Sobre la Construcción de Micro-kernel

Jochen Liedtke

GMD | German National Research Center for Information Technology

jochen.liedtke@gmd.de

Resumen

Desde el punto de vista de la tecnología del software, el concepto de micro-kernel es superior a los kernel integrados grandes. Por otro lado, se cree ampliamente que (a) los sistemas basados en micro-kernel son inherentemente ineficientes y (b) no son suficientemente flexibles. Contrario a esta creencia, mostramos y respaldamos con evidencia documental que la ineficiencia y la inflexibilidad de los micro-kernel actuales no se heredan de la idea básica, sino que principalmente provienen de la sobrecarga del kernel y/o de una implementación inadecuada.

Basados en razones funcionales, describimos algunos conceptos que deben ser implementados por un micro-kernel e ilustramos su flexibilidad. Luego, analizamos los puntos críticos de rendimiento. Mostramos qué rendimiento es alcanzable, que la eficiencia es suficiente con respecto a los macro-kernel y por qué algunas mediciones publicadas contradictorias no son evidentes. Además, describimos algunas técnicas de implementación e ilustramos por qué los micro-kernel no son inherentemente portables, aunque mejoran la portabilidad de todo el sistema.

1. Fundamento

Los sistemas basados en micro-kernel se han construido mucho antes de que el término mismo fuera introducido, por ejemplo, por Brinch Hansen en 1970 y Wulf et al. en 1974. Tradicionalmente, la palabra 'kernel' se usa para denotar la parte del sistema operativo que es obligatoria y común a todo el otro software. La idea básica del enfoque de micro-kernel es minimizar esta parte, es decir, implementar fuera del kernel todo lo posible.

Las ventajas tecnológicas de software de este enfoque son obvias:

(a) Una interfaz de micro-kernel clara refuerza una estructura de sistema más modular.

(b) Los servidores pueden usar los mecanismos proporcionados por el micro-kernel como cualquier otro programa de usuario. El mal funcionamiento del servidor está tan aislado como el mal funcionamiento de cualquier otro programa de usuario.

(c) El sistema es más flexible y adaptable. Diferentes estrategias y APIs, implementadas por diferentes servidores, pueden coexistir en el sistema.

Aunque se ha invertido mucho esfuerzo en la construcción de micro-kernel, el enfoque no es (todavía) generalmente aceptado. Esto se debe al hecho de que la mayoría de los micro-kernel existentes no funcionan suficientemente bien. La falta de eficiencia también limita severamente

la flexibilidad, ya que importantes mecanismos y principios no pueden usarse en la práctica debido al pobre rendimiento. En algunos casos, la interfaz del micro-kernel ha sido debilitada y servidores especiales han sido reintegrados en el kernel para recuperar eficiencia.

Se cree ampliamente que la mencionada ineficiencia (y por lo tanto inflexibilidad) es inherente al enfoque del micro-kernel. La creencia popular sostiene que los aumentos en los cambios de modo usuario-kernel y los cambios de espacio de direcciones son responsables. A primera vista, las mediciones de rendimiento publicadas parecen apoyar esta vista.

De hecho, los estudios de rendimiento citados midieron el rendimiento de un sistema particular basado en micro-kernel sin analizar las razones que limitan la eficiencia. Solo podemos adivinar si es causado por el enfoque del micro-kernel, por los conceptos implementados por este particular micro-kernel o por la implementación del micro-kernel. Dado que se sabe que el IPC convencional, uno de los cuellos de botella tradicionales del micro-kernel, puede implementarse un orden de magnitud más rápido de lo que se creía antes, la pregunta sigue abierta.

Por las razones mencionadas anteriormente, sentimos que es necesario un análisis conceptual que derive los conceptos de micro-kernel de los requisitos de funcionalidad pura (sección 2) y que discuta el rendimiento alcanzable (sección 4) y la flexibilidad (sección 3).

Otras secciones discuten la portabilidad (sección 5) y las oportunidades de algunos nuevos desarrollos (sección 6).

