ha de inhibir la llegada imbricada de nuevas upcalls sobre un mismo procesador hasta que haya finalizado la que hay en curso.

El aplazamiento de un aviso del kernel a la aplicación lo hemos realizado utilizando de nuevo el mecanismo de AST. Mach permite almacenar en la estructura de thread (para nuestros

13. Son las dos posibles opciones que existen en nuestra realización, aunque podrían facilitarse otras.

objetos eXc, lo hemos conservado) ASTs pendientes de tratamiento. Ésto es debido a que se pueden activar avisos asíncronos a un procesador virtual mientras está ejecutando código de usuario. Cuando dicho procesador virtual entre a realizar un servicio en el sistema -lo más tardar, a la siguiente interrupción del reloj-, en su salida a usuario, se examinarán estos ASTs y se tratarán.

Si el kernel comprueba en un AST de *scheduler-activation* que no puede ser tratado en ese momento, lo "memoriza" para la próxima vez en el campo correspondiente del eXc y sale a usuario, donde continua ejecutando el código desbancado de la upcall en curso.

5.6.2. Planificación de flujos a nivel usuario: continuaciones explícitas

Cuando se activa una upcall siempre hay una replanificación, aunque sólo sea por el hecho de que se salta a un código que ha de liberar rápidamente el procesador. Esta replanificación, puede ser muy sencilla, como ocurre cuando a la aplicación se le da un nuevo procesador: consiste simplemente en escoger al siguiente flujo preparado para ejecutar[14].

En el caso de un temporizador o de sustraer un procesador a la aplicación, la upcall recibe el contexto de uno o dos flujos, respectivamente. Antes de decidir qué flujo continua en ejecución, tiene que salvar estos contextos que le llegan y que también son candidatos a continuar, alguno de ellos, corriendo.

Cuando llega una notificación de bloqueo, la aplicación tiene que marcar a este flujo como no seleccionable para ejecutar, hasta que realmente acabe el servicio que le retiene. Para ello, existe una cola de flujos bloqueados por servicios en el sistema, donde se insertan. Esta cola está enteramente gestionada por código de upcalls, ya que son bloqueos y desbloqueos provenientes del sistema.

Si el evento que llega es la terminación de un bloqueo, además del flujo que se ha desbancado para la notificación, también el flujo desbloqueado es candidato a ser seleccionado para ejecutar. Para ello, primero se ha de extraer de la cola de bloqueados; al insertarlo en la cola de preparados, se ha de hacer con el contexto actual, que es el que llega del kernel, con el servicio pendiente ya finalizado.

Al extraer un flujo de la cola de preparados, ahora tenemos dos tipos diferentes de flujos: los que hayan cedido el procesador voluntariamente, a través de primitivas de la librería, y los que hayan pasado por un bloqueo del kernel.

En el primer caso, se habrá guardado un contexto de usuario mínimo, ya que la operación se realizó síncronamente, en un punto seguro -bien por primitivas de bloqueo en usuario, como condition_wait() o mutex_lock(), bien por cesión voluntaria a otro flujo como cthread_yield(), cthread_hint() o cthread_handoff()-.

En el segundo caso, se ha guardado todo un contexto nuevo, con registros modificados al acabar el servicio.

Para no complicar la gestión de los flujos preparados para ejecución y mantener la que ya había en la librería[15], hemos construido una continuación a nivel usuario para lo que hemos llamado "cambios de contexto largos".

Así, cuando un flujo bloqueado en un servicio del kernel es desbloqueado, se le prepara como continuación una rutina que es la que se encargará de restaurar todo el estado modificado.

Cuando el flujo se selecciona de la cola de preparados, utiliza la misma rutina de restaurar que los flujos de la librería y lo que pasa a ejecutar es la rutina de "restaurar contexto largo", que

14. Recordamos de nuevo que estamos explicando la gestión que proporciona la librería por defecto y que, evidentemente, la aplicación puede complicar notablemente cada tratamiento según le convenga a su trabajo.
15. Los diferentes flujos han de poder ser planificados igualmente por todas las rutinas de la librería que ya lo hacían anteriormente.

saltará después al verdadero punto de código donde se suspendió la ejecución del flujo.

Ésto se hace igualmente para todos los contextos que han sido desbancados de manera asíncrona en el servicio de *scheduler-activations*. Podemos ver esquemáticamente el mecanismo de extracción en la Figura 5-10.

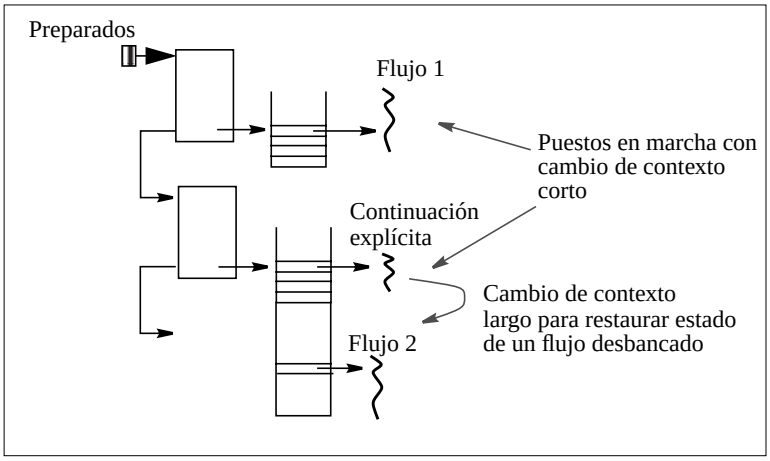

Figura 5-10 Representación de la cola de preparados en usuario, con flujos que han de restaurar diferentes tipos de contexto. Se ha conseguido la automatización en la extracción a partir de continuaciones explícitas a nivel de usuario.

5.6.3. Tipos de flujos ejecutables en la aplicación

La aplicación está formada por un número determinado de flujos durante toda su ejecución. Este número puede ser constante o variable, según se creen todos al inicio, o bien se vayan creando y destruyendo cuando lo vaya necesitando.

Cuando un flujo se bloquea, ya sea a nivel usuario, por una espera o sincronización con otro flujo de la misma aplicación, ya sea a nivel de sistema, por la espera de finalización de algún servicio, la aplicación dispone siempre de un contexto de ejecución -eXc- para planificar un nuevo flujo en ese procesador.

Los flujos que conoce la librería, como es usual en todos los mecanismos a nivel de usuario, son sólo identificables a través de su pila. Ésta es la única estructura de datos reconocida como propia en el momento de ejecución y, por tanto, allí es donde reside su identificador.

Al haber hecho interrumpibles y desbancables los flujos que se están ejecutando a nivel de usuario en cualquier momento, aparecen ahora cuatro tipos diferentes de código en ejecución y, por lo tanto, de pilas con las cuales se puede estar trabajando desde rutinas de la librería.

En primer lugar, puede ser un flujo creado por la librería, reconocido como tal y utilizando la pila que la misma librería le ha asignado.

Podría ser también un flujo definido por la librería pero que estuviera realizando una petición al subsistema UNIX (OSF/1), puesto que existe la posibilidad de trabajar en ese entorno, que es el que ofrece Mach por defecto. En este caso, se le proporciona una pila propia, de modo transparente a Cthreads [JULI91].

Otra posibilidad es que la aplicación se hubiera quedado sin flujos para trabajar y estuviera ejecutándose un *waiter*, o flujo degenerado, ala espera de poder planificar a alguien. La pila que proporciona es también degenerada y, por supuesto, no tiene identificación como flujo (para la aplicación, de hecho, no lo es).

El cuarto tipo de flujo que corre en la aplicación es el de las upcalls, cuya pila tampoco es

la esperada por las rutinas de la librería.

Hemos tomado por tanto, como norma de precaución, no utilizar en las rutinas que intervienen en la replanificación de flujos, ninguna función que necesite saber esta información que, dependiendo del momento podría acabar de modo funesto.

Vamos a ver, sin embargo, con más detalle, las situaciones en que la aplicación está en una estado de ejecución inestable, ya que no existe una identificación propia de dichos flujos, y algunas funciones dejan de trabajar con corrección, incluso provocando la finalización anormal de la aplicación.

El caso de desbancar un flujo que esté trabajando con el emulator de UNIX no lo contemplamos ya que no afecta de ningún modo a nuestro entorno, con las precauciones ya citadas. En otras implementaciones, se han tenido que modificar las rutinas de cambio de contexto para poder hacer desbancables los flujos de la aplicación. También se están estudiando otras versiones en las que se trabaja con el subsistema UNIX sin tener que recurrir al emulator [PATI93].

5.6.4. Los procesos nulos de la librería de CThreads: los *waiters*

Cuando una aplicación se queda sin flujos propios para planificar la librería de Cthreads tiene prevista la utilización de unos flujos “degenerados”: los *waiters*.

Decimos degenerados, porque carecen de todas las condiciones que cumplen los flujos del paquete. Para nosotros, la más importante es que no tienen identificación propia. Su pila es diferente también a la del resto de flujos, incluso el tamaño que por defecto se les asigna es menor que la de un cthread.

Todo ésto es lógico por la funcionalidad que la librería les da. Los ofrece como flujos nulos y su único trabajo consiste en retener el procesador virtual que le ha proporcionado el kernel, hasta que algún flujo de la aplicación pueda ser seleccionado [GIL93].

Como la librería no es preemptiva, sería peligroso ocupar el tiempo de espera con un mecanismo de encuesta. Por otra parte es evidente que, ejerciendo de proceso nulo, no puede bloquearse a nivel usuario. Por eso ha de buscar apoyo en el kernel y lo hace a través del mecanismo de mensajes.

El bucle principal de un *waiter* consiste en hacer una llamada al sistema bloqueándose a la espera de recibir un mensaje. Este mensaje lo enviará la primera rutina que desbloquee un flujo y vea que hay posibilidad de planificarlo en un procesador virtual ocupado por un *waiter.*

Los *waiters* provocan un bloqueo en el kernel, lo que haría que continuamente estuvieran subiendo upcalls de bloqueo a la aplicación que tendría que sucesivamente ir creando *waiters*. Este mecanismo, de buscar soporte en el kernel, no nos sirve. Aún más, es totalmente inadecuado.

