Parte I

Estado del arte de los sistemas operativos para entornos multiprocesadores

1

Arquitecturas y Sistemas Operativos para Multiprocesadores con Memoria Compartida

"A los pocos instantes, con las primeras ráfagas de viento, el orden se había desbaratado y él volvía a empezar alineando a su manera los frágiles residuos, broza de Saint-Julien, como si de la regularidad de su colocación dependiese la armonía del mundo, su propia paz del alma."

Carlos Pujol ("El lugar del aire")

ABSTRACT: *Los sistemas operativos para multiprocesadores, por las características de su hardware y los requisitos que imponen las aplicaciones paralelas que en ellos corren, tienen unas necesidades propias a la hora de planificar flujos para ser ejecutados.*

En este capítulo, damos una visión global del estado actual en la gestión de flujos y de procesadores en sistemas multiprocesadores; analizamos los elementos que pueden afectar de manera positiva o negativa a dicha planificación, partiendo de características esenciales e inherentes a estos entornos cuando hay multiprogramación.

Empezamos con una descripción general de las arquitecturas multiprocesador que nos encuadran en las que han sido objeto de nuestro trabajo.

Capítulo 1: Arquitecturas y Sistemas Operativos para Multiprocesadores con Memoria Compartida.

1.1. Introducción 16
1.2. Multiprocesadores con memoria compartida 17
1.2.1. ¿Qué es un multiprocesador? 17
1.3. Sistemas Operativos en multiprocesadores 19
1.3.1. Transporte de SO monoprocesador: UNIX 19
1.3.2. Diseño de sistemas operativos para multiprocesadores 20
1.3.3. Tecnología actual: Microkernels 20
1.4. Soporte de los sistemas operativos a la multiprogramación 21
1.4.1. Objetos planificables por el sistema 21
1.4.2. Estructuras para gestionar la planificación por parte del sistema 22
1.5. Políticas de planificación de flujos para sistemas multiprocesadores 23
1.5.1. Políticas de tiempo compartido 23
1.5.2. Planificación de grupos 24
1.5.3. Políticas de espacio compartido 25
1.6. Factores que influyen en la planificación de los multiprocesadores multiprogramados de propósito general 26
1.6.1. Mecanismos de optimización en la sincronización 27
1.6.2. Afinidad al procesador 28
1.7. Referencias y Bibliografía 29

1.1. Introducción

El procesamiento en paralelo se ha convertido en la propuesta más inmediata para incrementar la potencia de cálculo de los computadores actuales. Las aplicaciones científicas y de ingeniería de gran escala, como pueden ser la predicción del tiempo o la modelización de materiales y procesos entre otras, requieren velocidades de cálculo de GigaFlops y memorias de acceso rápido en cantidades de Terabytes. Sin embargo, es posible que, más que esta clase de aplicaciones de alto rendimiento para el cálculo (definidas en el proyecto HPCC[1]), sean importantes las propias motivaciones comerciales para el desarrollo de máquinas paralelas, incluyendo entre ellas la mejora de la relación coste/rendimiento de las máquinas, la escalabilidad de los diferentes requisitos de las aplicaciones y aumentar la fiabilidad y disponibilidad de los sistemas [PASA92].

Vamos a dar en este capítulo una rápida visión del hardware multiprocesador de las últimas décadas y de los principales conceptos que afectan a los sistemas operativos para máquinas paralelas. Éstos son los que les han dirigido hacia los resultados actuales que encontramos hoy en el mercado o en los centros de investigación.

Un sistema operativo multiprocesador debe incluir la mayoría de las funcionalidades básicas que hay presentes en un sistema monoprocesador. Aparecen, sin embargo, en su planteamiento y realización, mayores complejidades, debidas a la capacidad que se añade a cada funcionalidad en el hardware multiprocesador y, sobre todo, a los requisitos de rendimiento que se imponen al sistema operativo para dichos sistemas.

Problemas específicos a tratar directamente relacionados con el rendimiento son: la protección en espacios de direcciones extensos, la prevención de los abrazos mortales (*deadlocks*), el manejo de excepciones en programas paralelos de gran escala, el conseguir una representación eficiente de entidades activas asíncronas -procesos pesados o ligeros-, el proporcionar esquemas alternativos de comunicación y mecanismos de sincronización y, finalmente, el resolver la gestión de la planificación, asignando procesos a procesadores o distribuyendo de manera adecuada los datos en la memoria física del sistema.

Nuestro estudio va dirigido básicamente a sistemas multiprocesadores de memoria compartida, aunque quedará plasmada la convergencia que existe actualmente, tanto en los problemas como en las soluciones, hacia los sistemas multicomputadores. Esto es debido a los desarrollos tecnológicos actuales: los multiprocesadores, cuando se escalan a centenares de procesadores, se comportan más bien como sistemas distribuidos por las distancias y velocidades de acceso a la información.

Nos hemos centrado más en el rendimiento que en la fiabilidad de los sistemas; y más en el kernel de los sistemas operativos, que en el sistema operativo en global. El resto de elementos del sistema, de más alto nivel, como pueda ser el sistema de ficheros, interfaces de usuario, o soporte a la red o bases de datos, no son triviales ni menos importantes para el diseño y rendimiento efectivo del sistema, pero su estudio y discusión quedan al margen de este trabajo, y en parte sus mejoras quedan supeditadas a las mejoras del kernel subyaciente.

Por ser la materia que nos ocupa, trataremos en este capítulo de las diferentes políticas de planificación que encontramos hoy día en sistemas multiprocesadores, ya sean comerciales o de desarrollo. De las múltiples posibilidades que se presentan para nuestro trabajo, nos centraremos en una gestión dinámica de los procesos y procesadores; es decir, que se retoman decisiones durante la vida de una aplicación, no únicamente en su momento de inicio. Y las arquitecturas sobre las que se llevan a cabo serán de propósito general, con memoria compartida, sin considerar por ahora de una manera directa la situación más o menos lejana entre los procesadores y la memoria (NUMA) o el mecanismo de acceso a datos no propios (NORMA). Y sólo en cuanto que

1. High Performance Computing and Communications, USA

puedan afectar a estas políticas, veremos también algo sobre la localización de datos en memoria y mecanismos de sincronización y comunicación [MUKH92].

Es importante notar que, aunque la mayoría de las políticas de planificación de procesos que se llevan a cabo en un monoprocesador, pueden adaptarse teóricamente a un multiprocesador, no todas son convenientes, ni llevan a un resultado satisfactorio de funcionamiento del sistema. Ni siquiera existe una realización práctica de todas ellas en el mundo de los monoprocesadores propiamente. Y aún después de haber seleccionado un grupo reducido de entre las diversas políticas propuestas en los sistemas monoprocesador para multiprogramación, desde un amplio abanico de arquitecturas específicas y con diferentes modelos de programación, hay poco trabajo hecho para ver cómo afectan a las aplicaciones que pueden correr sobre ellas, con variaciones en la granularidad del paralelismo y la sincronización utilizada. Esto confluye en verdaderos problemas de rendimiento en sistemas que cuentan, sin embargo, con una gestión compleja y potente, desde el punto de vista del sistema operativo.

De todo ello vamos a hablar en los siguientes puntos de este capítulo.

1.2. Multiprocesadores con memoria compartida

1.2.1. ¿Qué es un multiprocesador?

A partir de la definición más sencilla -múltiples procesadores en una misma máquina-, empezaremos con una clasificación tradicional de los sistemas paralelos, para ir determinando el sistema que ha sido objeto de nuestro estudio: multiprocesadores de memoria compartida. Existen varios escritos sobre el tema, tanto en artículos como en libros de texto [TABA90], según la clasificación de Flynn, y omitimos de nuestra clasificación las arquitecturas SIMD, las orientadas a lenguajes aplicativos y las de aplicaciones sistólicas, no por restarles importancia sino porque nos alejan de nuestro camino. Tampoco entraremos en aquellas orientadas a aumentar el *throughput*, como puede ser las que trabajan con coprocesadores aritméticos, controladores inteligentes o procesadores vectoriales.

Se han hecho diferentes clasificaciones de sistemas multiprocesadores basándose en las características de sus diferentes componentes y la manera de acceder a ellos:

- Homogéneos versus heterogéneos, según esté compuesto por procesadores todos iguales o diferentes. En un sistema homogéneo, todos los procesadores son equipotentes e intercambiables.
- Fuertemente acoplados versus débilmente acoplados. En el primer caso, cada procesador tiene memoria privada y recursos de entrada/salida propios, y puede comunicar con otros procesadores por intercambio de mensajes. En los procesadores fuertemente acoplados, cada procesador tiene acceso directo a toda la memoria y a los dispositivos de entrada/salida. El reloj también es común.
- Interconexión por bus versus interconexión por red, según el tipo de interconexión para la comunicación con memoria: los sistema basados en el bus y los basados en la conmutación, también conocidos como sistemas multietapa. El sistema de bus común, o bus de tiempo compartido, es el sistema de interconexión más sencillo, aunque tiene un grave problema de degradación en el rendimiento, debido a la contención del bus y la memoria. Una solución a este problema es la incorporación de una memoria cache entre el procesador y la memoria principal que reduzca el tráfico del bus y los conflictos.
- Memoria compartida versus paso de mensajes, según el modelo hardware subyacente. Tanto programas que trabajen con memoria compartida, como diseñados

con mensajes, pueden ser ejecutados en un hardware de memoria compartida o de paso de mensajes. El problema es cuán eficiente puede ser una implementación en cada caso y cual será el rendimiento obtenido [ALMA89].

- *Master/slave* versus simétricos, según las funciones que lleguen a un determinado procesador (por ejemplo, repartición de las interrupciones,...). Aunque el software de sistema considere por igual a todos los procesadores en cuanto a recurso que ofrece a las aplicaciones, es normal encontrar sistemas con uno de sus procesadores dedicado a la atención de interrupciones [CORO90].

Desde el punto de vista de acceso a los datos, nos encontramos con:

- Multiprocesadores de memoria compartida, donde cada módulo de memoria es accesible desde todo procesador y compartido por todo ellos. De acuerdo al coste del acceso a la memoria, pueden clasificarse a su vez en:

- Multiprocesadores con Acceso Uniforme a Memoria (UMA), en los que el tiempo de acceso a cualquier módulo de memoria es el mismo para todos los procesadores del sistema. Un ejemplo son las arquitecturas basadas en un bus común [ALMA89].

- Multiprocesadores con Acceso no Uniforme a Memoria (NUMA), en los que el tiempo de acceso puede variar dependiendo del módulo de memoria y del procesador que accede a ella. Generalmente, han entrado en esta categoría los multiprocesadores con memoria local asociada a cada procesador: el acceso a la memoria local era más rápido que el acceso a cualquier otro módulo. Ejemplos de estas arquitecturas son el BBN Butterfly y el actual supercomputador Kendall Square Research (KSR) [KEND93].

- Multiprocesadores de memoria privada, o multiprocesadores sin acceso a memoria remota (NORMA), en los que cada procesador tiene su propia memoria local no accesible desde los otros procesadores del sistema. Significa que todo acceso a un dato de ese módulo ha de pasar obligatoriamente por una petición a dicho procesador. Entran aquí los hipercubos como los Ncubos y las actuales máquinas iSC mesh, las CM-5 de Thinking Machines y las redes de *transputers*. Se conocen también como multicomputadores, diferentes de los multiprocesadores propiamente dichos, por el modo de acceder a la memoria, y de los sistemas distribuidos, o *clusters* de estaciones de trabajo por la comunicación entre máquinas, llevada a cabo por hardware especializado, en el primer caso y que permiten algoritmos diferentes y más eficientes para la planificación, sincronización y envío de mensajes.

Las arquitecturas UMA son las más comunes para multiprocesadores en la actualidad, debido en parte a que la mayoría de máquinas se emplea únicamente para aumentar el *throughput* de la multiprogramación, en sistemas multiusuario de tiempo compartido, más que para la ejecución de programas paralelos de mayor escala.

Es interesante observar, sin embargo, que la tecnología actual está sesgando cada vez más las máquinas hacia el tipo de acceso NUMA.

En parte, debido a la distancia a la que se ha llegado entre la velocidad de los componentes de cálculo, por un lado, y la velocidad de acceso a memoria y de los componentes de comunicación, por otro. También por el incremento en el uso de memorias locales y caches para reducir la

contención en el acceso al bus -o buses- compartidos.

Debido a que las arquitecturas NUMA permiten un orden más elevado en el número de procesadores del sistema -de 128 a 512- y una mayor escalabilidad, muchos sistemas experimentales son máquinas NUMA y su número y popularidad se va incrementando en la actualidad.

Las máquinas NORMA, por su parte, son de un acoplamiento más débil que las NUMA, ya que no tienen soporte hardware para acceder de manera directa a módulos remotos de memoria. Sin embargo, avances recientes en la tecnología de los supercomputadores van acortando la diferencia de relación entre los accesos locales y los remotos en arquitecturas NORMA (1:500 de un acceso local frente a uno remoto) que las aproximan a los tiempos de acceso de máquinas NUMA, como la KSR (1:100). Se intuye, pues, que en el futuro tanto las máquinas NUMA como las NORMA van a requerir de un soporte similar por parte del sistema operativo y de unas similares herramientas en el soporte de la programación para lograr un rendimiento eficiente.

En el resto del capítulo vamos a centrarnos más en el primer aspecto del sistema operativo: como gestor de recursos que proporciona la posibilidad de paralelismo a los usuarios.

A partir de los primeros sistemas operativos multiprocesadores, pensados como una extensión de los monoprocesadores, surgen los SO propiamente para arquitecturas multiprocesadores. Finalmente, la tecnología actual son los microkernels [BRIC91].

1.3. Sistemas Operativos en multiprocesadores

El rendimiento que pueda obtenerse en cualquier sistema informático es resultado de la combinación del hardware y del software; y como elemento radical del software, el sistema operativo. Con la aparición de máquinas de propósito general de más de un procesador, surge la necesidad de adaptar los sistemas operativos ya existentes para la gestión de un nuevo recurso con múltiples instancias: el procesador.

Al hablar de sistemas operativos en entornos paralelos, hemos de distinguir cuidadosamente dos aspectos del trabajo paralelo. Por un lado, el sistema operativo como gestor de los recursos que utilizan los usuarios, ha de ser capaz de dar soporte a trabajos paralelos; por ejemplo, que un único programa pueda utilizar varios procesadores. Por otro lado, el propio sistema operativo está corriendo en un sistema multiprocesador y, por tanto, él mismo es un programa paralelo .

Las mismas rutinas del sistema operativo han de adaptarse al nuevo funcionamiento del entorno, ahora paralelo, y con problemas de concurrencia y exclusión mutua muchas veces encubiertos. La mayoría de las realizaciones de sistemas multiprocesadores suelen incluir el diseño de mecanismos para la sincronización y comunicación entre procesadores. Para una mayor eficiencia, se intenta incluso que estos mecanismos formen parte del hardware de la máquina, aumentando o modificando, si es posible, el conjunto de instrucciones del procesador.

Tres configuraciones básicas han sido adoptadas a la hora de clasificar un sistema operativo, según su comportamiento en presencia de múltiples procesadores: supervisores separados, *master/slave* y simétricos [ALMA89].

En cuanto al sistema operativo, como programa paralelo, se puede decir que su evolución ha pasado por diferentes etapas, como vamos a ver a continuación.

1.3.1. Transporte de SO monoprocesador: UNIX

Por sus características y su popularidad en todo el mundo informático, el sistema operativo más transportado a arquitecturas multiprocesador ha sido UNIX [NAVA91]. Pero el planteamiento de base sobre el que está construido pone un límite a la explotación del paralelismo.

Por una parte, su construcción monolítica hace costoso el trabajo de paralelizar todas las partes del sistema operativo. Choca, además, frontalmente con la exclusión mutua implícita original[2] que le hace trabajar en el código de kernel comportándose como un monitor[3] y despreocupándose, muchas veces, de garantizar la ejecución atómica de ciertas partes de código.

Por otro lado, su diseño ya parte de limitaciones importantes en la granularidad de recursos que puede ofrecer a las aplicaciones: a partir del proceso, ni puede dar soporte a que el código de usuario trabaje con más de un procesador, ni puede tampoco permitirlo a nivel de servicio del propio sistema.