2 Algunos Conceptos del Micro-kernel

En esta sección, razonamos sobre los conceptos mínimos o "primitivas" que un micro-kernel debería implementar.^[3] El criterio determinante utilizado es la funcionalidad, no el rendimiento. Más precisamente, un concepto solo es tolerado dentro del micro-kernel si moverlo fuera del kernel, es decir, permitir implementaciones competidoras, impediría la implementación de la funcionalidad requerida por el sistema.

Suponemos que el sistema objetivo debe soportar aplicaciones interactivas y/o no completamente confiables, es decir, debe tratar con la protección. Además, suponemos que el hardware implementa memoria virtual basada en páginas.

Un requisito inevitable para tal sistema es que un programador debe ser capaz de implementar un subsistema arbitrario S de tal manera que no pueda ser perturbado ni corrompido por otros subsistemas S0. Este es el principio de independencia: S puede ofrecer garantías independientes de S0. El segundo requisito es que otros subsistemas deben poder confiar en estas

garantías. Este es el principio de integridad: debe haber una manera para que S1 se dirija a S2 y establezca un canal de comunicación que no pueda ser corrompido ni interceptado por S0.

Suponiendo que el hardware y el kernel son confiables, se pueden implementar servicios de seguridad adicionales, como los descritos por Gasser et al. en 1989, por servidores. Su integridad puede ser asegurada por la administración del sistema o por servidores de arranque a nivel de usuario. Para ilustrar: un servidor de claves debería entregar pares de claves RSA públicas-secretas bajo demanda. Debe garantizar que cada par tenga la propiedad RSA deseada y que cada par se entregue solo una vez y solo al solicitante. El servidor de claves solo puede realizarse si hay mecanismos que (a) protejan su código y datos, (b) aseguren que nadie más lea o modifique la clave y (c) permitan al solicitante verificar si la clave proviene del servidor de claves. La búsqueda del servidor de claves se puede realizar mediante un servidor de nombres y verificarse mediante autenticación basada en claves públicas.

2.1 Espacios de Direcciones

A nivel de hardware, un espacio de direcciones es un mapeo que asocia cada página virtual a un marco de página física o la marca como `no accesible'. Por simplicidad, omitimos atributos de acceso como solo lectura y lectura/escritura. Este mapeo es implementado por el hardware de TLB y las tablas de páginas.

El micro-kernel, que es la capa obligatoria común a todos los subsistemas, debe ocultar el concepto de hardware de los espacios de direcciones, ya que de otro modo, implementar la protección sería imposible. El concepto de espacios de direcciones del micro-kernel debe ser controlado, pero debe permitir la implementación de esquemas de protección (y de no protección) arbitrarios sobre el micro-kernel. Debería ser simple y similar al concepto de hardware.

La idea básica es apoyar la construcción recursiva de espacios de direcciones fuera del kernel. Como si fuera magia, existe un espacio de direcciones 0 que representa esencialmente la memoria física y está controlado por el primer subsistema S0. Al inicio del sistema, todos los demás espacios de direcciones están vacíos. Para construir y mantener más espacios de direcciones sobre el 0, el micro-kernel proporciona tres operaciones:

Conceder (Grant). El dueño de un espacio de direcciones puede conceder cualquiera de sus páginas a otro espacio, siempre que el receptor esté de acuerdo. La página concedida se elimina del espacio de direcciones del que concede y se incluye en el espacio de direcciones del receptor. La restricción importante es que, en lugar de marcos de página físicos, quien concede solo puede conceder páginas a las que ya tiene acceso.

Mapear (Map). El dueño de un espacio de direcciones puede mapear cualquiera de sus páginas en otro espacio de direcciones, siempre que el receptor esté de acuerdo. Después, la página puede ser accedida en ambos espacios de direcciones. A diferencia de conceder, la página no se elimina del espacio de direcciones del que mapea. Al igual que en el caso de conceder, quien mapea solo puede mapear páginas a las que él mismo ya tiene acceso.

Limpiar (Flush). El dueño de un espacio de direcciones puede limpiar cualquiera de sus páginas. La página limpiada permanece accesible en el espacio de direcciones del que limpia, pero se elimina de todos los demás espacios de direcciones que recibieron la página directa o indirectamente de este. Aunque no se requiere el consentimiento explícito de los dueños de los espacios de direcciones afectados, la operación es segura, ya que está restringida a sus propias páginas. Los usuarios de estas páginas ya aceptaron la posibilidad de una limpieza cuando recibieron las páginas por mapeo o concesión.