Hemos sustituido los *waiters* por nuestros propios flujos nulos, que trabajan sobre encuesta para no utilizar recursos del kernel. De este modo, no sólo evitamos una cadena de upcalls de bloqueo no deseada -y con ella una multiplicación de flujos nulos-, sino que nos independizamos totalmente del apoyo del kernel, y hacemos portable la librería a otros sistemas.

Puesto que ahora el flujo nulo siempre es elegible para ejecutar, hemos proporcionado un flujo nulo para cada procesador que tenga asignado la aplicación. No están en la cola de preparados de la aplicación, sino que se ponen en marcha directamente, por handoff[16] (Figura 5-11).

En el siguiente capítulo veremos la realización de este entorno y las decisiones ligadas a ella que se han tomado o modificado.

16. Seguimos aquí la filosofía de Mach.

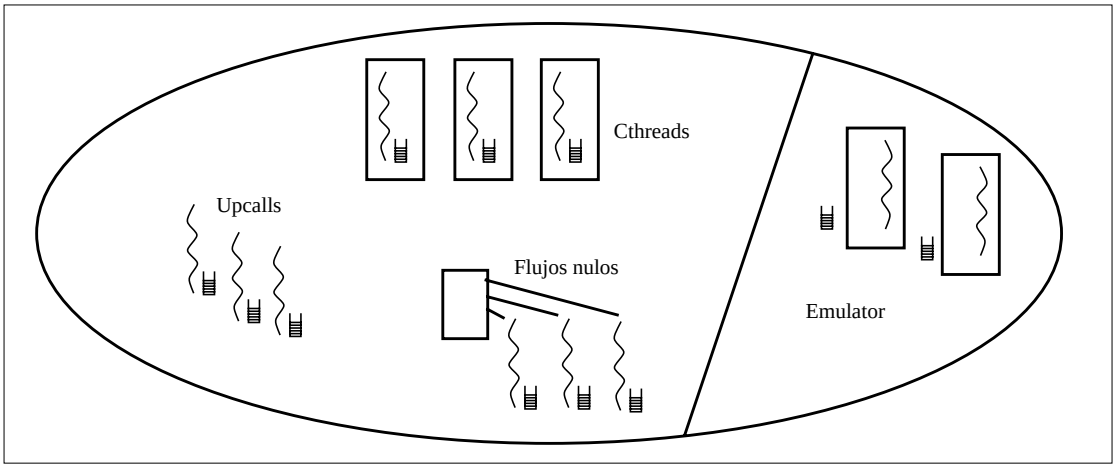

Figura 5-11 Diferentes flujos que hay en la aplicación, sustituyendo los *waiters* por flujos nulos capaces de ser reconocidos por la aplicación.

5.7. Referencias y Bibliografía

[ANDE90] “Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism”
Thomas E. Anderson,Brian N. Bershad,Edward D. Lazowska and Henry M. Levy
Technical Report 90-04-02, Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195, Technical Report 90-04-02, also in Operating Systems Review Vol.25 Num.5 October 1991.

[BART92] “Adding Scheduler Activations to Mach 3.0"
Paul Barton-Davis,Dylan McNamee,Raj Vaswani and Edward D. Lazowska
Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195
Technical Report 92-08-03, Revised October 1992.

[DRAV91] “Using Continuations to Implement Thread Management and Communication in Operating Systems”
Richard P. Draves, Brian N. Bershad, Richard F. Rashid and Randall W. Dean
Technical Report CMU-CS-91-115R, October 1991
also in ACM OSR Vol. 25 Num. 5, October 1991.

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

[JULI91] “Generalized Emulation Services for Mach 3.0. Overview, Experiences and Current Status”
Daniel P. Julin, Jonathan J. Chew, J. Mark Stevenson, Paulo Guedes, Paul Neves and Paul Roy
Proceedings of the Second USENIX Mach Symposium, Monterey, pp. 13-26, November 1991.

[MARK92] “Memory-Conscious Scheduling in Shared-Memory Multiprocessors”
Evangelos P. Markatos and Thomas J. Leblanc
Technical Report, Computer Science Department, University of Rochester, May 1992.

[MARS91] “First-Class User-Level Threads”
Brian D. Marsh et al.
ACM OSR, Vol. 25 Num. 5, October 1991.

[PATI93] “Redirecting System Calls in Mach 3.0: An Alternative to the Emulator”
Simon Patience
Proceedings of the Third USENIX Mach Symposium, 1993.

[TUCK89] “Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors”
Andrew Tucker and Anoop Gupta
ACM Operating Systems Review, Vol. 23 Num.5, December 1989.

6

Realización de los contextos de ejecución

"In order to be brilliant, you gotta be in touch with the universe"

Kevin Klein ("January Man")

ABSTRACT: Inherente a un entorno de las características del que estudiamos -multiprocesador, con aplicaciones multiflujo, realizando trabajos en paralelo-, está el solapamiento de varias acciones, de las mismas y diferentes características: es posible que dos flujos de una misma aplicación sufran un bloqueo al mismo tiempo, o en momentos muy cercanos; o que varios flujos provoquen una ráfaga de replanificaciones en la aplicación, por motivos diversos.

También es posible que, dadas las optimizaciones que hay en el código interno del kernel o debido, incluso, a la propia velocidad de los componentes hardware, la aplicación no tenga tiempo de gestionar un evento antes de que haya finalizado.

Ésto nos ha llevado a replantearnos algunos puntos de nuestra realización y proponer un diseño más elaborado, de acuerdo a los casos reales que acontecen en un sistema paralelo.

En este capítulo presentamos las situaciones que nos hemos encontrado y la solución que presentamos en respuesta; las complicaciones que pueden surgir y algunas propuestas realizadas para su solución; cómo puede afectar a nuestro entorno las optimizaciones existentes en el kernel de Mach, sobre el cual hemos hecho el desarrollo, y otras propuestas de mejoras a la realización actual que quedan como líneas abiertas para los que vengan detrás. También comentamos al final otras situaciones que existen en otras realizaciones de entornos similares y que nosotros, por el diseño que hemos realizado, no nos encontramos.

Capítulo 6: Realización de los contextos de ejecución

6.1. Introducción

En un entorno paralelo, las posibles colisiones e interferencias entre acciones son, a veces, de un orden superior al que, en un estudio sobre papel puede predecirse. A la hora de realizar el diseño de una solución, sea del tipo que sea, que tenga que llevarse a cabo en un entorno de estas características hay que aislarse, abstraerse, del sistema físico con el que vamos a trabajar. No podemos atar las ideas a las realizaciones si queremos que realmente supongan una base de desarrollo verdadera y válida en cualquier situación y entorno.

Hay que pensar un diseño teórico que no dependa de una realización concreta. No obstante, las ideas se han de llevar a cabo sobre una base física y plasmarlas en un soporte material para ver que realmente son efectivas y que la solución es real, funciona.

Y en este transporte o aplicación de las soluciones, nos sorprenden elementos que no habíamos tenido en cuenta y nos llevan a un feedback. Aquí se batalla realmente la prueba de fuego de la idea como solución válida. Permite remodelar el diseño, dar marcha atrás en algunas realizaciones, salir -o entrar- en situaciones que no se habían contemplado. La idea base no ha de modificarse, ha de ser general a una situación; y las soluciones, para ser buenas, han de reflejarse en cada caso particular.

Volvamos a situarnos en el entorno de trabajo que hemos dejado explicado en el capítulo anterior. Tenemos una aplicación con tantos procesadores virtuales como procesadores físicos le ha dado el sistema; cada procesador virtual representa de manera única a un procesador físico; y el sistema lo ofrece como contexto de ejecución (de ahí el nombre que les hemos dado, eXc) a la aplicación para que pueda planificar en ellos sus flujos.

La aplicación toma sus decisiones de planificación sobre sus flujos a partir de llamadas a la librería. Puede también tomar decisiones de planificación ante acontecimientos externos a ella que le notifique el kernel, como puede ser la variación en el número de procesadores que tiene asignados, o temporizaciones.

El kernel y la aplicación trabajan en consorcio, en igualdad de condiciones respecto a las decisiones que afectan a la ejecución de los flujos de la aplicación. En un entorno de trabajo así definido, la aplicación siempre sabe cuáles de sus flujos están efectivamente corriendo y sobre qué procesadores físicos.

Vamos a ver cómo llevar a cabo esta filosofía de trabajo con la librería CThreads^{+} y sobre Mach y los servidores que corren sobre él. Básicamente, OSF/1 MK y el CPU *server* que nosotros mismos hemos construido. Qué soporte, en cuanto a estructuras de datos, e información hay que mantener e ir actualizando; en qué momentos se actualiza y quién la actualiza.

Pasaremos luego a ver la gestión que se lleva en la aplicación de la información compartida con el kernel. Nuevos mecanismos de exclusión mutua para optimizar la inserción y extracción de flujos en la cola de preparados y nuevas primitivas para evitar el abrazo mortal al desbancar a un flujo en posesión de un *lock*.

Asimismo, veremos cómo gestiona el kernel aquellos acontecimientos que la aplicación deja en sus manos; es decir, que no suponen ningún aviso al usuario, suceden de modo "transparente" a la aplicación, pero manteniendo el entorno de trabajo con eXc y, por tanto, la información actualizada.

Por último, veremos el estudio dinámico de los diferentes eventos que puede gestionar la aplicación a partir de las *scheduler activations* producidas en el kernel. Cómo se lleva a cabo esta comunicación y cómo pueden interaccionar entre ellas. Veremos mecanismos de cortocircuíto para mejorar el rendimiento de la aplicación, cuando los acontecimientos llegan a una frecuencia mayor de la que ella puede absorver.

6.2. Estructuras de datos utilizadas.

En entornos experimentales se está trabajando y estudiando el mecanismo de upcalls con s-a, manteniendo un mapeo 1-1 entre procesadores virtuales y procesadores físicos. Los más conocidos son los de la Universidad de Washington sobre Topaz [ANDE90] y sobre Mach [BART92].

El objetivo básico en ambos estudios ha sido el comprobar que era posible un entorno en el cual el kernel subiera la gestión de los flujos a la aplicación ante determinados sucesos. Como ya hemos anotado con anterioridad, ellos dan la misma denominación de *scheduler activation* tanto al momento en que ocurre el evento a notificar, como el procesador virtual que sube dicho aviso; a este último nosotros lo hemos diferenciado con el nombre de eXc. El rendimiento que obtienen es bastante bajo, pero no era una de las finalidades de su estudio.