1.3.2. Diseño de sistemas operativos para multiprocesadores

En las dos últimas décadas se construyen nuevos sistemas operativos para máquinas multiprocesador, diseñados desde cero. Se rompe la atomicidad en la ejecución de código de sistema, hasta donde lo permita la semántica de las operaciones. Se separan los mecanismos -soportados por el kernel- de las políticas -gestionadas a niveles superiores-, potenciando más la versatilidad de las arquitecturas (Hydra [WULF74]).

La solución definitiva es plasmar la sencillez de UNIX en las operaciones de sistema a través de una realización multi-hebra. Los servicios del sistema operativo ya no se realizan como secuencias de llamadas a rutinas, sino que pasan a ser flujos con una identidad propia que pueden moverse en paralelo por el código de sistema, respetando pequeñas partes que se acceden en exclusión mutua (Sprite [OUST88]).

Hace falta, sin embargo, subir el paralelismo al nivel de usuario. Se hace necesaria la aparición de nuevos objetos, de nuevas abstracciones que permitan dar paralelismo también a los programas, limitados todavía a un único fluir secuencial: el proceso.

1.3.3. Tecnología actual: Microkernels

Los sistema basados en microkernels se estructuran como una colección de servidores (Mach [ACCE86], V kernel [CHER84]), o subsistemas (Chorus [CHOR91]), corriendo encima de un kernel mínimo: el microkernel, que realiza sólo las funciones de nivel más bajo del sistema.

Entre sus funciones primitivas se incluyen la gestión de procesos, la comunicación y sincronización entre procesos, la gestión de memoria de bajo nivel y una mínima gestión (omitida totalmente en alguno de ellos) de la entrada/salida (E/S). El resto de los servicios de un sistema operativo tradicional se encuentran en espacio de usuario, aunque generalmente, trabajan con ciertos privilegios respecto al resto de aplicaciones.

Las aplicaciones han de utilizar mecanismos de mensajes entre espacios de direcciones disjuntos, habitualmente enmascarados en llamadas a procedimiento remoto (RPCs), para obtener servicios del sistema. Esto convierte en elemento crítico al mecanismo de comunicación entre procesos (IPC) cuando se evalúa el rendimiento global del sistema.

Se ofrece a las aplicaciones abstracciones más finas para explotar la concurrencia: se pasa de una máquina virtual monoprocesador -el proceso-, a una máquina virtual multiprocesador, diferenciando el espacio de direcciones de los flujos de ejecución o procesadores virtuales. Aparecen así la task y los threads en sustitución del proceso UNIX: ya se pueden ejecutar programas con paralelismo real.

Su desarrollo ha ido evolucionando con su uso y por sus usuarios llegándose a una defini-

2. Esta exclusión mutua sobre todo el código del kernel ha fundamentado una programación despreocupada de las condiciones de carrera. Por ejemplo, se tardaron tres años en depurar la versión multiprocesador SVR3 basada en las primitivas de semáforos; siempre fallaba un *lock* u otro.

3. Esta atomicidad de trabajo en el kernel en relación a un flujo, la han heredado sin embargo sistemas de investigación de los más actuales, como veremos en capítulos posteriores (i.e. Mach).

ción más o menos madura de las abstracciones y modelos que defienden, pero a unos códigos intratables, enormes y caóticos. A partir de las abstracciones y filosofía de diseño, se necesita ahora una nueva reescritura más pura, más sencilla [BERN93].

1.4. Soporte de los sistemas operativos a la multiprogramación

El fin que se persigue en la multiprogramación, ya sea en sistemas monoprocesadores como en multiprocesadores, es aumentar la utilización de los recursos del sistema. Esto se logra a base de compartir un número concreto de procesadores (uno o varios) entre un número generalmente mayor (incluso de diferente orden de magnitud) de trabajos.

Los sistemas multiprogramados son sistemas complejos, con cierta sofisticación. Tienen un número variado de procesos preparados para ejecutar, que hay que mantener simultáneamente en memoria, o por lo menos información sobre ellos. Son procesos, por otra parte que se van alternando en el uso de todos los recursos del sistema, no sólo la CPU, sino también la memoria física, los dispositivos e incluso recursos lógicos, como puede ser la posesión de estructuras de sincronización (semáforos, esperas activas,...).

El tener trabajando sobre los mismos recursos del sistema a diferentes procesos, de diferentes aplicaciones, con diferentes necesidades y características, es un factor decisivo a la hora de elegir políticas de asignación de procesos a procesadores. La respuesta a las preguntas qué proceso ha de ser planificado en un momento dado y dónde; o visto de un modo más general, qué procesos pueden competir por qué conjunto de recursos y cómo, puede cambiar de manera drástica el rendimiento de un sistema.

1.4.1. Objetos planificables por el sistema

Dentro de las funciones de un sistema operativo está la de crear y gestionar -planificar- entidades activas de ejecución. Lo que clásica y tradicionalmente se ha conocido como gestión de procesos, o flujos. Cada vez que se crea una de estas entidades -procesos-, se crea para ella una máquina virtual [NUTT92]. Los rasgos de esta máquina virtual los determina el interfaz de llamadas al sistema; es decir, el conjunto de llamadas al sistema disponibles para que la aplicación interaccione con el sistema operativo.

La efectividad y buen rendimiento de la marcha del sistema depende en gran parte del funcionamiento de estas primitivas que el sistema ofrezca a la aplicación para expresar su paralelismo. Conforme el coste de crear y gestionar dicho paralelismo va disminuyendo, se hace más eficiente el definir trabajos de granularidad cada vez más fina [MARK93].

Esta capacidad del sistema de ofrecer a la aplicación entidades en las que ejecutar flujos, puede verse como la capacidad del sistema de ofrecer una máquina virtual única y exclusiva para cada aplicación. Este entorno -que lo aísla y protege a la vez de otros procesos- se compone del conjunto de recursos que el sistema le asigna: procesador, memoria principal y secundaria (donde residen el código ejecutable, los datos y la pila), acceso a los dispositivos de entrada/salida, espacio de almacenamiento en disco u otros dispositivos de soporte, herramientas de sincronización y comunicación con otros procesos; además de la información necesaria para poder identificarlo y gestionarlo: identificación del usuario, identificación del proceso, estado en el que se encuentra, prioridades, tiempo de ejecución que lleva, por citar algunos[4].

Una manera de expresar el paralelismo es utilizando procesos al estilo Unix, en los que no se distingue el flujo propiamente del espacio de direcciones asociado a él; están integrados los

4. Estos datos pueden ser más o menos exhaustivos dependiendo del sistema. Los que damos aquí están prácticamente en cualquier sistema de uso general que trabaje en tiempo compartido.

conceptos de máquina virtual y procesador virtual. Resultado de ello es que se les conozca como procesos pesados y que, como elementos de paralelismo, representen una eficiencia muy pobre, debido a dos razones principalmente. Primero, porque en el coste de crear y destruir entidades, siempre va unido al coste del flujo el de su espacio de direcciones encareciéndolo notablemente. El segundo motivo es en cuanto a la replanificación del procesador; un cambio de contexto supone siempre en este tipo de objetos un cambio de espacio de direcciones con la consiguiente actualización de los registros de mapeo de memoria virtual y, a largo plazo, actualización de la cache y del TLB.

La mayoría de los sistemas actuales plantean las entidades de planificación como una máquina virtual multiprocesador, en la que múltiples procesadores -virtuales- comparten memoria -virtual- y el resto de recursos -virtuales- de la máquina.

Se diferencia entonces el entorno de ejecución -proceso o task-, que sería la propia máquina, y el contexto local de cada procesador -thread de kernel-, que sería la unidad planificable por el procesador. Las ventajas que presentan frente al caso anterior son claras; baste citar entre ellas el hacer físicamente posible el paralelismo entre flujos de una misma aplicación.

Hablaremos con más detalle en capítulos posteriores de los procesadores virtuales, ya que el trabajar con un tipo de objeto u otro (pesado o ligero, proceso o thread), afecta al tipo de paralelismo soportado por el sistema más que a la política de planificación empleada. A partir de ahora, hablaremos de un objeto planificable, en general, como de un procesador virtual que el sistema ofrece al usuario para planificar los flujos de sus aplicaciones.

Las políticas de planificación del kernel versarán sobre los diferentes modos de mapear los procesadores virtuales que ofrece el sistema operativo a la aplicación en los procesadores físicos que ofrece la máquina.

1.4.2. Estructuras para gestionar la planificación por parte del sistema

Como el resto de servicios que ofrecen los sistemas operativos para máquinas paralelas, la parte de planificación -ya sea un módulo, un servidor, un subsistema, o todo ello junto-, ha de estructurarse para ser escalable a diferentes tamaños de máquina y distintos requerimientos de aplicaciones.

Al tener múltiples procesadores para poder mapear múltiples procesadores virtuales, aparece la posibilidad de ofrecer la planificación como un servicio centralizado del kernel, o distribuido entre los diferentes procesadores, desde un punto de vista estructural; es decir, en cuanto a estructuras de datos consultadas para la planificación. Y no desde el punto de vista de la gestión funcional en la que aparecería una comunicación entre procesadores para repartirse el trabajo, en el primer caso, o una política de auto-planificacción (*self-scheduling*) por procesador, en el segundo.

Una planificación estructuralmente centralizada abastece a todos los procesadores del sistema con una única cola de procesos preparados. Si la planificación sigue un modelo estructural distribuido, cada procesador tiene su propia cola de procesos. Se evita de esta manera la posible contención en la búsqueda de trabajo, corriendo a la vez el peligro de desbalancear la carga de tra-

bajo de los diferentes procesadores. Existen distintas políticas para evitar estos problemas, como introducir un proceso en la cola que actualmente se halle más vacía (puede ser una sobrecarga considerable, dependiendo del número de procesadores), o ir a buscar procesos a otras colas en caso de tener la propia vacía.

Existen sistemas que tienen también una cola con los procesadores que actualmente no tienen trabajo, de modo que se consulta en primer lugar para asignarle directamente el trabajo. Se evita de este modo la contención y la sobrecarga de introducir un proceso en una cola de preparados para extraerlo acto seguido [BLAC90].

Es importante también en los sistemas operativos multiprocesadores, para extraer la máxima eficiencia en la gestión de procesos, evaluar el coste de la planificación de los procesos nulos.

Cada procesador físico tiene su propio proceso nulo, que pone en ejecución cuando no tiene posibilidad de elegir a ningún otro proceso en el sistema. En un sistema basado en prioridades, los procesos nulos son los procesos de menor prioridad y por ello, podrían colocarse en la misma cola de preparados que el resto de procesos. Si cada procesador tiene su propia cola de procesos, el nulo será el último y sólo pasará a ejecutarse en caso de no haber ningún otro. En estos casos, se suele tener una cola diferenciada de procesos nulos a la que los procesadores acceden directamente, cada uno a su proceso nulo [NAVA91], o incluso, tener cada procesador un puntero directo a su proceso nulo para planificarlo cuando la cola de procesos está vacía [BLAC90].

1.5. Políticas de planificación de flujos para sistemas multiprocesadores

Las políticas de planificación para sistemas multiprocesadores multiprogramados que funcionan hoy día, o que se han propuesto, son muy variadas. Podemos agruparlas todas ellas en dos grandes familias: las políticas de tiempo compartido y las de espacio compartido [CROV91].

Las políticas de planificación de tiempo compartido, reparten el recurso procesador entre las aplicaciones en función del tiempo total del sistema, asignándoles ranuras -*slots*- de tiempo. Son adaptaciones de las políticas de planificación de sistemas monoprocesador de las que provienen en su fundamento [LEUT90].

En sistemas multiprocesadores, pueden plantearse otro tipo de políticas repartiendo todo el sistema en función de su espacio; es decir, particionando el número total de procesadores entre las aplicaciones del sistema.

Como caso especial de planificación en multiprocesadores está la planificación de grupos: se considera toda la máquina como una única partición y se multiplexa en el tiempo todo el grupo de procesadores como bloque entre las diferentes aplicaciones.

Vamos a ver a continuación con más detalle cada uno de los tipos de políticas de planificación, y dentro de cada uno de ellos, las más conocidas y utilizadas,.

1.5.1. Políticas de tiempo compartido

Para decidir qué proceso seleccionar para ser ejecutado en un determinado procesador, pueden utilizarse diferentes políticas, habitualmente según una prioridad asignada a cada proceso. Lo que define cada una de las diferentes políticas que se han propuesto son los parámetros bajo los que se

determina la prioridad de un proceso: el tiempo de CPU consumido en total desde el inicio de su ejecución, o consumido recientemente desde su última planificación, el tamaño del proceso (tiempo que se estime va a durar su ejecución), el número de flujos, o cualquier otro criterio previsto por el sistema, o por el usuario.

Los sistemas que se basan puramente en la prioridad que se asigna a un proceso, pueden tener problemas de inanición; para solucionar este problema, la mayoría de sistemas basados en prioridades las van cambiando en el tiempo mediante un mecanismo de envejecimiento (*aging*). Se conocen también como sistemas de prioridades dinámicas (UNIX, Mach y, en la práctica, todos los sistemas actuales que trabajan con políticas de tiempo compartido).

Existen, por último, políticas de planificación para procesos con grupos de flujos. Desde el punto de vista del usuario, lo importante no es el rendimiento de un flujo en particular, sino el del conjunto de trabajos que conforman la aplicación.

1.5.1.1. *Round-Robin* (RR)

La política más sencilla, conocida como de “tiempo compartido” por excelencia, es la de *round-robin*: se asigna a cada proceso un tiempo concreto de procesador, arbitrado por el reloj del sistema, y al finalizar este tiempo se cede la CPU al siguiente proceso de la cola de preparados.

El factor más importante es determinar el valor apropiado para intervalo de tiempo en que se ha de producir la expiración del tiempo asignado, o *quantum*: si el tiempo es muy pequeño, la frecuencia de cambios de contexto aumentará, si el tiempo es muy grande, estamos favoreciendo la desigualdad en el uso del procesador, en contra de los flujos que sean más interactivos. Lo ideal es que el *quantum* se ajuste lo máximo posible a las ráfagas naturales de los procesos que más E/S hacen en el sistema.

Existen dos versiones de política *round-robin* para multiprocesadores. La primera, sigue fielmente el esquema monoprocesador. La segunda, se orienta más a la semántica de aplicación paralela, formada por varios procesos, y utiliza trabajos, más que procesos, como unidades planificables. Se sustituye la cola compartida de procesos por una cola compartida de trabajos; cada entrada de esta cola, contiene una cola de procesos: los procesos que forman un job [LEUT90].

1.5.2. Planificación de grupos

Desde el punto de vista de la sincronización, lo ideal sería que las esperas se redujeran a lo que los programadores de cualquier aplicación paralela estiman. Para ello, tendrían que estar corriendo a la vez todos los procesos ejecutables de una misma aplicación. Es lo que Ousterhout llama ***coscheduling*** [OUST81], también conocido como gang scheduling o task forces.

La replanificación del conjunto de procesos se hace sobre el mismo conjunto de procesadores, lo que puede ser bueno para mantener la coherencia de la cache. De hecho, cada proceso se replanifica siempre sobre el mismo procesador y así se puede puede obtener incluso un entorno de ejecución para el software que sea estático (asignar la memoria más cercana a dicho procesador, por ejemplo) y se mantenga durante toda la vida de la aplicación.

El problema principal del *gang scheduling* es su control centralizado, que puede conducir a un cuello de botella. Además, se llega con facilidad a la fragmentación de procesadores, con infrautilización del sistema. En simulaciones hechas [GUPT91], la utilización del sistema aumenta al aumentar el *time slice*, principalmente por la reducción de cambios de contexto. Hay en general pocos datos sobre esta política y los resultados pueden dar muy diferentes (opuestos) si se tiene en cuenta o no los problemas de mantener la coherencia de la cache: hay que tener en cuenta en estas políticas que, al estar corriendo a la vez todos los flujos de una aplicación, hay una posibilidad muy elevada de estar accediendo a los mismos datos y, por tanto, invalidándose las

memorias caches continuamente. Generalmente, son políticas que se utilizan cuando el modelo base de comunicación entre los flujos de una aplicación es el paso de mensajes, para reducir las esperas de sincronización en las comunicaciones.

Además del *coscheduling* de Medusa, otros sistemas permiten diferentes adaptaciones de planificación de grupos, incluso conviviendo con otras políticas de tiempo compartido más tradicionales: IRIX de Silicon Graphics y Symunix [EDLE88].