El Apéndice A contiene una definición más precisa de los espacios de direcciones y las tres operaciones mencionadas.

Razonamiento

El concepto de espacio de direcciones descrito deja la gestión de la memoria y la paginación fuera del micro-kernel; solo se retienen dentro del kernel las operaciones de conceder, mapear y limpiar. El mapeo y la limpieza son necesarios para implementar gestores de memoria y paginadores sobre el micro-kernel.

La operación de conceder se requiere solo en situaciones muy especiales: considera un paginador F que combina dos sistemas de archivos subyacentes (implementados como paginadores f1 y f2, operando sobre el paginador estándar) en un sistema de archivos unificado (ver figura 1).

En este ejemplo, f1 mapea una de sus páginas a F, quien luego concede la página recibida al usuario A. Al conceder, la página desaparece de F, de modo que luego solo está disponible en f1 y el usuario A; los mapeos resultantes están denotados por la línea delgada: la página está mapeada en el usuario A, f1 y el paginador estándar. Limpiar la página por parte del paginador estándar afectaría a f1 y al usuario A, limpiar por f1 solo afectaría al usuario A. F no se ve afectado por ninguna limpieza (y no puede limpiar por sí mismo), ya que usó la página solo de manera transitoria. Si F hubiera usado el mapeo en lugar de conceder, habría necesitado replicar la mayor parte de la contabilidad que ya se realiza en f1 y f2. Además, conceder evita un posible desbordamiento del espacio de direcciones de F.

En general, conceder se utiliza cuando los mapeos de páginas deben pasar a través de un subsistema controlador sin sobrecargar el espacio de direcciones del controlador con todas las páginas mapeadas a través de él.

El modelo se puede extender fácilmente a los derechos de acceso en las páginas. Mapear y conceder copian el derecho de acceso de la página fuente o un subconjunto de ellos, es decir, pueden restringir el acceso pero no ampliarlo. Operaciones especiales de limpieza pueden eliminar solo derechos de acceso especificados.

I/O

Un espacio de direcciones es la abstracción natural para incorporar puertos de dispositivos. Esto es obvio para la entrada/salida (I/O) mapeada en memoria, pero también se pueden incluir puertos de I/O. La granularidad del control depende del procesador dado. El 386 y sus sucesores permiten control por puerto (una página muy pequeña por puerto) pero no permiten el mapeo de direcciones de puerto (esto fuerza mapeos con v = v0); el PowerPC utiliza I/O puramente mapeada en memoria, es decir, los puertos de dispositivos pueden ser controlados y mapeados con una granularidad de 4K.

Por lo tanto, el control de los derechos de I/O y los controladores de dispositivos también se realiza por gestores de memoria y paginadores sobre el micro-kernel.

2.2 Hilos e IPC (Comunicación Interproceso)

Un hilo es una actividad que se ejecuta dentro de un espacio de direcciones. Un hilo se caracteriza por un conjunto de registros, incluyendo al menos un puntero de instrucción, un puntero de pila y una información de estado. El estado de un hilo también incluye el espacio de direcciones en el que se ejecuta actualmente. Esta asociación dinámica o estática a espacios de direcciones es la razón decisiva para incluir el concepto de hilo (o algo equivalente) en el micro-kernel. Para prevenir la corrupción de espacios de direcciones, todos los cambios en el espacio de direcciones de un hilo deben ser controlados por el kernel. Esto implica que el micro-kernel incluye la noción de alguna actividad mencionada anteriormente. En algunos sistemas operativos, puede haber razones adicionales para introducir hilos como una abstracción básica, por ejemplo, la preemción.

Es importante destacar que la elección de un concepto concreto de hilo sigue siendo sujeta a decisiones de diseño específicas del sistema operativo.