Nosotros hemos mantenido el mismo objeto de procesador virtual que nos ofrecía el sistema (en nuestro caso threads de kernel), para utilizarlos como eXc.

En la aplicación hemos trabajado, como ya se ha explicado anteriormente, con threads de usuario; en concreto con la librería de Cthreads^{+}. Resaltamos aquí que, aunque podría trabajarse con cualquier entorno que ofrezca la abstracción de flujo a nivel de usuario, de modo que se pueda trabajar con él de una manera independiente y automática, conviene que este entorno tenga una cierta riqueza y variedad en la posibilidad de gestionar los flujos. Si no fuera así, se perdería toda la eficacia y eficiencia de la planificación de acontecimientos. Como mínimo, que ofrezca los que tenía el kernel. Un usuario o programador experimentado puede, además, mejorar las utilidades.

Además, el usuario ha de proporcionar las rutinas de tratamiento de los eventos que quiere que el sistema le notifique y la memoria que utilizará cada procesador en la activación de estas rutinas, como pila de trabajo.

A partir de este esquema básico, vamos a ver a continuación qué estructuras de datos ha de mantener el kernel y qué información gestiona. Igualmente, la propia aplicación.

6.2.1. Información que mantiene el sistema

Mantenemos como procesadores virtuales los threads de kernel de Mach. Pero para que puedan funcionar como contextos de ejecución que se proporcionan al usuario, hay que modificar las estructuras de datos originales de modo que mantengan más información.

En concreto, un procesador virtual nos ha de indicar:

- el procesador físico en el que está trabajando,
- el identificador de eXc que le corresponde, que es único en la vida del sistema[17], y
- si ante la s-a actual, se está comportando como thread tradicional, gestionando el evento el kernel; o si, por el contrario, ha de subir la notificación a la aplicación.

También hemos visto que el kernel trataba el concepto aplicación como pset (en Mach en concreto), por encima del número de tasks que la formen. Y la aplicación ha de mantener la información global y nos dirá:

- si trabaja con el modelo de eXc,
- qué rutinas de nivel usuario (upcalls) suministra para el tratamiento de cada acontecimiento que el kernel le notifique,
- de dónde debe el kernel coger memoria para las pilas que se utilizan en la ejecución de dichas rutinas.

La Figura 6-1 nos muestra los cambios realizados para mantener el concepto de aplicación

17. Recordemos que cada upcall supone la activación de un nuevo eXc, diferente de todos los anteriores. Esto no significa que no haya reutilización de los objetos de kernel -threads- que los representan.

y la información necesaria para poder soportar su nuevo modelo de trabajo:

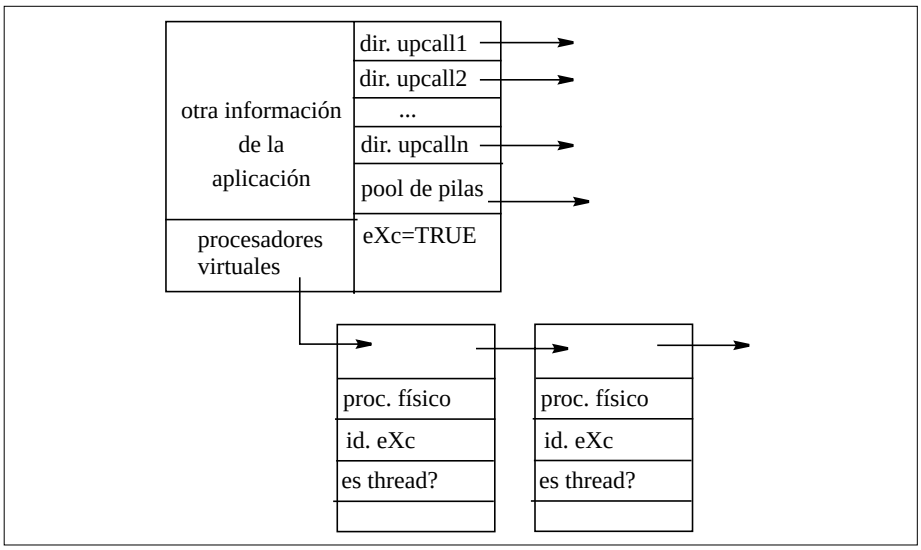

Figura 6-1 Estructuras de datos a nivel de kernel con la nueva información que han de mantener.

6.2.2. Información que mantiene el usuario

La aplicación tiene ahora un control absoluto sobre la ejecución de sus flujos y los recursos y objetos implicados en ello: qué flujo está corriendo, sobre qué procesador físico y en qué contexto de ejecución. Además es la aplicación quien tiene que gestionar el pool de memoria referente a las pilas que el kernel utilizará cuando los eXc salten a modo usuario al ejecutar las upcalls.

Hemos basado el diseño de la gestión de ejecución de las upcalls en el modelo de interrupción y hemos asignado una pila de usuario por cada procesador que el sistema otorga a la aplicación.

Con información compartida entre el kernel y el usuario y mecanismos de upcall para notificar eventos, existe principalmente la realización del entorno de trabajo en Psyche [MARS91].

Es un entorno experimental desarrollado desde cero, como ya hemos comentado con anterioridad. Destacamos sin embargo el soporte hardware que necesitan para acceder a memoria a recoger el estado del usuario, desde la propia aplicación.

Por estar trabajando en un entorno NUMA (BBN Butterfly), es importante mantener la localidad para obtener un buen rendimiento. Nosotros también opinamos que la aplicación ha de tener un control más potente sobre la planificación de sus flujos que las decisiones de elección del trabajo a realizar.

El control de la aplicación sobre la planificación de sus flujos pasa por el conocimiento de los recursos físicos sobre los cuales trabaja.

Esta información se mantiene en una tabla (Figura 6-1) con tantas entradas como el número máximo de procesadores que se pueden asignar a la aplicación manteniendo la informa-

ción correspondiente a cada procesador. El contenido de esta tabla es:

- identificador del procesador físico correspondiente. Si no tiene asignado dicho procesador, existe una marca de entrada inválida,
- identificador del eXc actual,
- identificador del flujo que está planificado en ese momento,
- dirección base de la pila en usuario para tratar las upcalls, y
- puntero al flujo nulo de dicho procesador.

También las estructuras de datos de datos utilizadas para los flujos de usuario (cthreads y cprocs, en nuestra realización) han tenido que ampliar sus campos de información. Así, por ejemplo, ahora mantienen el procesador virtual y el procesador físico sobre el que se están ejecutando, o se ejecutaron por última vez, en caso de no estar actualmente corriendo.

Esta información es necesaria en determinados casos, para identificar un flujo en concreto, a partir de la información que nos sube el sistema. En concreto, cuando ocurre un desbloqueo, sólo somos capaces de reconocer qué flujo de usuario era el que lo produjo, y hay que volver al estado de preparado para ejecutar, a través del identificador del eXc en el que corría cuando se bloqueó. Ahora nos lo devuelve el kernel en la upcall correspondiente.

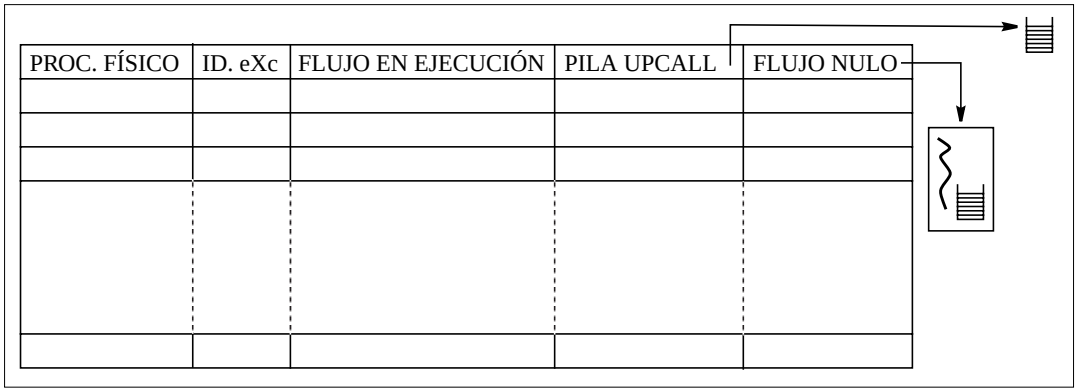

Figura 6-2 Tabla global de datos que se mantiene a nivel usuario

Recordemos que hay concurrencia a nivel de usuario y pueden haber pasado diversos flujos por el mismo eXc.

Todas las primitivas de la librería que realizan cambios de contexto han sido modificadas para actualizar convenientemente la tabla de estado global, y la información local de cada flujo.

Por otro lado, estamos manteniendo una información similar -equivalente- a la que manejan los kernels más actuales (Mach es uno de ellos). Así se permitirían políticas más potentes de planificación a nivel usuario, como puede ser de afinidad al procesador en que fue planificado por última vez, etc..., para explotar los aciertos en las memorias locales o caches de los procesadores físicos [MARK92], [CROV91].

6.2.3. Cola de flujos bloqueados en servicios del sistema

Para los flujos que están esperando un servicio del sistema, existe una nueva cola de bloqueados en

la aplicación.

En ella, están todos los flujos de los que se ha subido una notificación de bloqueo. La upcall de desbloqueo los sacará y volverá a insertar en la cola de preparados.

Hemos trabajado con modelo de *footprint* en la notificación de un desbloqueo: si el procesador sigue perteneciendo a la misma aplicación, el desbloqueo llega al mismo procesador en que se produjo el bloqueo. Para evitar contención, podríamos haber diseñado una cola de bloqueados por procesador.

6.3. Mecanismos de exclusión mutua entre la aplicación y el kernel

El compartir información sobre la planificación de los flujos y el hecho de que esta información pueda ser consultada, y en algunos casos modificada, tanto por el kernel como por la aplicación, hace necesario establecer unos mecanismos de trabajo en exclusión mutua.

También es importante notar que ahora los flujos de la aplicación (cualquier flujo) puede ser desbancable. Y un flujo desbancado podría estar precisamente en posesión de una exclusión mutua. Si no se tratan bien estos casos, podría acabarse en un abrazo mortal que implicara a kernel y aplicación.