1.5.3. Políticas de espacio compartido

Al tener más de un procesador en el sistema, pueden plantearse nuevas maneras de repartir el recurso procesador entre todas las aplicaciones del sistema. Por ejemplo, particionar el conjunto de procesadores entre todas ellas y adaptar el número de procesadores virtuales de cada aplicación al de procesadores físicos. Equivaldría a hacer una **multiplexación en el espacio**, más que en el tiempo.

La diferencia básica entre los particionados y la planificación de grupos es que, mientras este último coge todos los procesadores para cada aplicación, el particionado proporciona a cada aplicación una máquina más pequeña, dedicada, de la que se intenta obtener la máxima utilización.

Este tipo de planificación, por su propia naturaleza, al tener una aplicación corriendo en un número pequeño de procesadores, puede dar (según la naturaleza de la aplicación, claro está) una mejora inherente en cuanto a los aciertos en la cache, por su afinidad al procesador.

La partición de procesadores entre aplicaciones puede ser estática o dinámica. En el primer caso, se divide el número de procesadores del sistema entre las aplicaciones que vayan a ejecutarse. Esta partición puede ser, a su vez, regular (a todas las aplicaciones la misma cantidad), o irregular, dependiendo por ejemplo, del número de procesos que tenga cada aplicación. En las particiones dinámicas, el número de procesos de una aplicación varía dependiendo del número de procesadores que haya disponibles en el sistema. Al tener el mismo número de procesos que de procesadores, se eliminan cambios de contexto y se consigue, a la vez, un buen comportamiento en cuanto a los aciertos en la memoria cache y al sincronismo.

Tener un óptimo rendimiento en este tipo de planificación, supone tener un módulo en el sistema operativo encargado de monitorizar la carga del sistema y ajustar el número de procesadores asignado a cada aplicación.

La adaptación del número de procesos al de procesadores se hace mediante primitivas del sistema operativo, tipo *suspend()* y *resume()*, por lo que es más fácil en tecnologías que ofrezcan procesos ligeros, tipo *threads* de kernel, que con procesos de UNIX, mucho más pesados.

Generalmente, se combinan estas políticas de particionado de la máquina, con políticas de tiempo compartido. Se consigue así tener aplicaciones con un número variable de procesadores virtuales que multiplexan en el tiempo entre un grupo de procesadores físicos, dedicados a la aplicación. Existen algoritmos de planificación basados en este comportamiento intra aplicación (Memory Conscious Scheduling, University of Rochester [MARK93]).

1.6. Factores que influyen en la planificación de los multiprocesadores multiprogramados de propósito general

A la hora de evaluar qué políticas, qué mecanismos, qué objetos, qué estructuras de datos, y en general qué planificación es la más adecuada en un entorno multiprogramado, se han de medir varios factores.

Es opinión generalizada que el rendimiento de una aplicación empeora considerablemente cuando el número de procesos excede considerablemente el número de procesadores [TUCK89]. Ello es debido a:

- Los cambios de contexto se producen con más frecuencia cuando el número de procesos seleccionable en el sistema aumenta.
- La posibilidad de desbancar procesos que poseen un *lock*, mientras los que se están ejecutando están en una espera activa improductiva es mayor. Por el hecho de haber un número considerable de procesos en el sistema, es posible que tarde un tiempo no despreciable en volver a conseguir un procesador y poder entonces liberar el *lock*.
- Cuando un procesador está compartido por un número elevado de procesos y, por tanto, diferentes espacios de direcciones, el número de fallos de la cache se convierte en un factor importante de degradación del rendimiento.

Es importante notar aquí que, aunque este último punto podría evitarse en el caso de trabajar con threads de kernel, no es habitual tener en cuenta el que dos threads pertenezcan a la misma task a la hora de planificarlos.

En general, los sistemas que trabajan con políticas de tiempo compartido, tienden a considerar a todos los flujos planificables como independientes entre sí, tanto en sistemas monoprocesadores como multiprocesadores.

Existen, no obstante, algunos mecanismos en las primitivas de sincronización, lo mismo que en la gestión de la memoria cache, para cortocircuitar el protocolo impuesto por las políticas y permitir alguna mejora en el rendimiento[5].

Es extremadamente complicado, por no decir imposible, encontrar una política de multiprogramación que maximice siempre la utilización del procesador, asegure un uso equitativo a todas las aplicaciones y tenga un tratamiento para todos los posibles factores de sobrecarga. Los costes asociados a una política de planificación particular van asociados a la arquitectura subyaciente (coste de los fallos de la cache, tiempos de acceso a la memoria y al resto de recursos,...), mientras que el rendimiento obtenido depende de las características de las aplicaciones (número de procesos por aplicación, tamaño del estado asociado a cada proceso, frecuencia y tipo de sincronización) [MARK93].

5. Cfr. Capítulo 2.

La conclusión general es que, en sistemas multiprocesadores multiprogramados, conforme va aumentando el número de procesadores físicos del sistema, la política específica de planificación elegida influye cada vez menos en un aumento de rendimiento [STAL92].

1.6.1. Mecanismos de optimización en la sincronización

A partir de cualquier política de planificación de tiempo compartido, puede obtenerse algún tipo de optimización si se tiene conocimiento del código que se está ejecutando, o se permite a la aplicación pasar algún tipo de información al sistema; de esta manera se puede retrasar el desbanque de un proceso, señalar a un determinado procesador virtual (PV) para ejecutar, etc, mejorando el rendimiento del sistema.

Por el hecho de estar involucrados en un mismo objetivo final, los diferentes procesos de una aplicación paralela, interaccionan a menudo entre ellos, bien porque necesiten datos comunes, porque necesiten avisarse de algún evento, o esperar determinado trabajo, etc. En un entorno de memoria compartida ésto supone con frecuencia el tener que acceder a zonas de exclusión mutua, en las que la ejecución entre los varios flujos se secuencializa. Cuando un proceso se encuentra con el cerrojo echado a la hora de correr un determinado flujo, existen dos políticas básicas a seguir.

La primera, consiste en dejar al proceso en una espera activa hasta que pueda obtener paso (*spin lock*). Hasta ahora, este tipo de primitivas (*test&set*, *fetch&add*,...) han sido muy corrientes, porque se supone que son esperas muy cortas en comparación con el coste de un cambio de contexto. En entornos multiprogramados, este supuesto deja de ser cierto, cuando podemos encontrarnos con poseedores de una exclusión mútua que no están en ejecución, y no pueden por tanto liberarla, porque han sido desbancados por otros procesos (a lo peor, justamente, por procesos que luchan por conseguir el *lock*).

Aparece, entonces, la segunda opción: bloquear a los procesos que no pueden acceder a la exclusión mutua y ceder el uso de la CPU a otro proceso. El problema principal aquí es el coste de bloquear y cambiar de contexto, que puede paliarse, a menudo de manera ventajosa, si se deja al proceso un tiempo antes de decidir bloquearse. Ésto se debe a que, un proceso que espera un *lock*, o lo coge rápidamente (porque lo encuentra libre o porque tiene que hacer una espera insignificante), o bien ha de esperar largo tiempo. Eligiendo un tiempo adecuado de espera activa, generalmente conseguirá pasar y si no lo hace, es que el coste del bloqueo será casi seguro menos largo que el que le resta para conseguir entrar.

En determinadas aplicaciones, en las que existe comunicación entre diferentes procesos, puede ser útil el poder decidir qué proceso concreto seleccionar para reducir el tiempo de espera por sincronización: es lo que se conoce como *handoff*.

Por ejemplo, un proceso que esté esperando entrar en una zona de exclusión mutua, puede decidir ceder su tiempo restante de *quantum* al proceso que está trabajando en ella -en caso de estar este proceso sin ejecutar-, para acortar el tiempo de bloqueo. No es una técnica que mejore de manera significativa el *throughput* total del sistema [GUPT91], pero sí la latencia de ida-vuelta

para los procesos síncronos -como es el caso de un esquema de comunicación cliente-servidor- en un modelo RPC [BLAC90]. Si sólo se da al servidor el tiempo que resta de la ranura de tiempo, el número de cambios de contexto del sistema aumenta, lo que puede redundar en global en una disminución del rendimiento del sistema.

Es importante que los PV estén el mínimo tiempo posible en posesión de las secciones críticas, puesto que son zonas de ejecución inherentemente secuencial. Una manera sencilla de conseguirlo, es marcar mediante un *flag* a los procesos que se encuentren en esta situación para que no sean desbancados hasta que finalicen. Symunix permite esta opción [EDLE88].

Por otra parte, un proceso que quiere entrar en una zona de exclusión mutua y se encuentra en una espera activa (*busy waiting*) es un proceso que no está realizando ningún tipo de trabajo útil, retrasando incluso, en el peor de los casos, la liberación del *lock* que está esperando. Existen políticas de planificación de procesos en las que se marca a los procesos que están esperando un *lock*, de manera que sean los últimos a elegir para pasar a ejecutar. Generalmente, esta técnica se combina con la anterior [ZAHO88].

1.6.2. Afinidad al procesador

En entornos multiprogramados, en los que el número de procesos es bastante superior, a veces, al número de procesadores, una utilización eficiente de la memoria cache puede ser un factor decisivo en el rendimiento del sistema.

A la hora de asignar un proceso a un procesador, puede ser determinante el que, siendo el procesador en el que había corrido anteriormente dicho proceso (**afinidad al procesador**), se mantengan todavía en la cache los datos, o la mayoría de los datos, necesarios para empezar a arrancar el proceso (***footprint***)[6].

Existen diferentes tipos de *scheduling* basados en la afinidad de un proceso a un procesador. La política más sencilla es la de ***scheduling* fijo**, consistente en ligar un proceso a un procesador.

Otra política utilizada es la de **mínima intervención**, que consiste en seleccionar, dentro de los procesos ejecutables, aquel que más recientemente se ha ejecutado en dicho procesador, con lo cual la probabilidad de que todavía queden datos suyos en la cache, en principio es más alta que para cualquiera del resto de procesos seleccionables.

La decisión de elegir políticas que tengan en cuenta este hecho tendrá mayor o menor repercusión según el tipo de aplicación (que tenga un *footprint* más o menos grande o que se conserven sus datos de una a otra ejecución, por ejemplo): aunque es un factor que puede influir decisivamente en el rendimiento del sistema, pueden darse aplicaciones en las que de una ranura de tiempo a la siguiente se haya cambiado todo el contenido de la cache, ante lo cual, poco puede hacerse. En general, para cargas reales, se ha comprobado que los beneficios de la afinidad son pequeños [GUPT91] cuando los procesadores son compartidos entre varias aplicaciones.

6. Se define como footprint la "huella" que queda de un proceso en memoria cuando vuelve a ser planificado después de un cambio de contexto. Depende, no sólo de la propia aplicación, sino también del resto de procesos con los que esté compartiendo la memoria.

1.7. Referencias y Bibliografía

[ACCE86] “Mach: A New Kernel Foundation for UNIX Development"
Mike Accetta et al.
Proceedings of the Summer 1986 Usenix Conference, July 1986.

[ALMA89] "Highly Parallel Computing"
George S. Almasi and Allan J. Gottlieb
The Benjamin/Cummings Publishing Company, Inc., 1989.

[BERN93] Philippe Bernadat, OSF Research Institute, Grenoble. Comunicación personal.

[BLAC90] "Scheduling and Resource Management Techniques for Multiprocessors"
David L. Black, Ph D.
CMU-CS-90-152, Carnegie Mellon University, Pittsburgh, July 1990.

[BRIC91] "A new Look at Microkernel-Based UNIX Operating Systems: Lessons in Performance and Compatibility"
Allan Bricker, Michel Gien, Marc Guillemont, Jim Lipkis, Douglas Orr and Marc Rozier
Technical Report CS/TR-91-7, Chorus systèmes, February 1991.

[CHER84] “The V Kernel: A Software Base for Distributed Systems”
David Cheriton
IEEE Software, April 1984.

[CHOR91] “CHORUS/MIX V.3.2 r3
Chorus systèmes, 1991.

[CORO90] " 486/smp Symmetric, Compatible, Multiprocessor System"
Corollary Inc., Irvine, CA 92714, 1990.

[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).

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

[GUPT91] "The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications"
Anoop Gupta, Andrew Tucker and Shigeru Urushibara
ACM, 1991.

[KEND93] "KSR Parallel Programming"
Kendall Square Research Corporation, 7/1/93.

[LEUT90] "Issues in Multiprogrammed Multiprocessor Scheduling"

Scott Thomas Leutenegger
Ph.D. Thesis
University of Wisconsin, August 1990.

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

[MUKH92] "A Survey of Multiprocessors Operating System Kernels"
Bodhisattwa Mukherjee, Karsten Schwan and Prabha Gopinath
Technical Report GIT-CC-92/05, Georgia Institute of Technology, Atlanta, November 1993.

[NAVA91] "Paralelismo de Procesos e Interrupciones en el Transporte de UNIX a un Multiprocesador"
Nacho Navarro
Tesis Doctoral, UPC, Octubre 1991.

[NUTT92] "Centralized and Distributed Operating Systems"
Gary J. Nutt
Prentice Hall, 1992.

[OUST81] "Medusa: A Distributed Operating System"
John K. Ousterhout
UMI Research Press, 1981.

[OUST88] "The Sprite Network Operating System"
John K. Ousterhout et al.
COMPUTER, February 1988.

[PASA92] "System Software and Tools for High Performance Computing Environments"
Report on the Findings of tha Pasadena Workshop, April 14-16, 1992.

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

[TABA90] "Multiprocessors"
Daniel Tabak
Prentice-Hall, 1990.

[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.

[WULF74] "Hydra: The Kernel of a Multiprocessor Operating System"
W. Wulf, E. Cohen et al.
Communications of the ACM, Vol. 17, Num.6, June 1974.

[ZAHO88] “Spinning versus Blocking in Parallel Systems with Uncertainty”
John Zahorjan et al.
Proceedings of the International Symposium on Performance of Distributed and Parallel Systems, December 1988

[ZAHO90] "Processor Scheduling in Shared Memory Multiprocessors"
John Zahorjan and Cathy McCann
ACM Proceedings of SIGMETRICS'90, 1990.

2

Concurrencia y Paralelismo en Aplicaciones

“Para dos no hay pendiente demasiado empinada”

Ibsen

ABSTRACT: Después de habernos situado en el estado actual de las arquitecturas y software de sistema, en este capítulo vamos a ascender a nivel de usuario para ver el estado actual de la programación paralela. Partiremos de los paradigmas que dan soporte a que la aplicación se exprese concurrentemente basándonos sobre todo en los mecanismos que facilitan la comunicación, sincronización y puesta en marcha de diferentes trabajos en una misma aplicación.

Diferentes ayudas actuales para que la aplicación explote su paralelismo, dependiendo de su fisionomía; bien automáticamente, por compilación, o bien explícitamente a partir de librerías o del mismo lenguaje de programación empleado.

El sistema operativo es una pieza clave para alcanzar el rendimiento óptimo en un entorno paralelo multiprogramado. El es el que, tradicionalmente, ha venido ofreciendo al usuario la posibilidad de expresar concurrencia en sus aplicaciones y el que, de manera directa o implícita, afecta mediante el comportamiento de sus primitivas a la planificación a corto, medio y aún largo plazo, de sus flujos. En la segunda parte de este capítulo damos una visión del sistema operativo como herramienta de paralelismo, siendo nuestro objetivo el encontrar estrategias de planificación de flujos apropiadas a las aplicaciones paralelas.

Capítulo 2: Concurrencia y Paralelismo en Aplicaciones

2.1. Introducción 36
2.2. Paradigmas de programación concurrente 37
2.2.1. Compiladores y arquitecturas especializadas 38
2.2.2. Estructuras del lenguaje 39
2.2.3. Librerías 40
2.3. Programación con procesos ligeros 40
2.3.1. Mecanismos de creación 41
2.3.2. Mecanismos de sincronización y exclusión mutua 42
2.4. Soporte del sistema operativo a la concurrencia y paralelismo para las aplicaciones ..43
2.4.1. Procesos ligeros y Procesos 44
2.5. Algunas distribuciones actuales de paquetes multiflujo 46
2.5.1. Simplificar el trabajo sobre *threads* de kernel: CThreads 46
2.5.2. En busca de un estándar: Pthreads 47
2.6. Primitivas de sincronización a nivel sistema 47
2.6.1. Primitivas de sincronización ofrecidas para el usuario 47
2.6.2. Sincronización dentro del propio sistema 48
2.7. Aligerar los cambios de contexto: continuaciones explícitas 49
2.7.1. Modelos de ejecución en el kernel 49
2.8. Aligerar el contexto de los flujos: cooperación usuario-sistema 50
2.8.1. Scheduler-activations 50
2.8.2. Otros mecanismos de comunicación kernel-usuario: Objetos de primera clase ..52
2.9. Planificación en Mach 3.0 53
2.9.1. Colas de preparados 54
2.9.2. Estados de planificación y ejecución 55
2.9.3. *Threads* de kernel en el kernel de Mach 55
2.10. Mecanismo de *handoff* en el kernel de Mach 57
2.11. Continuaciones explícitas en el kernel de Mach 58
2.12. Referencias y Bibliografía 59

2.1. Introducción

Concurrencia es la noción de que dos o más cosas pueden suceder al mismo tiempo [DOEP87]. Cuando realmente suceden, hablamos de paralelismo. Desde la aplicación, el concepto de concurrencia implica diferentes flujos de control en un programa, independientemente del número de procesadores subyaciente. El número máximo de flujos que pueden ejecutarse concurrentemente en una aplicación diremos que es su grado de concurrencia [BLAC90] y de alguna manera, está identificando el máximo potencial de paralelismo que se puede conseguir de dicha aplicación.