En consecuencia, la comunicación entre espacios de direcciones, también llamada comunicación interproceso (IPC), debe ser soportada por el micro-kernel. El método clásico es transferir mensajes entre hilos mediante el micro-kernel. El IPC siempre impone un cierto acuerdo entre ambas partes de una comunicación: el emisor decide enviar información y determina su contenido; el receptor determina si está dispuesto a recibir información y es libre de interpretar el mensaje recibido. Por lo tanto, el IPC no es solo el concepto básico para la comunicación entre subsistemas, sino también, junto con los espacios de direcciones, la base de la independencia.

Otras formas de comunicación, como la llamada a procedimiento remoto (RPC) o la migración controlada de hilos entre espacios de direcciones, pueden construirse a partir de IPC basado en transferencia de mensajes.

Nota que las operaciones de concesión y mapeo (sección 2.1) necesitan IPC, ya que requieren un acuerdo entre el otorgante/mapeador y el receptor del mapeo.

Supervisión del IPC

Arquitecturas como las descritas por Yokote en 1993 y K!uhnhauser en 1995 no solo necesitan supervisar la memoria de los sujetos, sino también su comunicación. Esto se puede hacer introduciendo canales de comunicación o Clanes Liedtke 1992, que permiten la supervisión del IPC por servidores definidos por el usuario. Estos conceptos no se discuten aquí, ya que no pertenecen al conjunto mínimo de conceptos. Solo mencionamos que los Clanes no sobrecargan el micro-kernel: su costo base es de 2 ciclos por IPC.

Interrupciones

La abstracción natural para las interrupciones de hardware es el mensaje IPC. El hardware se considera como un conjunto de hilos que tienen identificadores de hilo especiales y envían mensajes vacíos (consistiendo solo en el id del emisor) a hilos de software asociados. Un hilo receptor concluye de la id de la fuente del mensaje si el mensaje proviene de una interrupción de hardware y de cuál interrupción:

Transformar las interrupciones en mensajes debe ser realizado por el kernel, pero el micro-kernel no está involucrado en el manejo específico de interrupciones de dispositivos. En particular, no sabe nada sobre la semántica de las interrupciones. En algunos procesadores, restablecer la interrupción es una acción específica del dispositivo que puede ser manejada por los controladores a nivel de usuario. La instrucción iret entonces se utiliza únicamente para sacar información de estado de la pila y/o volver al modo de usuario y puede ser ocultada por el

kernel. Sin embargo, si un procesador requiere una operación privilegiada para liberar una interrupción, el kernel ejecuta esta acción implícitamente cuando el controlador emite la siguiente operación IPC.

2.3 Identificadores Únicos

Un micro-kernel debe suministrar identificadores únicos (uid) para algo, ya sea para hilos, tareas o canales de comunicación. Los uid son necesarios para la comunicación local confiable y eficiente. Si S1 quiere enviar un mensaje a S2, necesita especificar el destino S2 (o algún canal que conduzca a S2). Por lo tanto, el micro-kernel debe saber qué uid se relaciona con S2. Por otro lado, el receptor S2 quiere estar seguro de que el mensaje proviene de S1. Por lo tanto, el identificador debe ser único, tanto en el espacio como en el tiempo.

En teoría, también podría usarse la criptografía. En la práctica, sin embargo, cifrar mensajes para la comunicación local es demasiado costoso y se debe confiar en el kernel de todos modos. S2 tampoco puede confiar en capacidades puramente suministradas por el usuario, ya que S1 u otra instancia podría duplicarlas y pasarlas a subsistemas no confiables sin control de S2.

3 Flexibilidad

Para ilustrar la flexibilidad de los conceptos básicos, esbozamos algunas aplicaciones que típicamente pertenecen al sistema operativo básico pero que pueden implementarse fácilmente sobre el micro-kernel. En esta sección, mostramos la flexibilidad principal de un micro-kernel. Si realmente es tan flexible en la práctica depende en gran medida de la eficiencia alcanzada del micro-kernel. El tema del rendimiento posterior se discute en la sección 4.

Gestor de Memoria. Un servidor que gestiona el espacio de direcciones inicial 0 es un gestor de memoria principal clásico, pero fuera del micro-kernel. Los gestores de memoria pueden apilarse fácilmente: M0 mapea o concede partes de la memoria física (0) a 1, controlado por M1, otras partes a 2, controlado por M2. Ahora tenemos dos gestores de memoria principal coexistentes.