Es importante recordar que, debido al aislamiento de cada aplicación respecto al resto en cuanto a la gestión y utilización de recursos, el mal comportamiento de una de las aplicaciones no afectaría al resto.

Vamos a ver, por parte de la aplicación, cómo se gestionan estas exclusiones mutuas.

6.3.1. Atomicidad en el tratamiento de upcalls

El código de una upcall se ejecuta de manera atómica, de modo que no puede interrumpirse su ejecución con la subida de otra notificación por parte del kernel.

La puesta en marcha de la upcall comienza en el interior del sistema, cuando se le construye el bloque de activación en la pila correspondiente. La aplicación, sin embargo es la que conoce cuándo acaba su ejecución, puede liberar la pila y pueden subir nuevos acontecimientos.

A través de un flag local a cada procesador, aplicación y sistema van coordinando el paso libre a nuevas notificaciones: el sistema es el que marca el flag como ocupado y la aplicación la que lo marca como libre.

Cuando llega una nueva notificación a un procesador, el sistema consulta el flag: si está ocupado, tiene que retrasar la notificación. Si está libre, lo marca como ocupado y empieza a preparar la subida a usuario del evento.

Para retrasar esta notificación la solución que hemos tomado es sencilla y supone un comportamiento similar al tratamiento de interrupciones. En el caso de estas últimas, el hardware permite que se inhiba el tratamiento de las interrupciones que lleguen mientras haya una en curso. Nosotros conseguimos el equivalente a través del mecanismo de AST.

Ésto es posible debido a los dos tipos de AST que maneja Mach: los que afectan al thread y los que afectan al procesador. Así, si el procesador no puede tratar en ese momento la upcall, “memoriza” el AST en el thread; la próxima vez que vuelva a entrar en código de kernel, volverá a intentar el tratamiento.

6.3.2. Atomicidad en la cola de preparados

Varias de las notificaciones que suben al usuario en las *scheduler activations* traen consigo nuevos estados de flujos que hay que actualizar. Pasan a ser elegibles por la aplicación para ejecución; unos, porque han acabado el servicio que les mantenía bloqueados; otros, porque estaban en ejecución en ese mismo procesador y han sido desbancados para subir la notificación que se halla en curso. La aplicación se plantea en ese momento un cambio de sus flujos activos.

Ahora, además de las primitivas de cambio de contexto de la librería, también las upcalls están compitiendo en el acceso a la cola de preparados, que se ha de hacer en exclusión mutua.

Es crítico para el rendimiento global de la aplicación que una upcall tenga que esperar en la entrada de una exclusión mutua, por el hecho de su ejecución atómica: puede estar reteniendo la subida de eventos que ha de notificar el sistema, e incluso puede retener la ejecución de otras upcalls que estén intentando acceder a la cola de preparados desde otros procesadores, acabando en un cuello de botella significativo. Lo adecuado, sería que no tuviera que esperar nunca.

Aprovechando la existencia de los flujos nulos de la aplicación, encargados de esperar por la aparición de flujos planificables y siguiendo un modelo ya tradicional que se utiliza para la planificación de procesadores virtuales en el kernel, hemos decidido pasar control directo desde la upcall al flujo nulo de su procesador para que sea él el que compita por el acceso a la cola de preparados.

Los flujos nulos de la aplicación son los encargados de tomar las decisiones de replanificación y acceder a la cola de preparados. Se evita así que las upcalls tengan que competir por un *lock* y provocar la caída de rendimiento de la aplicación.

No obstante, los contextos que suben actualizados desde el sistema en el bloque de activación de la upcall han de ser insertados también en la cola de preparados como candidatos a ser ejecutados.

Una posibilidad, hubiera sido pasar directamente el bloque de activación al flujo nulo y que él se encargara, tanto de extraer de la cola como de insertar en ella. Esta solución, sin embargo, supone una complicación en el diseño de los flujos nulos, que han de actuar de manera diferente dependiendo de la upcall que les esté dando paso. También se dificulta la gestión de las pilas que hay disponibles para el sistema.

Estudiando las diferentes filosofías en cuanto a los mecanismos de colas de preparados existentes [ANDE89] y las ventajas e inconvenientes de cada uno de ellos, hemos planteado una solución elegante y sencilla que lleva a una gestión óptima de la cola de preparados. Tanto en cuanto a la retención de los flujos que intentan acceder a planificar, como en cuanto al balanceo de ocupación de los procesadores.

Los dos mecanismos básicos para gestionar los flujos preparados para ejecutar son mantener una cola de preparados global o mantener una cola de preparados local a cada procesador.

Con una cola de preparados global se consigue el balanceo perfecto de la carga de la aplicación, a costa de convertir el acceso a la cola en un cuello de botella. Con una cola propia para cada procesador, se consigue eliminar el cuello de botella, con el peligro de tener procesadores sin trabajo y otros con colas de trabajo largas.

Nuestra solución ha consistido en considerar la inserción en la cola de preparados como local a cada procesador y la extracción de la cola, como global al sistema.

Los flujos de la aplicación, nulos o no, que son los que se encargan propiamente de replanificar, siguen teniendo que competir por el acceso en exclusión mutua a la cola.

Las upcalls, que ahora sólo se encargan de insertar en la cola pero no de extraer, no necesitan ninguna espera y, por tanto, acaban rápidamente su ejecución. Recordemos que los flujos nulos no se hallan en la cola de preparados y su planificación pasa por un *handoff* directo.

Pasemos a ver con más detalle la realización de este nuevo mecanismo y las estructuras de datos que hemos necesitado.

6.3.2.1. Inserción en la cola de preparados sin necesidad de exclusión mutua

Cada procesador tiene su propia cola de inserción de flujos preparados para ejecutar. Este hecho lo hemos representado a través de una tabla con tantas entradas como procesadores posibles pueda tener la aplicación. El esquema y pseudocódigo de la rutina de inserción en las colas locales de preparados puede verse en la Figura 6-3.

Cada vez que haya que insertar un flujo en preparados, se accederá a la entrada correspondiente al procesador en que se está realizando la operación, se buscará el primer sitio libre, y se dejará allí el flujo.

```
int eXc_ready_queue_insert (thread, processor)
cproc_t thread;
unsigned int processor;
{
Busca una posición libre el la fila de la tabla asociada al procesador:
    ready_table[processor][i] = thread;
}
```

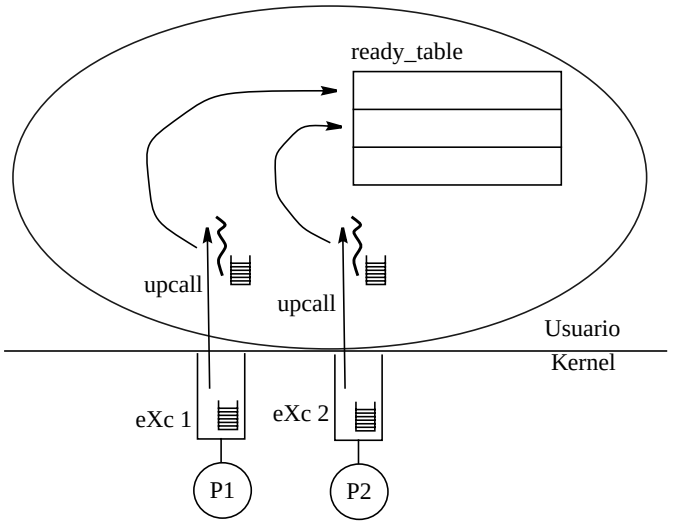

Figura 6-3 Inserción en la cola de preparados. Cada procesador busca una posición en su propia fila de la tabla, por lo que no es necesario ningún protocolo de exclusión mutua.

Con esta filosofía hemos mejorado incluso el rendimiento en algunas operaciones de la

librería, como las de desbloqueo de flujos, que ahora no tienen que competir por la cola de preparados. Entran aquí mutex_unlock(), condition_signal() y condition_broadcast().

Evidentemente, esta operación se ha de llevar a cabo en exclusión mutua, es decir de manera ininterrumpible: no puede desbancarse un flujo que está insertando en la cola de preparados. Si la operación la realiza una upcall, no hay ningún problema, puesto que ella misma no es interrumpible. Si lo realiza cualquier otro flujo de la aplicación, deberá activar el flag de no interrumpible. Como ahora, tanto el kernel como la aplicación pueden marcar este flag como ocupado, se ha de contar con el hardware apropiado para ello[18].

6.3.2.2. Extracción de la cola de preparados en exclusión mutua

La aplicación posee una cola global de extracción de flujos preparados para ejecutar que ha de ser accedida en exclusión mutua. Esta operación sólo la realizan flujos de la aplicación reconocidos por la librería.

Está claro que la cola como estructura de datos que contiene flujos preparados que pueden ser seleccionados para extraer, está formada no sólo por la cola original de la librería, sino que queda extendida a la tabla con las colas locales de cada procesador.

Cuando alguien tenga la exclusión mutua de la cola de preparados (y recordemos que sólo está compitiendo por ella si va a seleccionar un flujo para sacarlo), se encargará de pasar primero todos los flujos de las colas de inserción locales a la cola de extracción global. Y luego, seleccionará el flujo conveniente. Así garantizamos que todos los flujos que pueden ser planificados (están preparados) tienen realmente la oportunidad de ser elegidos y evitamos problemas de inanición (Figura 6-4).

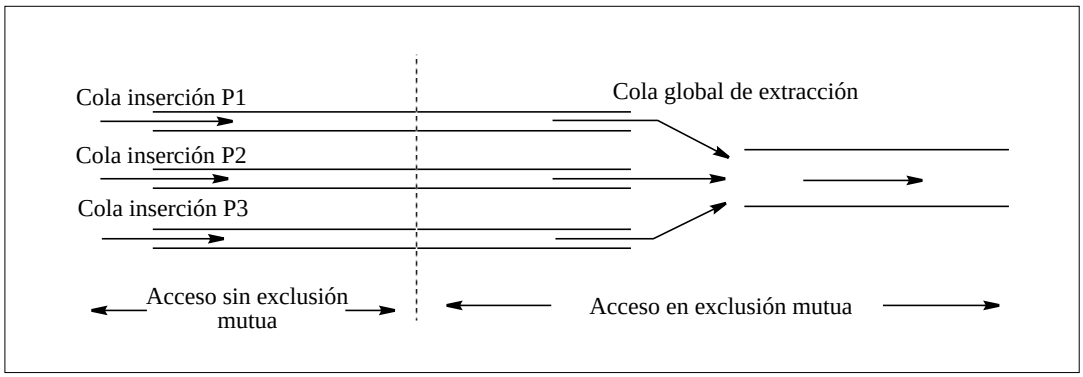

Figura 6-4 Esquema de extracción de la cola de preparados, extendida ahora a la cola global de extracción más las colas locales de inserción.