Los sistemas multiprocesadores que han aparecido con las nuevas tecnologías, permiten paralelismo en la multiprogramación entre diferentes aplicaciones. Actualmente, el desarrollo del software y el mayor conocimiento de las aplicaciones permiten además la ejecución de una aplicación con varios flujos y explotar la concurrencia que, de manera inherente muchas veces se encuentra en los programas. Aunque se trabaje sólo con un procesador, la programación paralela mejora la concurrencia y facilita la claridad y sencillez en la resolución de los problemas.

Uno de los objetivos principales en las aplicaciones actuales es, pues, explotar al máximo el paralelismo que le ofrece la tecnología; en definitiva, diseñar programas paralelos. Programar con paralelismo supone una nueva filosofía en el modo de programar, equivalente a la innovación que introdujo el objetivo de simplicidad en el diseño de UNIX -quizá su corolario-.

Lo que antes era un único flujo secuencial con dependencia de llamadas y esperas de finalización de eventos o condiciones, se divide -se piensa- ahora como múltiples flujos independientes, no totalmente asíncronos en el tiempo, sino con posibles relaciones de sincronización o comunicación entre ellos, dependiendo del tipo de aplicación.

En su forma más pura, un flujo concurrente es un ejecutor de instrucciones, no incluye un código o datos concretos, aunque sí que va construyendo un estado propio, unos datos locales, una ejecución determinada.

En un momento dado, de esos flujos podemos diferenciar los que la aplicación activa -a través de los mecanismos de creación y destrucción que se le ofrecen -y gestiona -a través de llamadas a las funciones de sincronización y comunicación entre flujos-. Diremos que, desde el punto de vista del diseñador de la aplicación, es una concurrencia “explícita”.

Hay otro tipo de concurrencia, que llamaremos “implícita”, que afecta igualmente a la ejecución de los flujos de la aplicación y posiblemente a su estado, pero que no depende de ella: son las decisiones que vienen desde el sistema operativo, o desde el hardware a través del sistema operativo. Entran aquí los cambios de contexto entre procesadores virtuales (concurrencia en el nivel sistema), los bloqueos de flujos a la espera de un servicio del sistema, o el tratamiento de excepciones e interrupciones. Los flujos que producen estas acciones se activan al margen de las decisiones de la propia aplicación, aunque a veces sea ella misma la que las provoque (por ejemplo, las excepciones por errores).

El que se dé o no realmente paralelismo es un factor que depende del hardware. Pero el sistema operativo ha de facilitar la expresión de concurrencia de la aplicación en paralelismo de procesadores virtuales, o como hemos nombrado nosotros, paralelismo a nivel de aplicación.

Así como hemos marcado como función del lenguaje y de las librerías -de las herramientas software de programación a nivel de usuario, en general- la de permitir a la aplicación expresar su concurrencia, una de las funciones del SO es, por otra parte, la de ofrecer el máximo paralelismo virtual, para poder llegar a un mapeo de 1-1 con la concurrencia de la aplicación.

Con este paso se convierte la programación en elegante y limpia, porque elimina toda la truculencia y complejidad que se añade a veces a un programa -sin tener una relación directa con el problema a resolver- por el hecho de no disponer de los suficientes recursos. Es facilitar al diseñador de aplicaciones todos los recursos y elementos necesarios: la máquina virtual hecha a la

medida de su aplicación.

Esta sencillez y simplicidad en la programación, ya hacen eficientes de por sí a las aplicaciones, libres de toda retórica y elucubración que sólo dificulta las correcciones, el seguimiento, el crecimiento de los programas y, posiblemente, hasta reduce su velocidad de ejecución. Además, diseñados y elaborados por expertos, estos mecanismos son una carga que se suprime al programador, que aún con mucho esfuerzo no sería capaz de igualar.

El SO es un interlocutor entre el usuario y la máquina, y ha de ofrecer al primero mecanismos más potentes que los de los SO tradicionales, pero también con menos políticas de gestión.

Los microkernels han sido un primer paso, que ha dejado en el kernel sólo los bloques básicos de gestión del sistema: planificación, y gestión del hardware. Sin embargo, aún tienen un comportamiento excesivamente pesado en determinados aspectos.

De manera natural, también en la gestión de los procesadores del sistema, se van quedando en el kernel sólo unos mecanismos básicos de planificación de flujos de código, mientras las políticas van subiendo a niveles superiores. Es posible que el primer sistema operativo en trabajar de esta manera fuera Hydra [WULF81]. Actualmente, Mach [BLAC90], Concentrix [JACO86] y Psyche [SCOT90], entre otros, hacen que el usuario colabore con el sistema operativo en la planificación de la CPU.

Frente a la noción de prioridad que ofrece el sistema operativo, el usuario gradúa el rendimiento de su aplicación (posiblemente del sistema en general) mediante un control más fino en la planificación, asignando procesadores específicos a aplicaciones específicas, marcando flujos como no desbancables, proponiendo la ejecución alternativa de otros flujos, etc.

En los siguientes apartados veremos los modelos que se ofrecen al usuario para expresar concurrencia. Desde los flujos de granularidad muy fina, sin estado ni identificación reconocida por el sistema operativo, hasta los paquetes multiflujo.

Nuestro trabajo se incluye en este último tipo de concurrencia, de mayor granularidad de la que se puede extraer de un compilador, pero también con más potencia y versatilidad a la hora de diferenciar trabajos y relaciones entre ellos. Es por eso que dedicamos una mayor atención al estudio de librerías de flujos. Veremos algunas realizaciones comerciales y los trabajos que se están desarrollando hacia la definición de un interfaz estándar.

En la segunda parte de este capítulo damos una visión del sistema operativo como herramienta de paralelismo, siendo nuestro objetivo el encontrar estrategias de planificación de flujos apropiadas a las aplicaciones paralelas.

Vamos a ver algunas partes importantes del sistema que pueden afectar a la planificación de flujos, y sólo en cuanto que pueden afectar a ello; básicamente, serán la gestión de la memoria, la sincronización y los mecanismos de comunicación entre flujos (IPC). También describiremos realizaciones actuales de nuevos mecanismos y objetos que mejoran el soporte al paralelismo por parte del sistema operativo a las aplicaciones actuales.

Pero eficacia no es sinónimo de complejidad: los propios algoritmos de planificación pueden mejorar si mantienen su generalidad pero vuelven a la sencillez de los primeros sistemas multiprogramados.

Por último, hablaremos de las directrices actuales, que van hacia la cooperación entre el usuario y el sistema para mejorar el rendimiento de las aplicaciones.

2.2. Paradigmas de programación concurrente

Es importante identificar el máximo potencial de paralelismo que puede extraerse de un programa. Sin embargo, la división de una aplicación en múltiples flujos concurrentes depende de varios fac-

tores. La granularidad de los flujos, el tipo de sincronización que se da entre ellos o los mecanismos de comunicación que utilizan, además de la arquitectura sobre la que se trabaja, son algunos de ellos.

Hay herramientas que crean flujos automáticamente en una aplicación que se ha diseñado secuencial y hay lenguajes y librerías que permiten crear y destruir flujos explícitamente. Todo ello da lugar a diferentes paradigmas de programación paralela. Incluso pueden aparecer modelos mixtos, con diferentes tipos de flujos.

2.2.1. Compiladores y arquitecturas especializadas

Sin que el programador modifique sus programas secuenciales, puede extraerse paralelismo automáticamente, a partir de compiladores que lo permitan. Son los llamados compiladores paralelos, como Parafrase [POLY89].

Generalmente, esta ejecución paralela de una o varias -pocas- instrucciones se consigue gracias al soporte de arquitecturas especializadas con procesadores esclavos (desde los que son muy sencillos hasta los que cuentan con capacidades escalares, pero pocas veces de propósito general). La idea es dividir un trabajo en múltiples trabajos más pequeños, independientes entre ellos, que se lanzan en paralelo a los diferentes procesadores de cálculo.

Si hay más trabajos que unidades de cálculo, instrucciones de bajo nivel generan una cola de trabajos pendientes que se van ejecutando conforme cada procesador acaba su tarea (modelo de ejecución de *task queue*, pero soportado por el hardware). De igual modo, estos procesadores esclavos son infrautilizados cuando los programas no incluyen esas instrucciones especiales o el número de tareas es inferior.

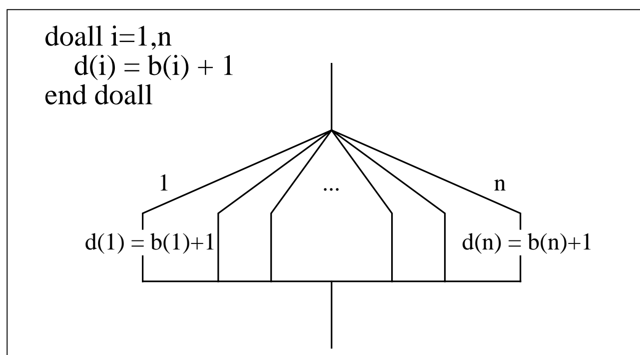

Figura 2-1 Este esquema sería el que resultaría de una sentencia DOALL, o FOR paralelo, en la que la secuencia principal envía la ejecución de una misma sentencia (o conjunto de ellas), pero diferentes datos, a un número N de flujos para que la calculen en paralelo. Al finalizar cada uno de ellos, envía el resultado a la secuencia principal, que continúa su ejecución.

Por el tamaño de código de que constan y al no haber sincronización ni comunicación intermedia, se habla de estos flujos como de granularidad fina y el sistema operativo no guarda ninguna información sobre ellos. Debido al poco estado e información que se mantiene de cada trabajo, no permiten imbricaciones con otras secuencias, de las que ya no se volvería. Por este motivo, sólo se permite un nivel de expansión de flujos.

Existen lenguajes con extensiones que generan directivas para el compilador. A través de ellas, se permite al programador señalar de manera explícita las secuencias de código que podrían ejecutarse concurrentemente. En esta línea están los PARBEGIN o DOALL, en FORTRAN. También hay que proporcionar sentencias que permitan forzar la secuencialidad de determinados tra-

bajos [CONV90].

En la Figura 2-1 podemos ver un ejemplo de la explosión de flujos que un compilador puede provocar al analizar las dependencias de un bucle que recorre unos datos en forma de vector o matriz.

Similar a la paralelización automática de bucles en FORTRAN, los "futures" se han desarrollado en el contexto de lenguajes aplicativos, tipo Lisp, con construcciones recursivas [WAGN92].

En la Figura 2-2 vemos cómo indicar, en tiempo de ejecución[7], que dos llamadas recursivas pueden ejecutarse concurrentemente, simplemente indicando en la primera la palabra "future":

```
( (future (funcion1(param))))
  (funcion2(param)))
```

Figura 2-2 La palabra "future" delante de la función recursiva funcion1(), indica que ésta se pueda ejecutar concurrentemente a funcion2().

La sincronización con el resultado de la función es implícita: el programador no pregunta en ningún momento por el resultado de la función. Si el flujo en ejecución necesita el valor y éste no ha llegado todavía, se bloquea de manera transparente.

Los "futures" son una herramienta muy útil para paralelizar funciones recursivas por el poco esfuerzo que suponen para el programador. Sin embargo, este mínimo esfuerzo se contrapone a una mayor eficiencia en tiempo de ejecución. Sobre todo, cuando el programador, podría reconocer cálculos de granularidad más fina como candidatos a ejecutarse concurrentemente.

2.2.2. Estructuras del lenguaje

Algunos lenguajes permiten crear y destruir flujos que ejecuten rutinas en paralelo. La idea que hay detrás del diseño de los "lenguajes paralelos" es que el programador se encargue de identificar qué trabajos pueden realizarse concurrentemente y tome las decisiones de cómo dividir ese trabajo, a partir de procesos o *tasks* [GEHA84]. Para ello, se ve un programa como compuesto de una parte principal y de un conjunto de subrutinas o funciones que son llamadas para ejecutar un determinado trabajo.

Cada una de estas rutinas y funciones, en un lenguaje imperativo, consta de un bloque de activación propio, que define su punto actual de ejecución, sus variables locales y su pila. Si en lugar de una secuencia de ejecución, se permiten varias, pueden definirse procesos que ejecuten las funciones y subrutinas en paralelo, con puntos de comunicación entre ellos para recoger los resultados.

La coordinación de estos flujos o procesos es más compleja y con más entidad que las tratadas en el punto anterior; puede soportar comunicación por mensajes como es el caso del rendezvous de Ada. Existe un bloque de activación propio y se permite la llamada a rutinas o funciones dentro de una *task* [ALMA89].

Estos procesos que el programador define y controla, pueden ser estáticos (Pascal concurrente y Modula) o dinámicos (Ada y PL/1). Depende de que se creen todos al comienzo de la ejecución del programa o puedan crearse en cualquier momento. La gestión de procesos dinámica necesita un soporte adicional en tiempo de ejecución.

7. Los lenguajes aplicativos son en su mayor parte interpretados.

2.2.3. Librerías

Otras veces, el control de creación y coordinación de flujos se le ofrece al usuario en forma de llamadas a librerías. El programador decide cuándo y dónde se crean los flujos en la aplicación y qué operaciones realizan. Con múltiples *threads* existen en un programa múltiples puntos de ejecución en cada instante, uno por *thread*, aunque podrían no ser todos ellos ejecutables simultáneamente.

Los flujos se ven en este modelo como un objeto dinámico, independiente de las rutinas (código) que pueden ejecutar. Se puede decir que tienen una cierta semántica, un objetivo de trabajo a realizar. Puede haber flujos diferentes que llamen a las mismas rutinas, en diferentes momentos; a la vez, un mismo flujo puede llamar a distintas rutinas y funciones sin perder su identidad.

En los flujos concurrentes que se crean a partir de las propias sentencias del lenguaje, puede verse el flujo como un ente pasivo, estático, que recibe una tarea concreta para realizar. La creación y destrucción podemos considerarlas automáticas desde el punto de vista del programador que, más que crear un flujo, define el trabajo y condiciones de creación del flujo[8].

Los flujos creados por librerías de usuarios es lo que propiamente conoceremos como procesos ligeros y en los que vamos a centrar nuestro estudio a partir de ahora.

Pensamos que son los que potencialmente más se parecen a la abstracción de flujo en ejecución que ofrece el sistema operativo. Por tanto, también la herramienta más adecuada para hacer más versátil y rica la gestión de flujos en el nivel de usuario.

2.3. Programación con procesos ligeros

El mecanismo más utilizado para concurrencia en la actualidad es el de procesos ligeros, o *threads*. Permiten dividir un programa en múltiples flujos de control, secuenciales cada uno de ellos, e independientes entre ellos [BIRR89].

A nivel de aplicación la concurrencia es ventajosa, tanto en sistemas monoprocesadores como en multiprocesadores. Permite trabajar con acontecimientos asíncronos programando síncronamente (pequeñas acciones secuenciales paralelas entre ellas, esa es la idea).

Entre sus ventajas, encontramos las siguientes:

- Permite a un programa aprovechar el hardware disponible: si es multiprocesador, mejor en sistemas de pocos procesadores (hasta 10) que muchos (alrededor de 1000).
- Adelanta trabajo útil mientras se espera la llegada de un dispositivo lento, al que se puede dedicar un flujo en concreto.
- Permite trabajar en varias acciones a la vez, adaptándose mejor a la manera de hacer del hombre.
- Puede dar un mismo servicio a diferentes clientes a la vez, replicando el mismo trabajo en diferentes flujos.
- Trabajos menos urgentes pueden retrasarse hasta que no haya nada más importante a

8. En la actualidad, los flujos de librería son un interfaz intermedio entre el lenguaje y el sistema. Así, por ejemplo, hay mapeos de *tasks* de Ada en flujos de la librería de Pthreads (Cfr. Proyecto PARTS).

realizar.