Paginador. Un Paginador puede integrarse con un gestor de memoria o usar un servidor de gestión de memoria. Los paginadores utilizan las primitivas de concesión, mapeo y limpieza del micro-kernel. Las interfaces restantes, paginador-cliente, paginador-servidor de memoria y paginador-controlador de dispositivo, se basan completamente en IPC y están definidas a nivel de usuario.

Los paginadores pueden usarse para implementar memoria virtual paginada tradicional y mapeo de archivos/bases de datos en espacios de direcciones de usuario, así como memoria residente no paginada para controladores de dispositivos y/o sistemas en tiempo real. Los paginadores apilados, es decir, múltiples capas de paginadores, pueden usarse para combinar control de acceso con paginadores existentes o para combinar varios paginadores (por ejemplo, uno por disco) en un objeto compuesto. Las estrategias de paginación suministradas por el usuario son manejadas a nivel de usuario y no están restringidas de ninguna manera por el micro-kernel. Los sistemas de archivos apilados pueden realizarse de manera similar.

Asignación de Recursos Multimedia. Las aplicaciones multimedia y otras aplicaciones en tiempo real requieren que los recursos de memoria se asignen de manera que permitan tiempos de ejecución predecibles. Los gestores de memoria y paginadores a nivel de usuario mencionados anteriormente permiten, por ejemplo, la asignación fija de memoria física para datos específicos o el bloqueo de datos en memoria por un tiempo dado.

Nota que los asignadores de recursos para multimedia y para compartir el tiempo pueden coexistir. La gestión de conflictos de asignación es parte del trabajo de los servidores.

Controlador de Dispositivos. Un controlador de dispositivos es un proceso que accede directamente a puertos de E/S de hardware mapeados en su espacio de direcciones y recibe

mensajes del hardware (interrupciones) a través del mecanismo IPC estándar. La memoria específica del dispositivo, por ejemplo, una pantalla, se maneja mediante gestores de memoria apropiados. En comparación con otros procesos a nivel de usuario, no hay nada especial en un controlador de dispositivos. No es necesario integrar ningún controlador de dispositivos en el micro-kernel.

Caché de Segundo Nivel y TLB. Mejorar las tasas de aciertos de una caché secundaria mediante la asignación o realojo de páginas puede ser implementado por medio de un paginador que aplica alguna política dependiente de la caché (esperemos que reduzca los conflictos) al asignar páginas virtuales en memoria física. En teoría, incluso un manejador de TLB de software podría implementarse de esta manera. En la práctica, el manejador de TLB de primer nivel será implementado en el hardware o en el micro-kernel. Sin embargo, un manejador de TLB de segundo nivel, por ejemplo, que maneje fallos de una tabla de páginas hasheada, podría implementarse como un servidor a nivel de usuario.

Comunicación Remota. El IPC remoto es implementado por servidores de comunicación que traducen mensajes locales a protocolos de comunicación externos y viceversa. El hardware de comunicación es accedido por controladores de dispositivos. Si se requiere compartir especial de búferes de comunicación y espacio de direcciones del usuario, el servidor de comunicación también actuará como un paginador especial para el cliente. El micro-kernel no está involucrado.

Servidor Unix. Las llamadas al sistema Unix se implementan por IPC. El servidor Unix puede actuar como un paginador para sus clientes y también usar compartir memoria para comunicarse con sus clientes. El servidor Unix mismo puede ser paginable o residente.

Conclusión. Un pequeño conjunto de conceptos del micro-kernel conduce a abstracciones que enfatizan la flexibilidad, siempre que se desempeñen lo suficientemente bien. Lo único que no puede implementarse sobre estas abstracciones es la arquitectura del procesador, los registros, las cachés de primer nivel y los TLB de primer nivel.

4 Rendimiento, Hechos y Rumores

4.1 Sobrecarga de Cambios

Es ampliamente creído que cambiar entre el modo kernel y el modo usuario, entre espacios de direcciones y entre hilos es inherentemente costoso. Algunas mediciones parecen apoyar esta creencia.