La aplicación ha de asegurarse de que siempre va a haber sitio para insertar en una cola local de preparados.

6.3.2.3. Desbanque del flujo nulo

Cuando la subida de una notificación supone el desbanque de un flujo, puede ser que el flujo desbancado sea el flujo nulo de ese procesador. Ésto puede ser así, bien porque no haya flujos planificables en la aplicación, o bien porque esté ejerciendo de planificador de una upcall anterior.

Si seguimos el comportamiento que hemos propuesto, como flujo desbancado, lo insertaríamos en la cola local de preparados, comportamiento erróneo, ya que podría llegar a planificarse

18. Basta con que la escritura en memoria sea atómica, puesto que el flag es local a cada procesador.

a través de la política de planificación de la aplicación. Como es un flujo que nunca se bloquea, caeríamos en un comportamiento anómalo en la aplicación.

Por otra parte, la upcall tiene que pasar control al flujo nulo del procesador y este control lo pasa en directo (mecanismo de *handoff*): conoce el identificador de su flujo nulo. Por ello puede saber si el flujo desbancado ha sido el nulo y pasarle el control allí donde quedó suspendido.

La Figura 6-5 muestra el algoritmo de comprobación del flujo desbancado que realiza cada rutina de upcall.

```
upcall_xxx:

salvar contexto desbancado en la pila del flujo;

si el flujo desbancado no es el nulo
       eXc_ready_queue_insert(preempted_thread, processor);
sino
        /* La upcall ha desbancado al flujo nulo
           del procesador actual */
       pasar control al flujo desbancado donde se quedó;
fsi;
activar al nulo de ese procesador;
```

Figura 6-5 Esquema del código que ejecuta una upcall para comprobar si el flujo nulo ha sido el desbancado.

6.3.3. Abrazo mortal por desbanque de un flujo

Al convertir la librería en preemptiva, existe la posibilidad de desbancar un flujo que esté en posesión de un *lock*. En principio, ese flujo volverá a ser planificado en otro momento posterior y por tanto, acabará liberando esa exclusión mutua.

Si la aplicación dota a sus flujos con diferentes prioridades, dado que nuestra política es de prioridades estáticas, podría correrse el peligro de que un flujo fuera desbancado en el interior de una exclusión mutua y, por existir flujos de mayor prioridad, no volviera a planificarse nunca, cayéndose así en un abrazo mortal. Este comportamiento, sin embargo, lo decide la aplicación. Por defecto, todos los flujos están trabajando a la misma prioridad.

Es responsabilidad de la aplicación, si trabaja con flujos a diferentes prioridades, hacer que se acceda a una exclusión mutua desde la prioridad adecuada, definiendo un orden en la adquisición de *locks*.

Es importante, de nuevo, volver aquí al concepto de aplicación como conjunto de flujos que cooperan a un fin. En algún momento de la ejecución de un flujo, su trabajo depende del de otro, ya sea por una sincronización, por la espera de un resultado,... Por tanto, pensamos que un flujo que no esté en ejecución y posea un *lock*, acabará quedando como único flujo planificable de la aplicación. Evidentemente, tardar tanto en salir de la exclusión mutua iría en detrimento del rendimiento de la aplicación.

El único caso que tenemos el deber de evitar en el diseño de nuestro entorno de trabajo es la exclusión mutua de la cola de preparados, puesto que es una gestión que se hace desde la librería. Si un flujo fuera desbancado mientras está trabajando con la cola de preparados y, por tanto,

con la exclusión mutua correspondiente cogida se caería en un abrazo mortal irrecuperable.

La otra cola gestionada a nivel de librería es la cola de flujos bloqueados en servicios del sistema. A ésta sólo se accede desde el código de una upcall (de bloqueo o desbloqueo). Al ser código no interrumpible, no hay peligro de caer en un abrazo mortal.

6.3.3.1. Desbanque de un flujo con la exclusión mutua sobre la cola de preparados cogida

Cuando la upcall pasa el control al flujo nulo para planificar, éste intenta coger el *lock* de la cola de preparados. Si ya está cogido, accede únicamente a la cola local de inserción y lo vuelve a intentar más tarde. Sin embargo, puede ser que el *lock* esté en posesión de alguno de los flujos desbancados que está en las colas de inserción y, por tanto, no se liberará nunca por este camino.

Del mismo modo que la upcall pasaba control al flujo nulo cuando éste era el desbancado, ahora el nulo tendría que ver si uno de los flujos que ha de insertar en preparados posee el *lock* de la cola global de preparados para pasarle directamente el procesador. Como además es un flujo que está planificando, él mismo se encargará de poner al flujo que corresponda en ese procesador, dependiendo de la política que utilice la aplicación.

Pasar el control directamente a un flujo desbancado plantea varias preguntas. La primera es obvia: hay que saber quién está en posesión del *lock*. Para ello hemos tenido que modificar la primitiva de *spin lock* en el caso de la cola de preparados. La nueva realización la comentamos en el siguiente punto.

Una vez tenemos la identidad del posesor del *lock*, hay que ver en qué procesador estaba corriendo. Si está en la cola de inserción de otro procesador, significa que hay otro procesador tratando una upcall (único código capaz de desbancar) y su flujo nulo se está cuestionando la misma pregunta: dejemos que sea él el que ponga en funcionamiento al flujo. Si está en nuestra propia fila, lo haremos nosotros.

Un flujo desbancado dentro de la exclusión mutua de la cola de preparados será puesto de nuevo en funcionamiento por el flujo nulo de su procesador.

Con esta política evitamos la contención y podemos acceder a la cola de inserción sin ningún tipo de exclusión mutua.

Se puede extraer un flujo de la cola local de preparados sin necesidad de protocolo de exclusión mutua.

6.3.3.2. Nueva primitiva de *spin lock*: marcar quién está en posesión del *lock*

La cola global de preparados cuenta ahora con una nueva primitiva de *spin lock* en la que queda registrado el flujo que tiene la exclusión mutua sobre ella.

Para dejar marcada su identidad tiene que acceder a la variable global correspondiente de manera ininterrumpible. Utilizamos para ello el mismo mecanismo que para las rutinas de upcall: el flag local de cada procesador. El algoritmo de la primitiva eXc_spin_try_lock() se detalla en el Apéndice C.

6.3.3.3. Sustracción de un procesador ¿Quién hereda sus flujos?

Cuando se sustrae un procesador a la aplicación, junto al flujo que estaba corriendo en el procesador que sube la notificación -llamémosle P1- sube el flujo que estaba ejecutándose en el procesador sustraído -llamémosle P2-.

En este caso, hemos decidido que el propio P1 herede el flujo de P2: se le actualiza la información y se inserta en la cola local de P1.

¿Qué ocurriría si hubiera ya otros flujos en la cola de inserción de P2? Nuestro algoritmo de paso de flujos de las colas locales a la cola global barre de una vez toda la tabla: no hay peligro de que queden flujos olvidados porque su procesador no pertenezca ya a la aplicación.

Si cualquiera de los dos flujos subidos en la notificación y que ahora son de P1, tuviera cogido el *lock* de la cola de preparados, siguiendo la política que hemos definido, sería el propio P1 el encargado de ponerlo en marcha.

Un caso especial a estudiar sería que el flujo heredado de P2 fuera el flujo nulo. En este caso, examinamos si tiene cogida la exclusión mutua sobre la cola de preparados. Si no es así, lo dejamos.

Si tiene cogida la exclusión mutua de la cola de preparados, hacemos un cambio de nulos entre P1 y P2, y pasamos control al flujo que ha de liberar la exclusión mutua (Figura 6-6).

Estas comprobaciones sobre el flujo nulo las hace la propia upcall.

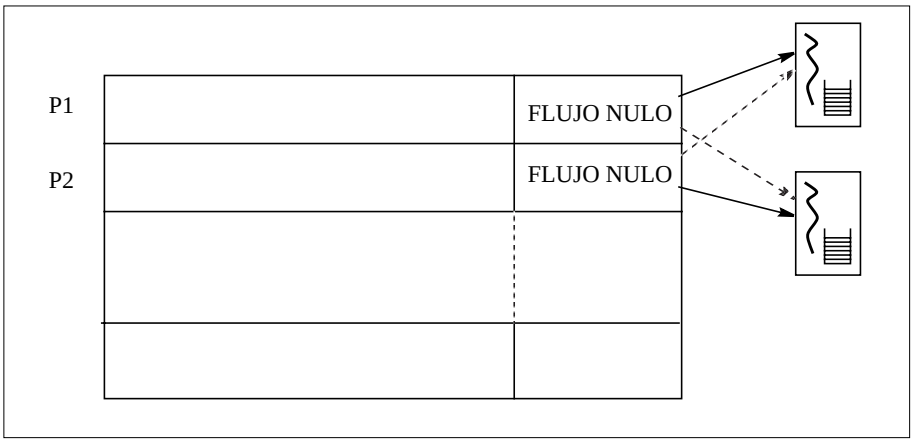

Figura 6-6 Intercambio de flujos nulos cuando el del procesador reasignado es el que tenía cogida la exclusión mutua sobre la cola de preparados: él es el que ha de continuar.

6.3.3.4. Thread nulo planificador

El flujo nulo consiste básicamente en un bucle infinito con dos funciones: ocupar el procesador virtual correspondiente -y, por lo tanto, también el procesador físico- e intentar una planificación si puede acceder a la cola global de preparados.

La primera vez que se pone en marcha es cuando se asigna un procesador a la aplicación. Se encarga de ello la rutina de tratamiento de la upcall de asignar procesador. Esta primera vez recibe como parámetro el procesador al cual pertenece. Después se queda en un bucle infinito de planificación.

```
sched_thread (processor):

 mientras (TRUE) hacer
     mientras (eXc_ready_table_to_ready_queue()) hacer
        recoger_who;
        si who estaba corriendo en mi procesador entonces
           recogerlo de la ready_table;
           pasarle el control;
        fsi;
     fmientras;
     eXc_sched_thread_suspend();
 fmientras;

eXc_ready_table_to_ready_queue():

intenta coger el lock de la ready_queue;
pasa todos los flujos de la ready_table a la ready_queue;
devuelve TRUE si el lock ya está cogido;

eXc_sched_thread_suspend():

pasa control al flujo correspondiente según la política
de planificación y actualiza la tabla global de estado;
```

Figura 6-7 Esquema de trabajo del flujo nulo de cada procesador.