Con procesos ligeros las funciones de creación, existencia, destrucción y sincronización son lo suficientemente baratas como para explotar la concurrencia que se necesite. Por su propia definición (procesos ligeros) las soluciones a los problemas han de ser sencillas y los algoritmos de gestión, rápidos y eficientes[9].

En la Figura 2-3 se muestra un ejemplo de la construcción de un servidor multiflujo. El programador decide la cantidad de trabajadores que van a atender a los mensajes de petición de servicios de los clientes, graduando su grado de concurrencia. Los flujos solicitarán entrar en zonas de exclusión mutua cuando quieran acceder a datos globales del servidor.

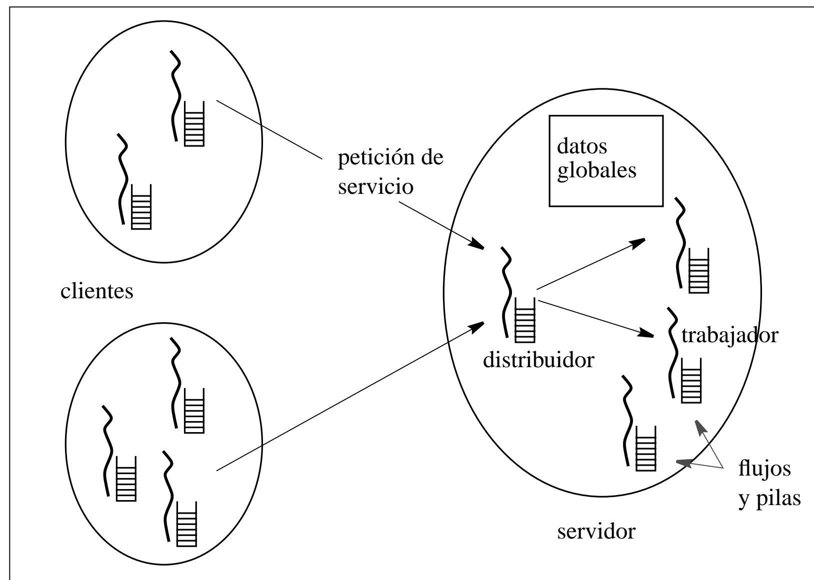

Figura 2-3 Ejemplo de programación con múltiples flujos de un servidor. Permite la atención de varias peticiones concurrentemente y la entrada/salida asíncrona.

Las librerías de *threads* permiten también objetos de granularidad media, con sincronizaciones entre ellos más frecuentes y ciclo de vida más corto. Es el caso de programas de visualización, trabajos en sistemas con múltiples ventanas, algoritmos de ordenación o búsqueda paralela, etc.

2.3.1. Mecanismos de creación

Las librerías multiflujo ofrecen llamadas para la creación de diferentes hilos de ejecución con una funcionalidad muy similar a la del entorno operativo de UNIX, familiar a la mayoría de usuarios.

Aunque el nombre varía dependiendo del paquete, tres son básicamente las funciones implicadas: fork(), join(), detach(). Con la llamada fork(), a la que generalmente se le pasa una función y, posiblemente unos parámetros, se crea un nuevo flujo que, a partir de ese momento está preparado para ejecutar. El flujo que ha realizado la llamada se dice que es el “padre” y, si la operación ha ido bien, recibe el identificador a partir del cual va a ser reconocido el nuevo flujo.

La coordinación del padre con la finalización del hijo se realiza a través de la llamada join(), y detach() en el caso en que no vaya a haber espera por finalización. Pocas veces se llama a join cuando se trabaja con flujos a nivel de librería. La mayoría de *threads* son de una granularidad

9. Cfr. con la filosofía de diseño de UNIX.

más bien gruesa, con puntos de comunicación aislados, pero vida independiente.

Por el coste que puede suponer la creación y destrucción en relación al trabajo a realizar, no es común que se utilicen para explosión de flujos de granularidad muy fina, tipo los generados automáticamente en sentencias DOALL y que son recogidos por el flujo principal al finalizar.

2.3.2. Mecanismos de sincronización y exclusión mutua

La programación con *threads* es inherentemente asíncrona sin necesidad de utilizar los antiguos esquemas de operaciones asíncronas (interrupciones, *signals*, excepciones). La manera más sencilla de sincronización entre *threads* es a través de memoria compartida [BIRR89]. Ésto implica una diferenciación explícita entre *threads* de la misma *task* y *threads* de *tasks* diferentes, que se comunican a través de mensajes.

El control de flujos a la espera de un recurso puede realizarse mediante esperas activas (*spin-lock*s o *spinning*) o bloqueando al flujo en cuestión. Hay que considerar, sin embargo, el peligro de trabajar con esperas activas si no existe paralelismo real en la aplicación, ya que podría caerse sin remisión en un bucle infinito, o *self-deadlock*. Por lo mismo, de cara a la portabilidad de las aplicaciones, es importante el control de las exclusiones mutuas ocultas que pueden darse en una programación que asume una ejecución en monoprocesadores.

Con variables de tipo monitor y variables de condición se permiten políticas de planificación más complicadas a la hora de decidir la secuencialidad y orden de entrada en una zona de exclusión mutua. Estos mecanismos suponen ya un primer paso hacia la planificación de flujos, decidida por la propia aplicación. A veces pueden llegar incluso a interaccionar directamente con el planificador del sistema. Es el caso de la inversión de prioridades, cuando un flujo no puede avanzar a la espera de que otro, menos prioritario pero en posesión de una exclusión mutua, la libere.

Para evaluar la mayor o menor bondad de las primitivas de sincronización que el sistema ofrece a la aplicación, es importante saber con el grado de paralelismo que se cuenta.

Las esperas activas ofrecen un método rápido y efectivo de comunicación. Sin embargo, cuando el número de flujos supera al de procesadores, las esperas activas pueden llevar a una caída brutal de rendimiento. Podría incluso caerse en abrazos mortales en aquellas primitivas que sincronizan varios procesos al estilo barrera (*barriers*).

A la hora de decidir si es conveniente una primitiva que bloquee o no, es importante medir la relación entre el coste de parar la ejecución de un flujo y poner a otro en marcha y el coste de esperar en un *spin-lock*, ocupando el procesador, para seguir avanzando más tarde.

En general, los bloqueos puede degradar el rendimiento cuando se utilizan primitivas de desbloqueo de grupos, es decir en *broadcast*. La utilización de wakeup(evento)[10] para dar paso al interior de una exclusión mutua es un ejemplo clásico: se despierta a un grupo de flujos, todos los que esperaban por el evento, pero sólo uno conseguirá pasar y el resto volverá a bloquearse [BIRR89].

Estudios realizados para diferentes aplicaciones paralelas, dan como mejor elección la llamada sincronización a dos niveles: esperar un cierto tiempo a conseguir la exclusión mutua y si no, bloquearse.

El tiempo de espera activa es relativo al coste del bloqueo, aunque una tesis experimental muestra que, en un punto de sincronización, o se consigue el paso en breve o se tardará bastante en hacerlo. Y entonces, es mejor bloquearse [ZAHO90], [GUPT91].

Ya se trabaje con un mecanismo de espera activa o con un mecanismo de bloqueo, el hecho de saber quién ha de ceder el paso o a quién hay que cedérselo, permite mejoras considerables en la construcción de herramientas. Se consigue un paso directo al proceso interesado (*handoff*) en lugar de trabajar con mecanismos automáticos [BLAC90].

En sistemas que trabajan con políticas de planificación por prioridades, se está estudiando el coordinar la posesión de exclusiones mutuas con dar la prioridad máxima a un proceso, consiguiendo así una "atomicidad" en la ejecución de partes de código que llevan a cuellos de botella del sistema [EDLE88]. En el sentido inverso, está el bajar la prioridad a los procesos ejecutando esperas activas, de modo que cedan el procesador a otros procesos que puedan avanzar y seguramente liberar la espera [ZAHO90].

2.4. Soporte del sistema operativo a la concurrencia y paralelismo para las aplicaciones

El sistema operativo es una herramienta adecuada y útil para dar soporte a la concurrencia y paralelismo que expresa la aplicación, sea cual sea su granularidad. Para ello, ha de ofrecer mecanismos y facilidades que simplifiquen la tarea del programador, respetando la premisa de que la sobrecarga introducida por la gestión de los flujos sea mínima.

El sistema es el que realmente gestionará los procesadores físicos como objetos propios y, con ello, el grado de paralelismo real de una aplicación en ejecución. El soporte que ofrece para que la aplicación pueda ejecutar sus flujos es el procesador virtual.

El procesador virtual es el objeto o abstracción que crea el sistema para contener el contexto de un flujo y corresponde a la imagen que tiene el usuario del procesador físico. Un flujo se está ejecutando realmente mientras el sistema asocie su procesador virtual a un procesador físico. El procesador virtual sirve igualmente para almacenar y conservar el valor del contexto del flujo cuando no está ejecutándose.

Dependiendo del número de procesadores virtuales (PV) que se ofrezcan a la aplicación y del número real de procesadores físicos (PF) que se le asignen, podrá haber paralelismo total, concurrencia a nivel de sistema, concurrencia a nivel de usuario o concurrencia a los dos niveles. La Figura 2-4 muestra gráficamente estas relaciones entre los dos mapeos existentes: entre flujos y

10. Enfocamos aquí el tema desde el punto de vista de la pérdida de eficiencia de despertar procesos que han de volver a ser bloqueados. No entramos en el tema de adaptación de estas primitivas de un entorno monoprocesador a uno multiprocesador (cfr. [NAVA91]).

procesadores virtuales, y entre estos últimos y los físicos (casos a, b, c y d) [GIL93].

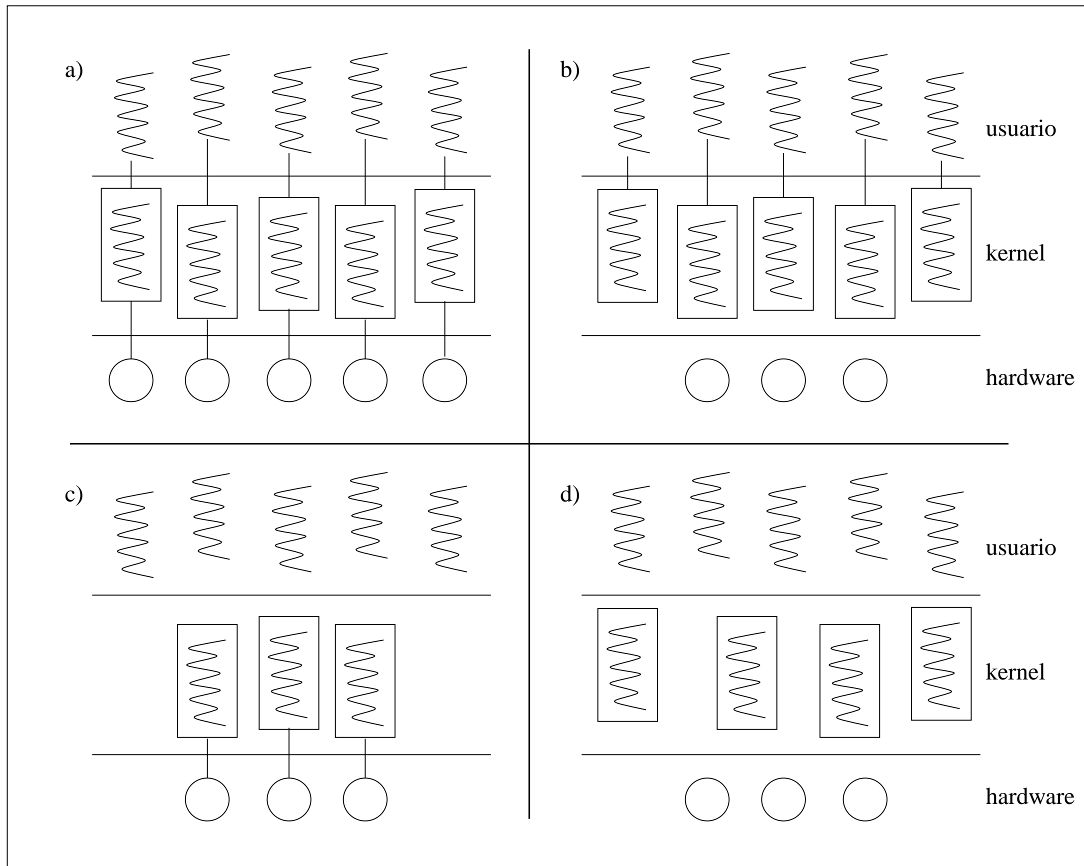

Figura 2-4 a) Paralelismo total: cada flujo tiene un procesador virtual y cada procesador virtual, un procesador físico. b) Concurrencia a nivel sistema: cada flujo tiene un procesador virtual, pero hay menos procesadores físicos, por lo que no todos los procesadores virtuales se estarán ejecutando simultáneamente. c) Concurrencia a nivel usuario: no hay tantos procesadores virtuales como flujos, y por tanto no pueden correr todos los flujos en el mismo momento. d) Concurrencia a nivel usuario y a nivel sistema.

2.4.1. Procesos ligeros y Procesos

Hay diferentes políticas para decidir, cuando hay concurrencia, qué flujo concreto se mapea en cada momento en un PV, y qué PV se mapea en cada PF [BLAC90], [CROV91], [GUPT91]. No vamos a entrar ahora en ninguna de ellas, sino que nos centraremos en qué es -o qué debería ser- un procesador virtual ofrecido por el sistema operativo.

En el sistema UNIX, y tradicionalmente, la ejecución de un flujo ha estado ligada a un proceso. En este caso el sistema únicamente nos permite ofrecer paralelismo a través de sus procesos. Destacamos dos consecuencias de esa limitación:

- A nivel de aplicación, encontramos que se necesita una fuerte compartición de recursos, ya que generalmente son flujos cooperantes, que acceden a los mismos datos. También son flujos que se comunican con frecuencia, por el mismo motivo.
- En cambio, a nivel de sistema, en un proceso al estilo UNIX, los dos entes máquina virtual y procesador virtual están integrados y orientados a la protección; la

separación entre procesos a la hora de cambiar un mapeo proceso-procesador, hace que el cambio de contexto tenga un coste elevado (cambio de registros, cambio de zona de memoria, cambio de estado de los dispositivos,...).

Si tenemos concurrencia a nivel de usuario con procesos (librerías de *threads* que permiten varios flujos sobre el mismo proceso), aparecen dos niveles de planificación totalmente independientes. El planificador del kernel lleva la gestión de los procesos que han de estar en ejecución (uno o varios para la aplicación) y el planificador de la librería lleva la planificación de los *threads*.

El kernel no tiene conocimiento de la existencia de los varios flujos que la aplicación está gestionando sobre el mismo procesador virtual. Así, el bloqueo de un flujo en un servicio del sistema, conlleva el bloqueo de su procesador virtual y, por tanto, para la ejecución de todos los demás flujos, que no pueden seguir su ejecución.

Hay un problema grave de incomunicación entre las decisiones del sistema -más generales- y las del usuario -específicas de la aplicación- que pierde funcionalidad y eficiencia. Con las librerías de usuario se consigue ganar en concurrencia, pero el paralelismo queda reducido al número de procesos que tenga asignados la aplicación.

Otro problema serio de limitación de recursos que se plantea cuando se diseñan programas paralelos sobre procesos es que el número de flujos puede ser del orden de cientos y como el kernel ve un solo flujo, la memoria máxima, el número máximo de canales abiertos, el poder hacer una única llamada al kernel simultánea, el tratamiento de excepciones,... todo ello está limitado a nivel de proceso (previsto para un solo flujo).

Si volvemos a la idea de procesador virtual que nos ofrece el sistema a través del proceso, vemos que en realidad están integrados en el mismo concepto el procesador virtual y la propia máquina virtual que el sistema ofrece a la aplicación[11].

Puesto que como conceptos están perfectamente diferenciados, nada nos impide separarlos y poder hablar de una máquina virtual multiprocesador, con varios procesadores virtuales. Cada uno de esos procesadores virtuales, será un proceso ligero de kernel.