4.1.1 Cambios de kernel-Usuario

Ousterhout (1990) midió los costos de ejecutar la llamada al kernel "nula" `getpid`. Dado que la operación real de `getpid` consiste solo en algunas cargas y almacenamientos, este método mide los costos básicos de una llamada al kernel. Normalizado a una máquina hipotética con una calificación de 10 MIPS (10 VAX 11/780 o aproximadamente un 486 a 50 MHz), mostró que la mayoría de las máquinas necesitan entre 20 y 30 µs por `getpid`, y una incluso requirió 63 µs. Corroborando estos resultados, medimos 18 µs por llamada al micro-kernel de Mach `get_self_thread`. De hecho, los costos medidos de la llamada al kernel son altos.

Para analizar los costos medidos, nuestro argumento se basa en un procesador 486 (50 MHz). Tomamos un procesador x86, porque los cambios de modo kernel-usuario son extremadamente costosos en estos procesadores. A diferencia del procesador de peor caso, usamos una medición de mejor caso para la discusión, 18 µs para Mach en un 486/50.

Los costos medidos por llamada al kernel son de 18 * 50 = 900 ciclos. La instrucción de máquina básica para entrar en modo kernel cuesta 71 ciclos, seguido por 36 ciclos adicionales para volver al modo usuario. Estas dos instrucciones cambian entre la pila del usuario y del kernel y empujan/sacan un registro de acumulación y un puntero de instrucción. Por lo tanto, 107 ciclos (aproximadamente 2 µs) es un límite inferior en los cambios de modo kernel-usuario. Los

800 ciclos restantes o más son pura sobrecarga del kernel. Con este término, denotamos todos los ciclos que son únicamente debido a la construcción del kernel, sin importar si se gastan en ejecutar instrucciones (800 ciclos equivalen a 500 instrucciones) o en fallos de caché y TLB (800 ciclos equivalen a 270 fallos de caché primaria y 90 fallos de TLB). Tenemos que concluir que los kernels medidos hacen mucho trabajo al entrar y salir del kernel. Nota que este trabajo por definición no tiene ningún efecto neto.

¿Es realmente necesario una sobrecarga del kernel de 800 ciclos? La respuesta es no. Prueba empírica: L3 Liedtke (1993) tiene una sobrecarga mínima del kernel de 15 ciclos. Si la llamada al micro-kernel se ejecuta con poca frecuencia, puede aumentar hasta 57 ciclos adicionales (3 fallos de TLB, 10 fallos de caché). Los costos completos de la llamada al kernel L3 son, por lo tanto, de 123 a 180 ciclos, en su mayoría menos de 3 μs.

El micro-kernel L3 está orientado a procesos, usa una pila de kernel por hilo y soporta procesos de usuario persistentes (es decir, el kernel puede ser intercambiado sin afectar al sistema restante, incluso si un proceso reside actualmente en modo kernel). Por lo tanto, debería ser posible para cualquier otro micro-kernel lograr una sobrecarga de llamada al kernel comparativamente baja en el mismo hardware.

Otros procesadores pueden requerir una sobrecarga ligeramente mayor, pero ofrecen operaciones básicas sustancialmente más baratas para entrar y salir del modo kernel. Desde un punto de vista arquitectónico, llamar al kernel desde el modo usuario es simplemente una llamada indirecta, complementada por un cambio de pila y el establecimiento del bit interno 'kernel' para permitir operaciones privilegiadas. En consecuencia, volver del modo kernel es una operación de retorno normal complementada por volver a la pila del usuario y restablecer el bit 'kernel'. Si el procesador tiene registros de puntero de pila diferentes para la pila del usuario y del kernel, los costos de cambio de pila pueden ocultarse. Conceptualmente, entrar y salir del modo kernel puede funcionar exactamente como una instrucción de llamada y retorno indirecto normal (que no dependen de la predicción de ramas). Idealmente, esto significa 2+2=4 ciclos en un procesador de 1 emisión.