6.4. Mecanismos de subida de notificaciones desde el kernel

Una vez reconocido en el kernel que hay que subir una notificación al usuario, podemos diferenciar dos momentos básicos en el tratamiento del aviso hasta que llegue al espacio de usuario: primero, activar la *scheduler activation* y preparar el eXc de acuerdo al evento que haya llegado, después construir el bloque de activación correspondiente y realizar la salida a usuario saltando a la upcall correspondiente.

La primera operación es la que hemos llamado activación de una *scheduler activation* (sa_activate()); la segunda corresponde a la salida del kernel de un eXc (exit_kernel_eXc()).

Hemos visto en el capítulo anteriores que podíamos clasificar las notificaciones en dos tipos: aquellas que producían la ejecución del propio flujo y aquellas que provenían de agentes externos a cualquiera de los flujos que estaban en ejecución en ese momento. Por decirlo de otra manera: las notificaciones que subían en su propio procesador y las que necesitaban desbancar a un procesador.

Vamos a ver a continuación, las herramientas que necesita el kernel en cada caso.

6.4.1. Activar una upcall en el propio procesador

Las upcalls que suben con procesador propio son las de bloqueo de un flujo y la de otorgar un nuevo procesador a la aplicación. Ambas se caracterizan por la subida de un nuevo eXc sin ningún contexto anterior, que se ofrece a la aplicación para que planifique un nuevo flujo.

En ambos casos, se comprueba si es realmente una *scheduler activation*. Básicamente, consiste en examinar que se está trabajando en un entorno de eXc y que la aplicación ha pedido gestionar dicho evento.

Si es un momento de *scheduler activation*, hay que subir la correspondiente notificación a

la aplicación, a la vez que se pone en marcha el servicio pedido.

En el caso de bloqueo, el eXc que está trabajando en ese momento tiene el contexto del flujo que está pendiente de ser servido y en el que ha de continuar antes de retornar al usuario: no es un punto seguro en su estado. Hay pues que proporcionar un nuevo eXc a la aplicación que notifique el evento y a la vez sirva como contexto de ejecución para planificar un nuevo flujo. Para ello, se crea una nueva estructura thread a la que se le actualizan los campos correspondientes. Entre ellos, se le construye una continuación explícita a la función exit_kernel_eXc(), a donde saltará una vez se realice el cambio de contexto con el procesador virtual que va a bloquearse.

El action thread de Mach es el encargado de añadir (y también quitar) nuevos procesadores a la aplicación.

Una vez que se ha realizado la operación, recordemos que el action_thread está vinculado al procesador en cuestión. Antes de dejarlo, mira si la aplicación a la que ha añadido el procesador está trabajando con modelo de eXc y, por tanto, acaba de producirse una *scheduler activation*.

Si es así, lo mismo que en el caso anterior, crea un nuevo eXc que notifique el evento a la aplicación y a la vez sirva como contexto de ejecución para el flujo que planifique en él. Se desliga después del procesador y realiza el cambio de contexto con el nuevo procesador virtual que tiene igualmente una continuación explícita a exit_kernel_eXc().

En exit_kernel_eXc() se construye la salida a usuario (bloque de activación y continuación a la rutina de tratamiento correspondiente) y finalmente se sale. El esquema está representado en la Figura 6-8.

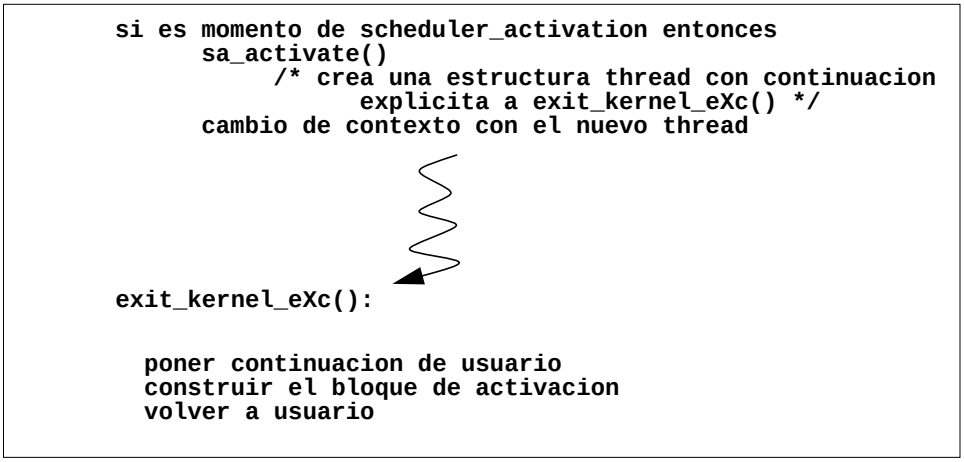

Figura 6-8 Esquema general para subir una notificación en el propio procesador en un entorno de eXc.

6.4.2. Desbancar un procesador para subir una notificación a la aplicación

Acabar el bloqueo de un flujo o quitar un procesador a la aplicación supone desbancar a un procesador de la aplicación del trabajo que está realizando para poder activar la *scheduler activation* correspondiente.

Desbancar a un procesador es forzarle a hacer un cambio de contexto. Nosotros hemos utilizado para ello el mecanismo de ASTs, ampliando el número de los que ya ofrecía Mach. El nuevo AST, es un AST de desbanque (AST_PREEMPT_PROCESSOR).

En este caso, el momento de reconocimiento de una *scheduler activation* no coincide con el inicio de su tratamiento. Cuando el procesador virtual llegue a tratar el AST, detectará el aviso y cederá el procesador físico al nuevo eXc (recordemos que cada subida de notificación implica un

nuevo eXc aunque no suponga creación de procesador virtual).

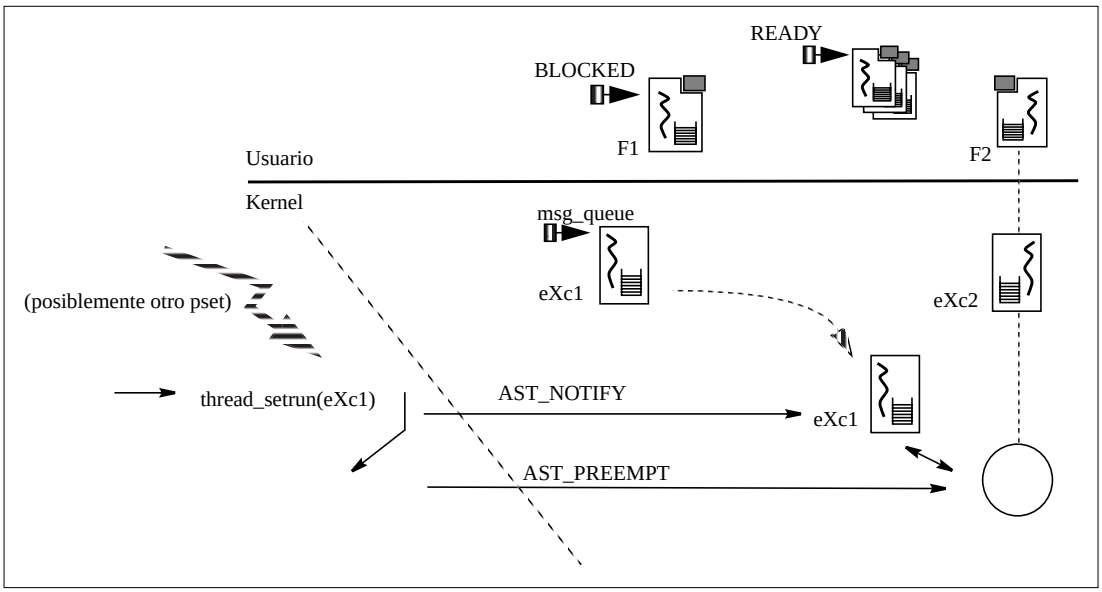

Figura 6-9 Secuencia de tratamiento en el kernel cuando ocurre un desbloqueo en un entorno de eXc.

Este momento será en un punto seguro: en la salida a usuario, como ya hemos comentado en el capítulo anterior.

El eXc al que acaban de dar el procesador empieza a finalizar su trabajo en el kernel: termina, si es el caso, el trabajo que le quede pendiente y prepara su salida a usuario.

Cuando se produce una upcall que supone desbanque de flujo, hay que subir junto con la notificación del evento, al flujo que se ha desbancado. La aplicación decidirá cuál de los dos flujos, o incluso un tercero, continua ejecutándose en ese procesador (Figura 6-10).

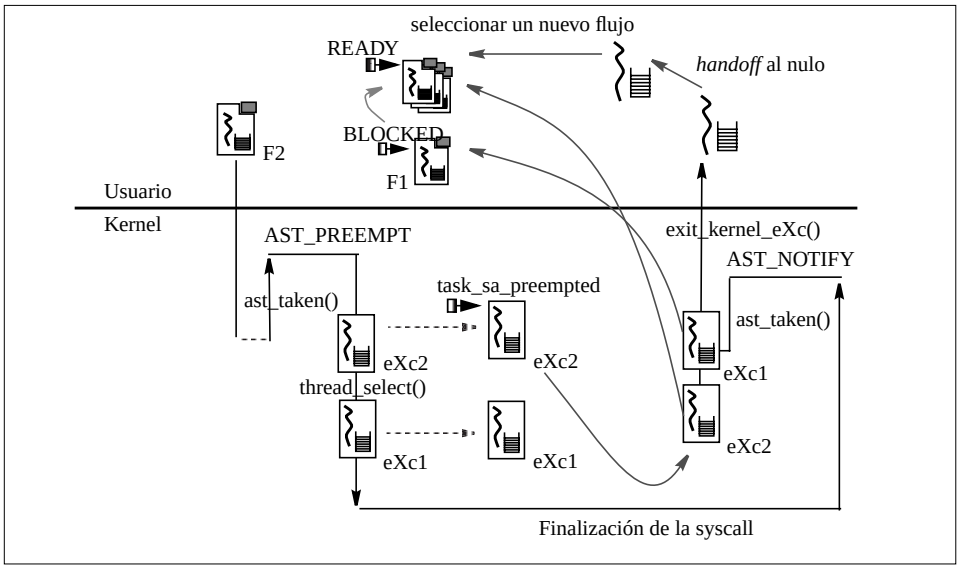

Figura 6-10 Subida del flujo desbancado y del flujo resumido, y elección de un nuevo flujo para ejecutar.