Los tradicionales procesos de UNIX y sistemas derivados, entendidos como un entorno de ejecución y un flujo corriendo en su interior, se escinden ahora en dos nuevos paradigmas: por un lado, el entorno de ejecución -proceso, *task*, o actor, según la terminología utilizada- y, por otro lado, el flujo de ejecución que puede correr en él - el *thread*-. A partir de esta separación, el sistema puede reconocer más de un hilo de ejecución corriendo en el mismo espacio de direcciones. Existe la posibilidad de proporcionar a cada uno un procesador físico: existe la posibilidad de paralelismo.

Al igual que con los procesos ligeros que proporcionan las librerías, cada proceso ligero del sistema tiene una identidad propia, reconocida y gestionada por el sistema: un identificador, los registros, una pila, una prioridad,... (Figura 2-5).

Las mejoras son notables en cuanto a eficiencia de los cambios de contexto entre procesadores virtuales de una misma *task*, y la facilidad de programación con la compartición de datos y recursos entre *threads*, que son reconocidos por el kernel. Pero sigue habiendo un problema de fondo, quizás ahora acrecentado: la gran cantidad de estructuras de datos -y su tamaño- necesarias

11. Así, el coste de tener una aplicación formada por varios procesos, es análogo al “peso” de comunicación que podemos encontrar en un sistema multicomputador frente al de un multiprocesador de memoria compartida.

para la gestión de los antes decenas y ahora centenas de procesadores virtuales. El descriptor de un proceso ligero dentro del sistema no es mucho menor que el de la *task* o el proceso, ya que la parte que más espacio ocupa es la pila[12]. El kernel así crece y ocupa mucha memoria física, robándosela a las *tasks* que han de trabajar con menos memoria.

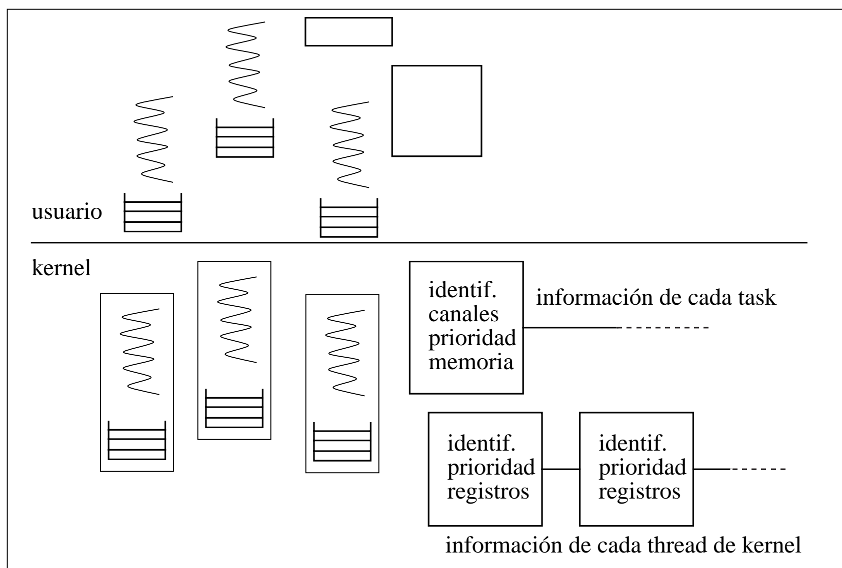

Figura 2-5 *Threads* de kernel. Parte de los datos que mantenía antes el sistema para el proceso (ahora *task*), la mantiene ahora en la estructura de datos de cada *thread*.

2.5. Algunas distribuciones actuales de paquetes multiflujo

Existen entornos de ejecución, sobre todo enfocados a sistemas de desarrollo, que ofrecen la posibilidad de programar con procesos ligeros a nivel de usuario. Entre ellos, el Uniform System [THOM88] sobre arquitecturas NUMA, y PCR ("Portable Common Runtime") [WEIS89], sobre SunOS/SPARC.

Existen también algunos sistemas que ofrecen librerías multiflujo en su distribución. A continuación pasamos a citar algunas de ellas.

2.5.1. Simplificar el trabajo sobre *threads* de kernel: CThreads

Con objeto de simplificar la programación sobre *threads* de kernel en el microkernel Mach 3.0 [COOP88], se diseñó la librería de CThreads. Viene en las distribuciones de Mach, tanto en la de CMU, como en la de OSF/1. Su objetivo principal es la programación de los servidores que componen el entorno que ofrece el microkernel.

Permite la creación de flujos de usuario que compartan un número limitado de *threads* de kernel. Ofrece mecanismos de sincronización entre flujos, con cambio de contexto en caso de bloqueo por exclusión mutua o por sobrepasar un límite de flujos haciendo simultáneamente llamadas al kernel. Un flujo puede ceder su procesador virtual a otro, pero no posee políticas de planificación ni la posibilidad de desbanque entre flujos.

Por formar parte de nuestro entorno de desarrollo, nos referiremos a ella con más detalle

12. Actualmente se da una página por pila y una página de memoria en las arquitecturas de hoy no suele bajar de los 4Kbytes.

en el capítulo correspondiente[13].

2.5.2. En busca de un estándar: Pthreads

Se está realizando, recientemente, el esfuerzo de especificar un estándar IEEE para sistemas operativo transportables sobre sistemas abiertos. Como parte de este trabajo, está la extensión de *threads* POSIX 1003.4a, que describe el interfaz de *threads* de usuario sobre memoria compartida, con un contexto menor que el tradicional de los procesos [IEEE92]. El interfaz de Pthreads especifica un modelo de *threads* basado en prioridades y con políticas de planificación preemptivas, gestión de *signals* y primitivas de sincronización que permiten exclusión mutua. La definición de las llamadas a la librería sigue en curso y las realizaciones existentes no suelen cumplir toda su especificación.

Se están realizando versiones de Pthreads, tanto para sistemas monoprocesadores, como para sistemas multiprocesadores. En función de la arquitectura y del sistema operativo subyaciente, el interfaz se ha realizado de diferentes maneras.

La realización sobre SunOS/SPARC, hecha por Frank Mueller de la Florida State University, es quizá la más completa y que más servicios realizados tiene [MUEL93]. Se asume, sin embargo un entorno UNIX monoprocesador y se utilizan algunas de las utilidades, que captura la propia librería para redefinirlas. Es el caso de los *signals*. Su realización entra dentro del proyecto PARTS, para adaptar un entorno de Ada sobre flujos de usuario y es de libre distribución. La hemos incluido en este apartado por ser "POSIX-complaint".

En las distribuciones de OSF/1 vienen dos librerías multiflujo: CThreads y PThreads. Esta última en versión multiprocesador [OSF92], a diferencia de la de [MUEL93]. En este caso se nota la influencia de CThreads en las estructuras de datos y primitivas. Sin embargo, las funciones de planificación más potentes definidas por POSIX, como son el permitir distintas políticas, algunas de ellas preemptivas, no están realizadas.

2.6. Primitivas de sincronización a nivel sistema

Tradicionalmente, el sistema operativo ha dado soporte a las aplicaciones para la sincronización y comunicación entre procesos a través de llamadas al sistema. A este nivel, los métodos utilizados influyen en el rendimiento del sistema, al interactuar con las políticas de planificación y por ser los procesos -procesadores virtuales- los que realmente están asignados a los procesadores físicos.

Por otro lado, el sistema operativo, como aplicación paralela él mismo, también cuenta con puntos de sincronización y exclusiones mutuas en su propio código.

2.6.1. Primitivas de sincronización ofrecidas para el usuario

Una aplicación paralela, está compuesta de diversos procesadores virtuales que cooperan y se comunican entre ellos. Esta comunicación entre procesos (IPC) puede afectar a su propia ejecución y, por tanto, interaccionar con la planificación de los mismos.

Los mecanismos de comunicación que tradicionalmente viene ofreciendo el SO a la aplicación son dos: memoria compartida y paso de mensajes.

Cuando dos procesos utilizan memoria compartida para comunicarse entre ellos, el sistema ha de proporcionar el soporte necesario para que exista esta zona de memoria compartida. En los microkernels actuales, si los procesadores virtuales son *threads* de kernel en la misma *task*, el

---

13. Ref. "Mecanismos y Políticas fuera del kernel", apartado 4.3.

acceso a memoria compartida es automático, por definición. Si son *threads* de kernel pertenecientes a diferentes *tasks*, el sistema ha de ofrecer algún servicio (llamada al sistema) que permita definir regiones de memoria mapeables en los diferentes espacios de memoria y accesibles desde todos ellos por igual.

La coherencia de la información mantenida en esa memoria compartida, es responsabilidad de la propia aplicación. Utilizará para ello mecanismos de acceso en exclusión mutua y primitivas de sincronización facilitadas por las librerías, de las que ya hemos hablado anteriormente.

Además de los mecanismos de sincronización que puede ofrecer la librería o el lenguaje, el propio sistema operativo puede cooperar con la aplicación. Así, el hecho de que un flujo de usuario quede bloqueado a la espera de un evento, no significa que quede bloqueado su procesador virtual. Si no hay más trabajos planificables en la aplicación y ésta no quiere una espera activa -o no le conviene, como puede ocurrir cuando se está en un sistema monoprocesador, o la aplicación cuenta sólo con un procesador virtual y corre el peligro de quedarse en un abrazo mortal-, puede forzar el bloqueo del procesador virtual para que el sistema deje pasar a otro.

Los mecanismos de control de concurrencia ofrecidos por el sistema operativo para memoria compartida son clásicos en la literatura sobre el tema (semáforos y eventos, principalmente) [ALMA89] [BACH86] [GUPT91].

El otro mecanismo que proporciona el sistema para sincronización y comunicación entre procesos es el mecanismo de paso de mensajes.

Este mecanismo se adapta mejor a los modelos de cliente/servidor que se diseñan actualmente en las tecnologías microkernel. Sirven igual sobre arquitecturas UMA, que NUMA o NORMA, por lo que permite una mejor transportabilidad de las aplicaciones. El envío puede ser directo a un proceso o a un *port* o *mailbox* general. Generalmente, se necesitan derechos de envío y recepción, que puede repartir el propio proceso dueño del canal lógico. El sistema, sin embargo, es el que controla la comunicación y, por tanto, da las reglas del juego [ACCE86], [CHER84].

2.6.2. Sincronización dentro del propio sistema

Para el código del propio sistema operativo, es esencial que la retención sea mínima. Con la extracción de partes de trabajo fuera del propio sistema en las tecnologías microkernel, ya se ha conseguido parte de este objetivo: hacer desbancables servicios que antes no lo eran.

Al tratar con datos globales a todas las aplicaciones y al propio sistema (a través de las rutinas de servicio de interrupciones y excepciones), los cuellos de botella en los accesos a estructuras de información (colas, tablas,...) son un punto crítico en todo sistema operativo.

Una técnica extendida hoy día consiste en unir a cada estructura información sobre si es conveniente o no esperar por la obtención de la estructura. En las operaciones de inserción y extracción de colas, por ejemplo, se asocia un contador de los elementos que hay en la cola: si no hay ninguno, no se intenta la extracción. Como la consulta de este dato no se realiza atómicamente, sólo es una pista; pero ayuda a mejorar el rendimiento [BIRR89], [BLAC90].

Otro factor de sincronización que aparece en los sistemas operativos paralelos, es el hecho justamente de estar ejecutándose en paralelo diferentes acciones, algunas de ellas posiblemente sobre los mismos objetos. Así, por ejemplo, un flujo puede empezar el proceso de bloqueo hasta el mismo instante de parar su ejecución. En ese instante, puede darse cuenta de que el evento por el cual iba a esperar ya ha llegado y por tanto, continuar él en ejecución. Es lo que se conoce también como bloqueo en dos pasos: primero notificar el bloqueo, después bloquearse.

Este mecanismo también se hace a nivel de aplicación con flujos de usuario[GIL93].

Las filosofías más extendidas hoy día abogan por "debilitar" los protocolos de entrada y salida en las exclusiones mutuas.

Tradicionalmente, dichos protocolos aumentan el tiempo de trabajo en sistema y, la mayoría de las veces es para comprobar que el camino está libre. La idea es que, en general, la exclusión mutua está libre y, por tanto no es necesario ningún protocolo de entrada. Es más eficiente seguir adelante y, en caso necesario, dar marcha atrás (la probabilidad de hacerlo es muy pequeña). Cuando el hardware lo permite, esta marcha atrás puede hacerse en más puntos del código y más eficientemente: son las secuencias atómicas recuperables o "ras" (*recoverable atomic sequences*) [BERS92].

2.7. Aligerar los cambios de contexto: continuaciones explícitas

Es frecuente, en las aplicaciones actuales, trabajar con un gran número de flujos, lo que provoca un aumento en la frecuencia de transferencias de control entre flujos. La latencia inherente a las continuaciones implícitas existentes (pila de kernel y Process Control Block) afectan directamente al rendimiento.

Por otra parte, las soluciones que se ofrecen hoy -aplicaciones a nivel usuario que realizan trabajos tradicionales del SO-, aumentan también la frecuencia de transferencias de control entre espacios de direcciones diversos.

Los modelos multiflujo trabajan con múltiples procesadores virtuales por espacio de direcciones. Ésto puede significar la existencia simultánea de cientos de procesadores virtuales por máquina, que supondría mantener la gestión simultánea de cientos de pilas de kernel por máquina.

Sin embargo:

El coste asociado a cambiar de entorno de ejecución depende, no sólo de la cantidad de información que compone el contexto de un flujo, sino también del modelo de ejecución bajo el que se trabaje.

2.7.1. Modelos de ejecución en el kernel

Ha habido hasta ahora dos modelos básicos de ejecución dentro del kernel:

Modelo de proceso, en el que, cuando un proceso se bloquea en kernel, guarda parte de su contexto en la pila, por lo que se ha de conservar su pila de kernel durante todo el tiempo que dure su bloqueo, y existe una pila por cada flujo de ejecución en el kernel.

Modelo de interrupción, en el que no se mantiene información dentro del kernel cuando el proceso se bloquea, por lo que no es necesario mantener la pila de kernel del proceso durante el bloqueo, y es suficiente mantener una pila de kernel por procesador.

Se propone, ahora, un tercer modelo de ejecución [DRAV91]: ***Modelo de continuaciones***, en el que un flujo de ejecución puede trabajar con el modelo de interrupción (continuación explícita) o con el modelo de proceso (continuación implícita). Puede existir más de una pila de kernel por procesador, pero menos de una por flujo.

Para indicar las continuaciones explícitas, existen objetos de kernel indicando lo que va a

hacer un *thread* después de la transferencia de control que se ha hecho, en términos de interfaz independiente de la máquina y en representación compacta.

Las optimizaciones conseguidas en tiempo de ejecución afectan tanto al espacio de datos requerido como al tiempo necesario para llevar a cabo una acción. Se reduce el espacio al no tener que mantener el estado necesario para volver a poner en marcha el flujo en la pila. Generalmente se trata de puntos en que el estado a mantener es muy pequeño o incluso nulo. Debido también a este poco contexto que hay que guardar desde que se para una ejecución hasta que vuelve a reemprenderse, se reduce la latencia de los cambios de contexto entre flujos.

Mach 3.0 trabaja con este modelo de continuación explícita dentro del kernel, en los puntos donde reconoce que no es necesario guardar el contexto. Lo veremos con más detalle en el apartado 2.11.

2.8. Aligerar el contexto de los flujos: cooperación usuario-sistema

Existen actualmente varias propuestas que, partiendo de diferentes mecanismos y filosofías, confluyen en una misma opinión: ciertas decisiones que hasta ahora se tomaban en el kernel, deberían subir a nivel usuario, manteniendo una conversación kernel-usuario y usuario-kernel y/o permitiendo al usuario la lectura de información residente en el kernel.

El objetivo es:

- Proporcionar la **funcionalidad** de los *threads* de kernel para que no existan procesadores sin trabajo mientras haya flujos de ejecución disponibles; que no haya flujos esperando ser ejecutados mientras trabajan otros de menor prioridad y que, cuando un flujo entre en el kernel para bloquearse, se pueda planificar otro del mismo, o diferente, espacio de direcciones.
- Mantener el **rendimiento** de los paquetes de *threads* más rápidos, del orden de magnitud de una llamada a procedimiento para cada operación.
- Tener la **flexibilidad** de poder cambiar con facilidad la política de planificación para los flujos de una aplicación, o incluso de poder proporcionar diferentes modelos de concurrencia para diferentes aplicaciones dentro del mismo sistema.