Para que la nueva notificación sepa que ha de subir otro eXc con ella, hemos utilizado otro

AST, esta vez al thread en lugar de al procesador: AST_NOTIFY, para notificarle que recoja al eXc que le ha cedido el procesador.

El tratamiento de AST_NOTIFY incluye, además de la recogida del flujo desbancado, el tratamiento de la *scheduler activation*, como hemos visto en el ejemplo del punto 5.4 y se muestra en la Figura 6-10.

6.5. Solapamiento de upcalls

Desde el momento en que se produce una *scheduler activation*, hasta que el usuario acaba su gestión, pasa un cierto tiempo. Si consideramos que estamos, además en un entorno paralelo, con múltiples procesadores trabajando a la vez para la misma aplicación, tanto dentro como fuera del sistema, este tiempo es lo suficientemente amplio como para que, antes de acabar el tratamiento de una notificación haya llegado otra.

Estamos trabajando con un modelo de interrupción y cada procesador trabaja con su propia pila para construir el bloque de activación de la upcall correspondiente. Desde el punto de vista del sistema, acontecimientos que llegan a procesadores distintos no interfieren entre ellos: cada uno gestiona su propia subida a usuario independientemente. Una vez en usuario, cada upcall hará su tratamiento correspondiente, con la coordinación que hemos comentado en puntos anteriores, si fuera necesario.

6.5.1. Atomicidad en el tratamiento de upcalls: retardar la subida de nuevos avisos

Si el nuevo aviso llega en un procesador en el que ya se está tratando una notificación, sería un acontecimiento que supone desbanque del procesador, puesto que las upcalls no piden servicio al sistema y menos si puede acabar en bloqueo[19].

El tratamiento de una upcall es ininterrumpible por lo que, en estos casos, lo que haremos será retrasar el tratamiento de la segunda upcall hasta que haya acabado la anterior. El punto de detectar este retraso necesario será en el tratamiento de los AST, puesto que es donde se produce el desbanque de un flujo. Por tanto, cuando llegue un AST_PREEMPT, habrá que consultar el flag de inhibición. Si está activado, se almacena el AST en el thread y se continua la salida a usuario: el flujo de usuario que se está ejecutando es una upcall y así la dejamos finalizar.

6.6. Repetición de s-a en el tratamiento de un mismo servicio

En el capítulo anterior hemos visto que cuando un flujo bloqueado a la espera de un servicio del kernel se desbloquea, se le notificaría al usuario en la finalización de dicho servicio, por ser ése un punto seguro de su estado. El momento del desbloqueo generalmente llegará mediado el servicio en el sistema.

Parte de este servicio, sin embargo, puede incluir otros bloqueos. Si se notificaran a la aplicación, ésta no sabría qué hacer: a ella no le ha llegado todavía la notificación de desbloqueo y por tanto, le estamos notificando el bloqueo de un flujo que para ella ya estaba bloqueado con anterioridad.

Es por ello que sólo se le notifica la primera vez que cae en un bloqueo: la aplicación lo tendrá apartado de ejecución hasta el final del servicio, cuando se notifique el desbloqueo.

19. Es evidente que no se trata de añadir procesador, puesto que ya estaba trabajando para la aplicación.

En los servicios que la aplicación pide sólo se notificará una vez el bloqueo -la primera vez que suceda- y otra el desbloqueo -cuando retorne a usuario-.

6.7. Gestión transparente al usuario: excepciones

La entrada de un flujo en código del sistema puede ocurrir, básicamente, por tres acontecimientos: por una petición suya de un servicio, por el tratamiento de una excepción o para servir una interrupción de un dispositivo. El primer caso, es un servicio pedido por la propia aplicación, los otros dos no. Sin embargo, todos ocurren en el contexto de un flujo de la aplicación.

Supongamos, por ejemplo, que sucede un fallo de página. En esta situación, también por unos momentos existe un bloqueo interno del flujo que ha provocado el fallo. Pero la aplicación no ha pedido ningún servicio, es una gestión interna del sistema que la aplicación no entiende, que no existe, en una palabra, para ella. Por este motivo, hemos decidido que, puesto que las excepciones no las trata ni las pide la aplicación, estos acontecimientos no le serán notificados.

Por otra parte, en el ejemplo que hemos utilizado, la aplicación no puede prever cuál es la mejor acción a tomar: qué flujo es el más adecuado, que no provoque otro fallo de página, por ejemplo.

Imaginemos, además, que este fallo de página es producido por el código del propio planificador de usuario: cada vez que se subiera el acontecimiento y que intentara gestionarlo, se provocaría el mismo fallo de página, saltaría al bloqueo,... sería una recursividad insalvable, que escapa además al control que la propia aplicación tiene del comportamiento y posibles bloqueos de sus flujos.

Todas las situaciones de bloqueo de un flujo ocasionadas por excepciones del sistema, no han de ser gestionadas por la aplicación. Ni tan sólo notificadas, ya que ocurren de una manera transparente a ella.

Diremos que estos bloqueos no ocurren en circunstancia de *scheduler activation*. Para ello, cada vez que llegue una excepción, pondremos una marca de “no *scheduler activation”* (no_sa) para indicar que, aunque llegue un acontecimiento de los que la aplicación ha dicho que trataría (bloqueo o desbloqueo, en nuestro caso), no va a concurrir en s-a y por tanto la gestionará el sistema.

Esta marca será quitada cuando acabe el servicio de la interrupción o excepción.

Como pueden llegar imbricadamente varias interrupciones de distinto nivel, siguiendo la filosofía de Mach para los *locks* [BLAC91] hemos convertido la marca en un contador: si el contador es superior a cero, no se reconoce una *scheduler activation*.

Esta marca es local a cada procesador virtual y siempre vale cero cuando se está trabajando código de usuario.

6.8. Fases de un eXc ejecutando código en modo kernel

Vistos los eventos que la aplicación quiere tratar ella misma y los momentos que son causa de activar una upcall o no, podemos decir que cada uno de los flujos de la aplicación va pasando durante toda su vida por dos fases de trabajo: fase de eXc y fase de thread.

Decimos que el flujo de una aplicación está en fase de eXc cuando su ejecución puede reconocer una *scheduler activation* y por tanto activar una upcall. En cualquier otro caso, diremos que trabaja en fase de thread.

Ejecutando código de usuario siempre se comporta como thread respecto a los acontecimientos del kernel (evidentemente, sólo pueden llegar en esta situación o una interrupción o una excepción).

Ejecutando código del sistema, sólo trabajará en fase eXc si se reconoce uno de los eventos que la aplicación quiere que se le notifiquen y se dan, además, las siguientes circunstancias:

- el evento no es provocado por una excepción que pueda haber llegado en medio del servicio, y
- no se ha notificado ya ese mismo evento en el tratamiento de ese servicio.

Estas dos fases del flujo quedan representadas en la Figura 6-11:

Los eventos que quedan en disposición de la aplicación y no dependen del estado del flujo (temporizadores, asignación y desasignación de procesadores) se notifican siempre si la aplicación quiere tratarlos.

Una situación de excepción puede llegar en cualquier momento, también mientras se está realizando una llamada al sistema. Entonces, pueden darse casos de imbricación de mecanismos, que suponen entrar como eXc, pasar a comportarse como thread, seguir como thread hasta que finalice la excepción y volver a pasar a eXc.

Así, vemos que una aplicación trabajando con eXc, puede comportarse, en diferentes momentos como eXc, y subirá una notificación, y como thread, y entonces los bloqueos los gestionará el kernel de manera transparente.

Vamos a estudiar con más detalle estas dos fases de la aplicación, cuándo pasa de una a otra y cómo se gestiona cada una de ellas.

6.9. Ordenación en el tratamiento de los nuevos ASTs

Una vez que se ha reconocido y tratado una *scheduler activation*, el procesador virtual pasa a comportarse como thread (Figura 6-11). Es decir, sólo se sube la notificación de un acontecimiento en cada activación de upcall. Ésto es así tanto para las que se producen en el propio procesador como las que necesitan pasar por un desbanque y tratamiento de AST.

Una vez que ya ha comenzado la construcción del bloque de activación para la upcall correspondiente, esta upcall ya es ininterrumpible hasta su finalización en usuario, con la gestión de flujos que haga la aplicación.

Cuando este inicio de upcall, y activación del flag de inhibición, pasa por el mecanismo de AST la secuencia es la siguiente: se entra en la rutina de tratamiento de los ASTs (ast_taken()) y se van comprobando uno a uno si ése es el que hay que tratar. No sólo para los que hemos añadido,

sino también los que ya había en el sistema.

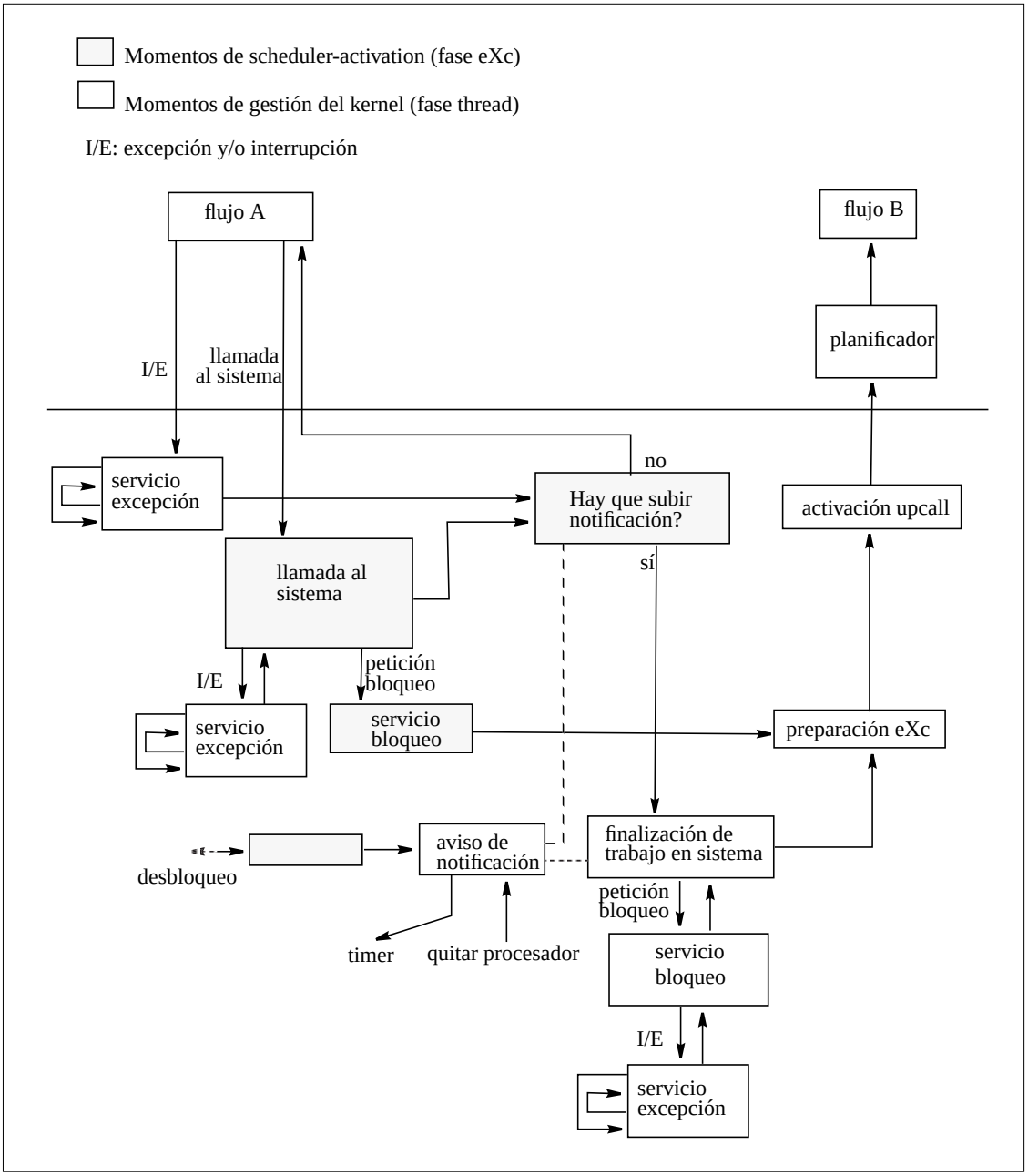

Figura 6-11 Fases por la que puede pasar una aplicación: fase eXc, más externa, y fase thread, en el interior del kernel.

Para pasar únicamente por uno cada vez y no subir más de una notificación, sólo tratamos el primero y los siguientes los “retrasamos” almacenándolos en el thread: la próxima vez que se entre en el sistema, se tratarán.

Si el orden de pregunta en la activación de los ASTs no es correcto, nos podemos encontrar con comportamientos no deseados. Así, si se ha producido un desbanque y estamos finalizando el trabajo en sistema, nuestro eXc tiene activado un AST_NOTIFY que no será tratado hasta la salida a usuario. Si en ese impasse se le programa un AST_PREEMPT, a la salida a usuario tendrá que tratar uno de los dos. Si el de desbanque fuera el primero, el resto se perdería ya

para siempre, puesto que el eXc desbancado se deshecha.

Lo mismo ocurriría si, en lugar de un AST_NOTIFY se tratara de un AST_TIMER.

Para no perder ninguna notificación, el orden correcto de tratar los ASTs es: AST_NOTIFY, AST_TIMER y AST_PREEMPT.

6.10. Algunas optimizaciones en la gestión de upcalls

La activación de la upcall pasa por dos momentos: uno, dentro del propio kernel, para asignar el eXc y construir el entorno que se va a subir al usuario (copia de registros, parámetros,...), y otro, ya en zona de usuario, cuando se ejecuta la propia upcall, hasta que se planifica un flujo de la aplicación. Dependiendo de en cual de las dos fases de tratamiento, kernel o usuario, estemos, podemos solucionar de maneras diferentes este atropello de notificaciones.

6.10.1. *Bypass* de notificaciones

Debido a las optimizaciones que se dan dentro del propio Mach para la gestión de mensajes y petición de servicios -*handoffs*- y también por la velocidad relativa de los componentes hardware -no es lo mismo pedir un servicio a través de la red, que pedirlo a una aplicación en la misma máquina-, puede ocurrir que una upcall de desbloqueo, llegue antes de que se haya tratado la notificación de bloqueo del propio flujo [BART92].

Cuando ocurre un bloqueo, la aplicación aparta al flujo correspondiente, marcándolo como bloqueado y selecciona a un nuevo flujo para ejecución. Si antes de acabar esta operación llega el desbloqueo, puede encontrarse con diversas situaciones:

- si la upcall anterior todavía no ha acabado de marcar como bloqueado al flujo, tendrá que esperar a que libere la exclusión mutua sobre la cola correspondiente para, acto seguido, deshacer la operación y poner al flujo como preparado para ejecutar, provocando además dos replanificaciones seguidas sobre el mismo procesador.
- si la upcall anterior ya ha acabado de marcar al flujo como bloqueado, deshará rápidamente la operación e, igualmente, se producirán dos replanificaciones seguidas.

En cualquiera de los casos, es evidente que se está produciendo una sobrecarga en la gestión y que, seguramente, si el usuario supiera que el bloqueo de su flujo es tan breve, no se plantearía un cambio de contexto, ni siquiera a nivel de usuario.

No todos los eventos que afecten a la ejecución de un flujo deben notificarse a la aplicación. Sólo aquellos, cuya duración sea lo suficientemente grande como para que la aplicación pueda gestionar el tiempo activando otro trabajo.

Para poder cumplir esta afirmación, tendríamos que saber lo que va a durar un bloqueo, con antelación a que suceda, y generalmente ésto no nos es posible. Por ello,

PROPONEMOS iniciar el aviso de todos los bloqueos que acontezcan en fase eXc. Pero **si se solapan los avisos de bloqueo y desbloqueo, anularemos el aviso de bloqueo que hay en curso y pasaremos a ejecutar en fase thread, de modo que tampoco se activará la notificación de desbloqueo**.

Lo mejor, pues, es cortocircuitar la operación de bloqueo, de manera que al usuario no le llegue ninguna notificación: no hay tiempo suficiente entre el bloqueo y desbloqueo para que la aplicación pueda aprovecharlo poniendo en marcha a otro de sus flujos. Y en esta situación, tener que gestionar los dos avisos (el de bloqueo y el de desbloqueo), sólo supondrían una sobrecarga de gestión.

Sin embargo, no siempre puede cortocircuitarse el tratamiento. Hemos visto que, la llegada del desbloqueo de un flujo puede llegar en diferentes momentos del tratamiento del bloqueo de ese mismo flujo. Para no dejar a la aplicación en un estado inestable, a veces hay que esperar a que termine el bloqueo. Este momento lo marca la inserción en la cola de bloqueados del thread en cuestión. Una vez que se ha iniciado la modificación del estado de la cola de bloqueados, hay que finalizar la operación hasta quedar en un valor estable. Para ello, hemos previsto un tercer valor en el flag de inhibición de una upcall que es “bypass”: mientras se está en este estado, la upcall puede ser anulada; si ya se ha pasado a no interrumpible, hay que esperar al final y subir el siguiente aviso.

6.10.2. Reciclaje de procesadores virtuales

Este solapamiento en la llegada de avisos a la aplicación viene dado en parte por los diferentes tiempos de tratamiento que requiere cada upcall, tanto dentro del kernel como ya en la propia aplicación[20].

Para mejorar el rendimiento de la aplicación en la gestión de eventos, y siguiendo una política ya utilizada por los paquetes multiflujo para sus objetos, hemos reciclado los objetos de kernel.

Los procesadores virtuales que se liberen al desbancar un flujo quedarán en posesión de la aplicación. Cuando se produzca una s-
a y necesite un nuevo procesador virtual para el nuevo eXc, la estructura ya estará preparada para la subida.

De este modo, se han mejorado mucho los tiempos de gestión en el interior del kernel y, como consecuencia, el rendimiento de la aplicación, ya que con un pequeño pool de threads de kernel necesarios para las primeras notificaciones, si ocurren en paralelo, el resto de ellas tiene un coste de asignación de eXc cercano a cero.

6.11. Referencias y Bibliografía

[ANDE89] “The Performance Implications of Thread Management Alternatives for Shared-

20. Se verán los tiempos correspondientes en el Capítulo 7 “Evaluación....”

Memory Multiprocessors”
Thomas E. Anderson et al.
Performance Evaluation Review Vol.17 Num.1, May 1989 172.

[ANDE90] “Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism”
Thomas E. Anderson,Brian N. Bershad,Edward D. Lazowska and Henry M. Levy
Technical Report 90-04-02, Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195, Technical Report 90-04-02, also in Operating Systems Review Vol.25 Num.5 October 1991.

[BART92] “Adding Scheduler Activations to Mach 3.0"
Paul Barton-Davis,Dylan McNamee,Raj Vaswani and Edward D. Lazowska
Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195
Technical Report 92-08-03, Revised October 1992.

[BLAC91] “Locking and Reference Counting in the Mach Kernel”
David L. Black, Avadis Tevanian Jr., David B. Golub and Michael W. Young
1991 International Conference on Parallel Processing, Austin, August 1991.

[CROV91] “Multiprogramming on Multiprocessors”
Mark Crovella, Prakash Das, Czarek Dubnicki, Thomas LeBlanc and Evangelos Markatos
Technical Report 385, The University of Rochester, Computer Science Dep. Rochester, New York, February 1991 (revised May 1991).

[MARK92] “Memory-Conscious Scheduling in Shared-Memory Multiprocessors”
Evangelos P. Markatos and Thomas J. Leblanc
Technical Report, Computer Science Department, Univ. of Rochester, May 1992.

[MARS91] “First-Class User-Level Threads”
Brian D. Marsh et al.
ACM OSR, Vol.25 Num.5, October 1991.