En definitiva, se trata de proveer al nivel usuario con el mismo tipo de información temporal y opción de planificar que la que dispone habitualmente el kernel. Con ese fin, surgió el mecanismo de las scheduler-activations y los objetos de primera clase a nivel usuario, entre otros. Ambos se han desarrollado en entornos experimentales y los comentamos a continuación con un poco más de detalle.

2.8.1. *Scheduler-activations*

En el modelo de *scheduler-activations*, propuesto por [ANDE90], cada aplicación dispone de un multiprocesador virtual dedicado y cada aplicación sabe exactamente cuántos y qué procesadores tiene asignados y controla totalmente los flujos que trabajan sobre ellos, como si estuvieran corriendo directamente sobre la máquina física.

El kernel tiene el control completo de la asignación de procesadores a espacios de direcciones, que incluso puede ir variando en ejecución, pero no de la gestión de los acontecimientos que afectan a la planificación de los flujos en ejecución: en lugar de interpretarlos, avisa al planificador de la aplicación de los eventos que puedan afectarle (cambio de los procesadores que tiene asignados o cuando un *thread* de usuario se bloquea o se despierta en el kernel), para que los gestione él. Así, se consigue la misma **funcionalidad** que en los sistemas que trabajan con *threads* de kernel.

La aplicación, por su parte, avisa al kernel cuando necesita más o menos procesadores; en general, sólo de aquellos eventos a nivel de usuario que puedan afectar a la asignación de procesadores, quedando las decisiones de *scheduling* de *threads* a nivel de usuario, sin el coste de tener que comunicar con el kernel, y consiguiéndose así un mayor **rendimiento**.

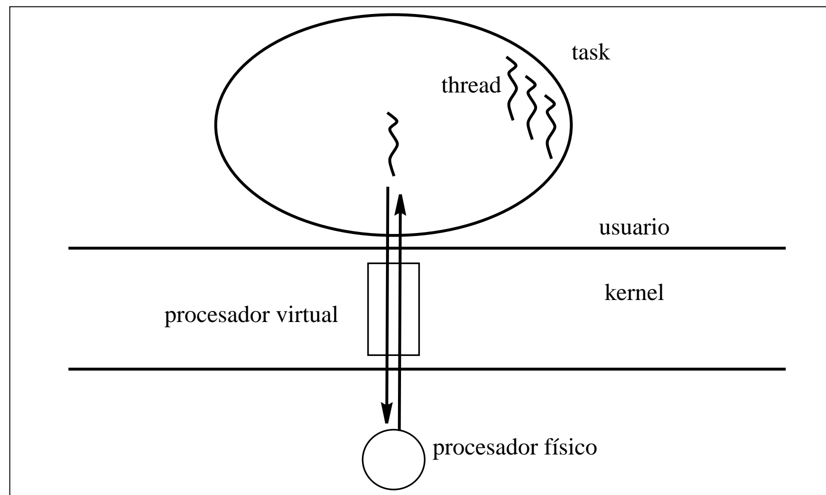

Figura 2-6 La aplicación tiene control directo sobre el procesador físico que tiene asignado cada uno de sus flujos, a través del procesador virtual (*scheduler-activation*) que le proporciona el sistema.

El mecanismo utilizado es el de *scheduler-activations*: es un contexto de ejecución de un evento enviado por el kernel a un espacio de direcciones por medio de una ***upcall*** (llamada del kernel hacia el usuario).

El nombre de *scheduler-activation* (*s-a*, para abreviar) viene motivado por el hecho de que cada evento que llega del kernel hace que el planificador de la aplicación reconsidere la planificación de los flujos que tienen que estar ejecutándose.

Las funciones de un *s-a* son:

- Servir como **contexto de ejecución** para los *threads* de usuario.
- **Notificar** al sistema de *threads* de nivel usuario de **un evento** ocurrido en el kernel.
- Proporcionar un **espacio donde salvar el contexto** del procesador de la activación actual de un flujo cuando se bloquee en el kernel (por desbanque o a la espera de una E/S).

La idea subyaciente es ofrecer una “vasija” a la aplicación para que ésta la llene con el flujo que crea conveniente, como muestra la Figura 2-7.

Cada estructura de datos de una *s-a* tiene una pila mapeada en el espacio de kernel, para el flujo que corre en su contexto cuando trabaja en modo kernel, y una pila mapeada en el espacio de usuario, en la que corre el planificador de la aplicación cuando se activa. En la aplicación se mantiene información sobre los flujos que hay en cada una de las *s-a*.

Toda interacción del kernel con la aplicación se produce en términos de *scheduler-activations*, por lo que se puede trabajar con cualquier modelo de concurrencia por encima de las *scheduler-activations*.

La diferencia crucial entre una *s-a* y un *thread* de kernel es que un flujo bloqueado en el kernel, cuando se desbloquee, no será planificado por el kernel directamente. Se utilizará una nueva activación para avisar a la aplicación del evento y ella decidirá qué flujo puede ser planificado en el procesador[14].

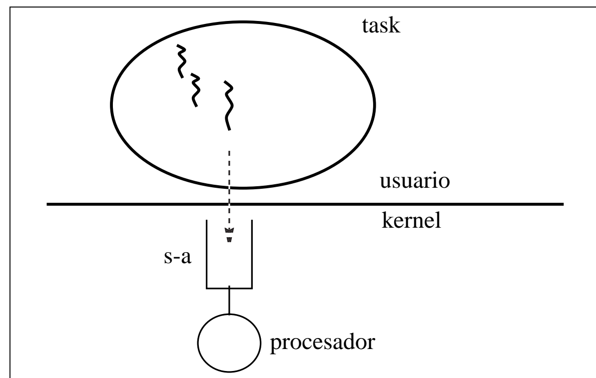

Figura 2-7 Las *s-a* son vasijas en las cuales la aplicación va poniendo el flujo que conviene en cada momento.

2.8.2. Otros mecanismos de comunicación kernel-usuario: Objetos de primera clase

En la misma línea está el trabajo presentado por [MARS91], cuyo propósito es el de subir mecanismos y convenciones del kernel para otorgar un *status* de primera clase a los *threads* de usuario. Así se permite que, usados de una manera razonable, puedan utilizarse de manera eficiente los procesadores virtuales proporcionados por el kernel, dejando los detalles de la definición de modelos y políticas a nivel usuario.

Para ello, es necesario:

- Memoria compartida que permita una comunicación asíncrona entre kernel y usuario, a través de la cual el kernel tiene acceso a la información sobre el estado de los flujos que tiene el paquete.
- Interrupciones software enviadas por el planificador del kernel a la aplicación (upcalls), ante los eventos que sean gestionados por la propia aplicación.
- Una convención sobre lo que ha de ser el interfaz de planificación entre el kernel y la aplicación para facilitar la interacción entre diferentes paquetes de *threads* u otros modelos de flujos a nivel de usuario.

La información sobre el estado de los flujos se mantiene en el espacio de usuario y es también ahí donde se realiza la mayoría de las operaciones, incluyendo creación, destrucción, sincronización y cambios de contexto.

Las estructuras de datos compartidas por el kernel y el usuario para la planificación están situadas en unos pseudo-registros (un banco por cada procesador físico), mapeados sólo para lectura en el espacio de direcciones de cada aplicación, en una posición fija, estática (Figura 2-8). La información que contienen es: identidad del procesador, punteros al PV actual y punteros al espacio de direcciones activo actualmente. Los usuarios pueden modificar la información apuntada por estos punteros y convierten así sus objetos en objetos de “primera clase”[15].

Para proporcionar verdadero paralelismo a la aplicación, existe un mapeo uno a uno entre los procesadores físicos y los virtuales, que están representados por procesos de kernel, posiblemente más de uno en el mismo espacio de direcciones. De esta manera, además, la aplicación puede controlar de manera “directa” los procesadores físicos sobre los que trabaja, como en el modelo anterior de las *scheduler-activations*.

14. En un sistema tradicional, cuando el kernel para un *thread* de kernel, nunca se avisa al nivel de usuario del acontecimiento, ni tampoco cuando más tarde vuelva a ser resumido.
15. Se habla de objetos de primera clase para designar aquellos objetos que tienen privilegios de kernel para determinadas acciones.

Este paradigma de trabajo se ha desarrollado en un entorno totalmente experimental, sobre un kernel construido desde cero: Psyche [SCOT90].

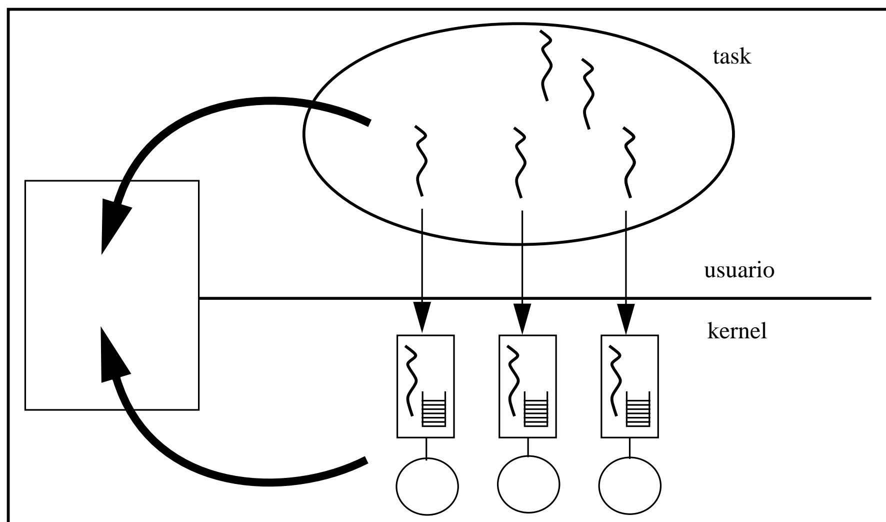

Figura 2-8 Gracias a un espacio de direcciones común al usuario y al kernel, la aplicación puede controlar la planificación de sus flujos y tomar la mayoría de decisiones.

2.9. Planificación en Mach 3.0

La planificación de Mach ha ido cambiando, madurando, a lo largo de su existencia, contribuyendo a ello los trabajos de investigación que sobre ella se han realizado. El más importante hasta el momento es el desarrollado por David Black [BLAC90].

Los objetos de Mach directamente relacionados con la planificación son los *threads*, las *tasks*, los procesadores y los *processor-sets* (pset, para abreviar). Estos últimos, son la base de la planificación en Mach que proporciona el kernel. Ofrecen una submáquina lógica sobre la cual se aplican las políticas de planificación de Mach a los *threads* y procesadores que lo componen.

A través de los psets, el sistema permite dedicar procesadores a las aplicaciones, de modo que trabajen como si estuvieran aisladas en un multiprocesador más pequeño.

Por defecto, y en la inicialización del sistema, existe un único pset, el pset "default", que contiene todos los procesadores y *threads* del *host*. Los *threads* del kernel pertenecen a este pset y por eso siempre tiene que haber, como mínimo, un procesador en él. Además, el procesador master del sistema, que se encarga de realizar operaciones globales del sistema como unidad (actualización de la hora, por ejemplo) también pertenece a este pset.

La realización de Mach 3.0 proviene directamente de la de Mach 2.5, versión monolítica del microkernel, con 4.xBSD en su interior. El tener que adaptar el funcionamiento de Mach y el de UNIX, sin alterar en sus fundamentos la semántica ofrecida por este último, y el hecho de que la planificación de procesos UNIX no es sencilla de seguir, ni de modificar, pensamos que han sido dos factores que han influido fuertemente en el diseño de la propia planificación de flujos en Mach. No obstante, se han mejorado algunos de los inconvenientes de realización achacados a UNIX [GIL94].

Es importante notar ya desde el comienzo que la planificación de Mach está diseñada para un sistema multiprocesador, y en éso basa sus políticas y mecanismos, aunque funciona también en sistemas monoprocesador. La filosofía que sigue es la de detectar y corregir los fallos, en lugar

de impedirlos. El ser un sistema abierto a la investigación, de libre acceso y en continuo desarrollo, le han convertido en base de experimentación y origen de muchas de las iniciativas explicadas en los apartados anteriores.

2.9.1. Colas de preparados

Las unidades de planificación en Mach son los *threads*. Saber que dos *thread* pertenecen a la misma *task* es útil para optimizar el cambio de contexto entre ellos, pero no se utiliza como política de selección de un *thread*. No existe una planificación centralizada, sino que cada procesador se planifica a sí mismo: es self scheduling. El kernel gestiona la planificación de *threads* preparados para ejecutar a partir del mecanismo de colas que introdujo UNIX (*run queues*).

Existen dos tipos de colas de preparados: locales y globales. El sistema mantiene una cola de preparados local a cada procesador, con *threads* que sólo pueden ser planificados en ese procesador y una cola de preparados global por pset, en la que están los *threads* que pueden ejecutarse en cualquiera de los procesadores de ese pset.

En la cola local de un procesador sólo hay *threads* que se hayan ligado a ese procesador. Es una operación que queda circunscrita al código del sistema y sólo se aplica en circunstancias especiales; por ejemplo, para realizar operaciones sólo autorizadas al master, un *thread* tiene que ser planificado en dicho procesador, y algunos *threads* de kernel también necesitan ir a un procesador en concreto para actuar sobre él[16].

La realización de ambas colas de preparados, local y global, es la misma, utilizando la misma estructura de datos. Esta estructura contiene los siguientes elementos:

- runq[], es la cola multinivel que contiene todos los elementos. En total, son 32 niveles, cada uno correspondiente a una prioridad, de 0 a 31, siendo la prioridad 0 la mayor y la 31 la menor.
- low, es un indicio sobre la cola que contiene al *thread* preparado para ejecutar de mayor prioridad. Así se consigue una búsqueda más optimizada a la hora de replanificar. Este valor se ha de ir actualizando y sólo garantiza que no hay ningún *thread* preparado a una prioridad mayor.
- count, es el número actual de *threads* preparados para ejecutar en esa cola. Esto permite evitar la sobrecarga de acceder en exclusión mutua en caso de estar vacía.

Además, existe un *lock* para acceder en exclusión mutua a la cola en la inserción y extracción de *threads*. Son colas con gestión FIFO.

Existen dos posibles políticas de planificación gestionadas por el kernel: tiempo compartido y prioridades fijas. Con esta última se puede emular un comportamiento de tiempo real por software[17]. Para tiempo compartido, Mach trabaja con prioridades dinámicas: se asocia un *quantum* de tiempo a cada *thread* al final del cual se le recalcula la prioridad.

Una cuestión que se plantea es cómo dividir el tiempo entre las diferentes prioridades y cómo las prioridades afectan al uso de los procesadores. En multiprocesadores, está comprobado experimentalmente que se reduce el número -y con ello el tiempo total- de cambios de contexto alargando el *quantum* cuando el número de flujos es menor que dos veces el número de procesadores [BLAC90]. Mach, además, para evitar que dos procesadores coincidan en el mismo instante de tiempo en su decisión de cambio de contexto, alarga el *quantum* con fracciones del *quantum* extendido de modo que el tiempo esperado para acceder a la runq sea el mínimo *quantum*[18].

16. Ver más adelante el action *thread*.
17. Existen además proyectos de investigación en curso desarrollando versiones de Mach para tiempo real (ARTS).
18. Para evitar apelotonar los procesadores y que el acceso a la runq cause retrasos.

2.9.2. Estados de planificación y ejecución

Existe una interacción entre el módulo encargado de los remplazos de memoria y el módulo de planificación. Para que un *thread* sea planificable, es condición indispensable que su pila de kernel esté en memoria física (si no, no es posible tratar los fallos de página). Extraer un *thread* de memoria (*swap out*) consiste simplemente en marcar como reemplazable su pila de kernel por el mecanismo de memoria virtual. Cuando vuelve a entrar un *thread*, su pila de kernel se lleva a memoria física y se fija, no permitiendo su salida.

En Mach existen 14 posibles estados de planificación [BLAC90] para un *thread* que podemos agrupar en:

- *Threads* en un procesador (cuatro estados posibles),
- *threads* encolados para ser ejecutados (dos estados posibles),
- *threads* bloqueados (tres estados posibles), y
- *threads* fuera de memoria (cinco estados posibles).

Como combinación de estos estados, y desde el punto de vista de su planificación, un *thread* está en un momento dado “ejecutable”, “esperando un evento” o “suspendido”. Un *thread* en estado “ejecutable” puede estar corriendo en un procesador, o en una cola de preparados esperando ser seleccionado por un procesador. Si está “esperando un evento” estará bloqueado en la cola correspondiente a la espera de que dicho evento ocurra, o de que se cancele el bloqueo por otros motivos.

El estado “suspendido” lo utiliza el kernel para evitar la ejecución de un *thread*. Se emplea habitualmente para *threads* que van a ser destruídos, o cuando necesita registrarse el estado actual del *thread* (*snapshot* state). Este último caso es de utilidad cuando se trabaja con librerías multiflujo, o para pasar información a programas de usuario, en general. Un *thread* suspendido no está en ninguna cola, ni de preparados ni de espera de un evento.

2.9.3. *Threads* de kernel en el kernel de Mach

El propio kernel de Mach es visto en muchos aspectos como una *task* más, que ofrece servicios especiales. Como toda *task*, tiene sus *threads* para realizar determinados trabajos. Son, claro está, *threads* especiales, equivalentes a los procesos de sistema en otras tecnologías. Así, por ejemplo, no tienen estado de usuario.

No todos los servicios que ofrece Mach lo hace a partir de *threads*, pero sí que cuenta con unos determinados *threads*. Algunos de ellos, por su influencia directa en la planificación de las aplicaciones del sistema, vamos a verlos a continuación.

2.9.3.1. Idle thread

Como en todo sistema multiprocesador, cada procesador tiene su propio idle thread, para absorver el tiempo que el procesador no está realizando un trabajo útil a ninguna de las aplicaciones del sistema.

El trabajo del idle thread consiste en un bucle de encuesta continua, mirando si hay algún *thread* en alguna de las colas de preparados (la global o la local de su procesador) o si le han asignado un *thread* en directo, por encontrarse su procesador en la cola de procesadores sin trabajo[19].

19. Existe un puntero en la estructura de datos del procesador que apunta al *thread* que se le ha pasado en directo, haciendo *bypass* de la runq.

2.9.3.2. Recálculo de prioridades: sched thread

Cada *thread* en Mach modifica su prioridad, bien con la interrupción de reloj o cuando un evento desbloquea al *thread*. Hay un *quantum* asociado a cada *thread* y en su finalización, se provoca un cambio de contexto si existe un *thread* de igual o mayor prioridad que el actual. Si no existe dicho *thread*, se asigna al *thread* un nuevo *quantum*.

Para los *threads* que llevan tiempo sin ser ejecutados y, por tanto, pueden tardar tiempo en subir su prioridad, un *thread* de kernel, el sched thread, recorre la cola cada dos segundos actualizando las prioridades de los *threads*.

2.9.3.3. Gestión de procesadores: action thread

Para el control de procesadores, Mach tiene un *thread* específico: el action thread[20]. Es el encargado de parar o poner en marcha los procesadores, de forma lógica, y también el encargado de asignarlos a aplicaciones.

Su forma de trabajo es la siguiente: habitualmente está bloqueado a la espera de un evento. Cuando se pide una operación sobre un procesador (pararlo, ponerlo en marcha, moverlo de pset) se inserta una petición en la cola de trabajos del action thread y se despierta a éste.

Una vez en funcionamiento, va sirviendo los trabajos hasta que la cola quede vacía y vuelve a bloquearse. Como Mach tiene una política de self-scheduling en todas sus acciones, sólo se puede modificar el estado de un procesador desde el mismo procesador. Para ello, el action thread tiene que "vincularse" al procesador sobre el que va a operar insertándose en su cola local de preparados y esperar al siguiente punto de replanificación del procesador (como máximo, un *tick* de reloj). Sólo entonces puede actuar.

Debido a lo complejo del mecanismo y a los elementos que implica en ello (información del procesador, del pset, de los *threads*, de las *tasks*), el action thread es un ejemplo claro de acciones que pueden dar marcha atrás por haber cambiado las condiciones de inicio. Por ejemplo, podría reasignarse un procesador a un pset diferente, antes de haber finalizado una operación anterior de asignación.

No obstante, la asignación de procesadores a psets no es una operación frecuente ni usual, fuera de ambientes de investigación. Y parar y poner en marcha procesadores, es inestable en la mayoría de entornos actuales de trabajo.

2.9.3.4. Asignación de procesadores

En Mach existen tres componentes para la gestión y asignación de procesadores:

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

El interfaz del kernel está diseñado alrededor de la abstracción de *processor-set*, que agrupa un conjunto de procesadores y un conjunto de *threads*. Puede verse como una submáquina dentro de la propia máquina, en la que los procesadores que le han sido asignados sólo pueden seleccionar como planificables los *threads* que pertenecen a dicho *processor-set*.

La protección y autentificación en la asignación de procesadores se hace controlando el acceso a los *ports* que representan los objetos involucrados (*port rights*). Los *ports* de los procesa-

20. En las distribuciones actuales sólo hay un action thread. Pero está diseñado para que puedan coexistir varios en el mismo kernel.

dores (control *port*) son del servidor para garantizar su control; para obtenerlos, es necesario el *port* del *host* que, de momento sigue un mecanismo UNIX: hay que ser super user[21]. Este *port* proporciona el privilegio de control de los recursos físicos, mientras que el *port* de control del pset (en principio, pertenece a la aplicación que lo ha creado) proporciona el poder controlar la planificación del *processor set*.

Para cada *processor set*, existe su copia de estructuras de datos:

- Una cola de *threads* preparados para ejecución global para sus procesadores, y
- una lista de procesadores idle.

Además, en la estructura de datos del *processor set* están las cabezas de las listas de *tasks*, *threads* y procesadores que incluye.

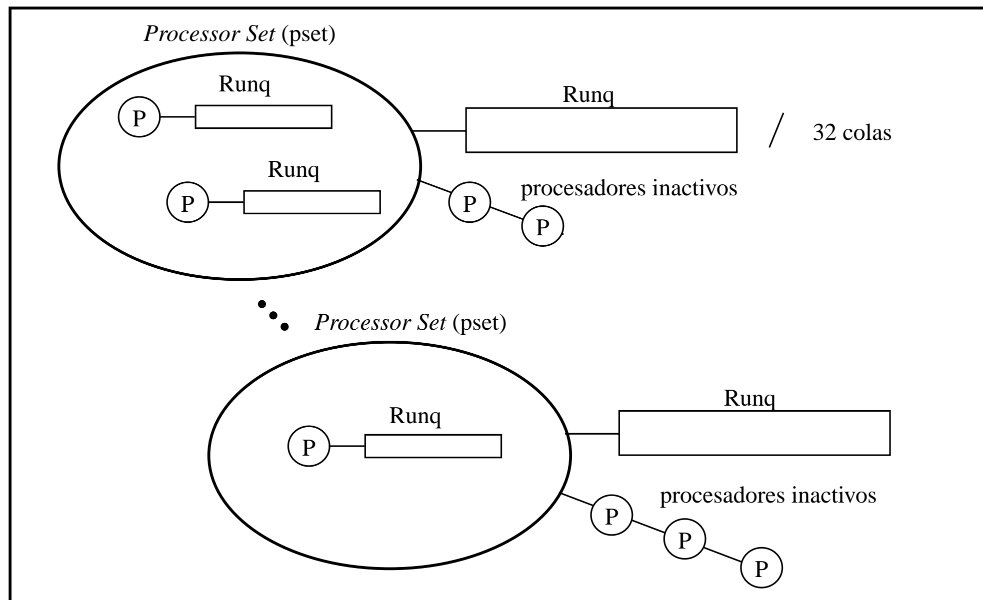

Figura 2-9 Relación entre psets, procesadores y colas de preparados. Cada pset tiene, además su cola de procesadores sin trabajo.

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).

2.10. Mecanismo de *handoff* en el kernel de Mach

Cuando un flujo va a bloquearse en la sincronización de un servicio y sabe en concreto qué otro flujo es el que le ha de proporcionar ese servicio, puede cederle directamente el procesador y el tiempo que le resta de él, por el mecanismo de *handoff*. Este mecanismo existe en Mach tanto a nivel de usuario como en el interior del kernel.

A nivel usuario, existe una llamada al sistema que permite dar pistas al kernel sobre la conveniencia de un cambio de contexto. Estas pistas pueden proporcionarse pidiendo un recálculo de prioridades, bajándose explícitamente la prioridad, bloqueándose por un tiempo determinado, o proporcionándole un *thread* concreto para planificar. En este último caso no se aplica la política de

21. Otra opción es que se le asigne a la primera *task* del sistema y ella decida.

prioridades.

En el interior del kernel, el mecanismo de *handoff* se ha introducido para optimizar el código de comunicaciones. Mach es un kernel que ofrece sus servicios basándose en el modelo de cliente-servidor: prácticamente la totalidad de las llamadas al sistema van a través de mensajes, utilizando el trap mach_msg_trap[22].

El servicio a mach_msg_trap examina el amplio abanico de posibilidades que pueden darse, e intenta la optimización de todos los casos, ya que es el servicio más utilizado del sistema: tanto para la comunicación con el resto de *tasks*, como para requerir los propios servicios del sistema.

Cuando la opción del servicio es de un envío y recepción, es decir, un recibo con respuesta (*reply*), la optimización o camino más rápido es intentar hacer un *handoff* entre el flujo que envía y el que ha de recibir:

En esta situación, el *thread* que envía no podrá continuar hasta que el receptor realmente acabe su trabajo y devuelva una respuesta: lo mejor es pasarle a él directamente el procesador, sin tener que esperar a que, después de habernos bloqueado nosotros, algún procesador lo replanifique[23].

Hacemos aquí una pausa para ver con más detalle el proceso.

El servicio que se ha pedido incluye todo el camino de envío de mensaje-recepción y servicio a la petición-vuelta de servicio finalizado. De hecho, es cuando se recibe este último mensaje que queda por concluida la petición. Así, en realidad, el desbloqueo del receptor y su recorrido dentro del kernel, es trabajo propiamente del cliente, aunque realizado por un *thread* distinto[24]. No es un altruismo cualquiera el ceder tiempo de procesador y el procesador mismo a dicho *thread*, sino más bien un adelantar trabajo que va a redundar en beneficio.

Sólo en el caso de no existir ningún impedimento a los requisitos de planificación de Mach (que el receptor pertenezca a diferente *processor set* o que esté ligado a un procesador diferente), podrá llevarse a cabo el *handoff*.

2.11. Continuaciones explícitas en el kernel de Mach

Para optimizar el espacio utilizado y la gestión de los *threads* de kernel de Mach, se ha rehecho su esquema de trabajo original para que puedan trabajar con modelo de interrupción, es decir con continuaciones explícitas.

El modelo de trabajo de un *thread* de kernel es el de un servidor que se haya en un bucle infinito a la espera de un servicio. Cuando llega una petición se pone a trabajar y al acabar vuelve a bloquearse a la espera. Los servicios son independientes uno de otro, pues provienen de diferentes *threads* o diferentes procesadores, o diferentes momentos y situación. Por lo tanto, cada petición, se sabe que va a empezar siempre en un punto concreto (el principio del servicio) y con un estado concreto (nulo, por parte del *thread* de kernel).

Con esta base, se ha eliminado el bucle infinito por un código secuencial (correspondiente al interior del bucle en las primeras versiones de Mach) que comienza en cada servicio y que acaba con un bloqueo, indicando una continuación explícita: de nuevo el comienzo del flujo.

Otro punto que se ha mejorado en eficiencia con continuaciones explícitas es el de las

---

22. Los otros tres o cuatro traps que existen en el sistema son para llamadas locales o llamadas en experimentación.
23. Mach, además ofrece una optimización mayor con un *handoff* de la propia pila. No entramos aquí en este segundo *handoff* porque nos parece que no ha lugar.
24. Podríamos equipararlo perfectamente a la entrada en kernel de un *thread* para realizar un servicio de éste, y el cambio de modo respectivo.

comunicaciones con mensajes. De hecho, una de las condiciones para poder realizar el *handoff* entre envío y recepción es saber exactamente dónde ha de continuar el trabajo el receptor. Es decir, tiene que haber una continuación explícita. En este caso además puede realizarse no sólo un paso directo a otro flujo, si no que se le puede pasar directamente la pila (*stack handoff*).

Recordemos que cuando se trabaja con continuaciones explícitas (modelo de interrupción), no se conserva la pila del objeto. Al *thread* que vamos a activar por el *handoff*, le tenemos que buscar, por tanto, una nueva pila. En ella además, tenemos que introducirle los parámetros e información del mensaje que está recibiendo. Por parte del cliente, el estado actual de su bloque de activación -su pila-, se le va a modificar conforme se lleve a cabo el servicio. Además, en él están los parámetros e información que necesita el servidor para continuar: lo más eficiente es pasarle también la pila en su estado actual. Así, no solo se gana el tiempo de gestionar la memoria para asignar una nueva pila al receptor, sino que se ahorra todo el paso y copia de parámetros, tanto en un sentido como en el otro, cuando el servicio haya finalizado: se le devolverá la pila con el estado correcto y seguro.

2.12. Referencias y Bibliografía

[ACCE86] “Mach: A New Kernel Foundation for UNIX Development”
Mike Accetta et al.
Proceedings of the Summer 1986 Usenix Conference, July 1986.

[ALMA89] “Highly Parallel Computing”
George S. Almasi and Allan J. Gottlieb
The Benjamin/Cummings Publishing Company, Inc., 1989.

[ANDE90] “Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism”
Technical Report 90-04-02, Department of Computer Science and Engineering, University of Washington, Seattle, WA, October 1990.

[BACH86] “The Design of the Unix Operating System”
Maurice J. Bach
Prentice-Hall International Edtions, 1986

[BERS92] “Fast Mutual Exclusion for Uniprocessors“
Brian N. Bershad, David D. Redell and John R. Ellis
Fifth Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS V), October 1992.

[BIRR89] “An Introduction to Programming with Threads”
Andrew D. Birrell
Research Report 35, DEC Systems Research Center, 1989.

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

[CHER84] “The V Kernel: A Software Base for Distributed Systems”
David Cheriton
IEEE Software, April 1984.

[CONV90] “CONVEX FORTRAN Optimization Guide”
CONVEX Computer Corporation, Second Edition, 1990.

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

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

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

[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 OSR Vol.25 Num.5, October 1991.

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

[GEHA84] “Ada Concurrent Programming”
Narain Gehani
Prentice-Hall, 1984.

[GIL 93] “Aligerar los procesos ligeros. Estado del arte.”
Marisa Gil y Nacho Navarro
UPC/DAC Report RR93/07, Febrero 1993.

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

[GUPT91] “The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications”
Anoop Gupta, A. Tucker and S. Urushibara
ACM SIGMETRICS, May 1991
Performance Evaluation Review, Vol.19 Num.1, 1991

[JACO86] “A User-Tunable Multiple Processor Scheduler”
Herb Jacobs
USENIX Winter Conference Proceedings, pp 183-191, January 1986.

[MARS91] “First-Class User-Level Threads”

Brian D. Marsh et al.
ACM OSR, Vol.25 Num.5, October 1991.

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

[NAVA91] “Paralelismo de Procesos e Interrupciones en el Transporte de UNIX a un Multiprocesador”
Nacho Navarro
Tesis Doctoral, UPC, Octubre 1991.

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

[POLY89] “Parafrase 2: An Environment for Parallelizing, Partitioning, Synchronizing and Scheduling of Programs on Multiprocessors”
C. Polychronopoulos et al.
Proceedings of the International Conference on Parallel Processing, Vol.II, pp. 39-48, 1989.

[SCOT90] “Multi-Model Parallel Programming in Psyche”
Michael L. Scott, Thomas J. LeBlanc and Brian D. Marsh
Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming, Seattle WA, March 1990.

[THOM88] “The Uniform System: An Approach to Runtime Support for Large Scale Shared Memory Parallel Processors”
Robert H. Thomas and Will Crowther
Proceedings of ICPP 88, Vol. II Software, August 1988.

[WAGN92] “Portable, Efficient Futures”
David B. Wagner and Bradley G. Calder
Technical Report CU-CS-609-92, Department of Computer Science
University of Colorado, Boulder, COLORADO, August 1992.

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

[WULF81] “Hydra/C.mmp: An Experimental Computer System”
W. A. Wulf, R. Lavin and S. P. Harbison
McGraw-Hill, 1981.

[ZAHO90] “Processor Scheduling in Shared Memory Multiprocessors”
John Zahorjan and Cathy McCann
Proceedings of SIGMETRICS90, 1990