Conclusión. Comparado con el mínimo teórico, los cambios de modo kernel-usuario son costosos en algunos procesadores. Comparado con los kernels existentes, sin embargo, pueden mejorarse de 6 a 10 veces mediante una construcción adecuada del micro-kernel. Los cambios de modo kernel-usuario no son un problema conceptual serio, sino uno de implementación.

4.1.2 Cambios de Espacio de Direcciones

También se considera comúnmente que los cambios de espacio de direcciones son costosos. Todas las mediciones conocidas por el autor relacionadas con este tema tratan sobre los costos combinados de cambio de hilos y de espacio de direcciones. Por lo tanto, en esta sección, analizamos solo los costos arquitectónicos del procesador para los cambios puros de espacio de direcciones. Las mediciones combinadas se discuten junto con los cambios de hilos.

La mayoría de los procesadores modernos utilizan una caché primaria indexada físicamente que no se ve afectada por los cambios de espacio de direcciones. Cambiar la tabla de páginas suele ser muy barato: de 1 a 10 ciclos. Los costos reales están determinados por la arquitectura de la TLB.

Algunos procesadores (por ejemplo, MIPS R4000) utilizan TLBs etiquetadas, donde cada entrada no solo contiene la dirección de la página virtual sino también el identificador del espacio de direcciones. Por lo tanto, cambiar el espacio de direcciones es transparente para la TLB y no cuesta ciclos adicionales. Sin embargo, el cambio de espacio de direcciones puede inducir costos indirectos, ya que las páginas compartidas ocupan una entrada de TLB por espacio de direcciones. Suponiendo que el micro-kernel (compartido por todos los espacios de direcciones) tiene un conjunto de trabajo pequeño y que hay suficientes entradas de TLB, el problema no debería ser grave. Sin embargo, no podemos apoyar esto empíricamente, ya que no conocemos un micro-kernel adecuado que funcione en tal procesador.

La mayoría de los procesadores actuales (por ejemplo, 486, Pentium, PowerPC y Alpha) incluyen TLBs no etiquetadas. Un cambio de espacio de direcciones, por lo tanto, requiere un vaciado de la TLB. Los costos reales están determinados por las operaciones de carga de TLB que son necesarias para restablecer el conjunto de trabajo actual más tarde. Si el conjunto de trabajo consiste en n páginas, la TLB es completamente asociativa, tiene s entradas y un fallo de TLB cuesta m ciclos, en total se requieren como máximo $\min(n, s) \times m$ ciclos.

Aparentemente, las TLBs no etiquetadas más grandes conducen a un problema de rendimiento. Por ejemplo, recargar completamente las TLBs de datos y código del Pentium requiere al menos $(32 + 64) \times 9 = 864$ ciclos. Por lo tanto, interceptar un programa cada 100 µs podría implicar una sobrecarga de hasta el 9%. Aunque usar la TLB completa es poco realista, este cálculo de peor caso muestra que cambiar las tablas de páginas puede volverse crítico en algunas situaciones.

Afortunadamente, esto no es un problema, ya que en Pentium y PowerPC, los cambios de espacio de direcciones pueden manejarse de manera diferente. La arquitectura PowerPC incluye registros de segmento que pueden ser controlados por el micro-kernel y ofrecen una instalación de traducción de direcciones adicional desde el espacio de direcciones local de 2^32 bytes a un espacio global de 2^52 bytes. Si consideramos el espacio global como un conjunto de un millón de espacios locales, los cambios de espacio de direcciones pueden implementarse recargando los

registros de segmento en lugar de cambiar la tabla de páginas. Con 29 ciclos para un cambio de segmento de 3.5 GB o 12 ciclos para un cambio de segmento de 1 GB, la sobrecarga es baja en comparación con un vaciado de la TLB que ya no se requiere. De hecho, tenemos una TLB etiquetada.

Las cosas no son tan fáciles en el Pentium o el 486. Dado que los segmentos se mapean en un espacio de 2^32 bytes, mapear múltiples espacios de direcciones de usuario en un espacio lineal debe manejarse dinámicamente y depende de los tamaños realmente utilizados de los espacios de direcciones de usuario activos. La técnica de implementación correspondiente es transparente para el usuario y elimina el posible cuello de botella de rendimiento. Entonces, la sobrecarga de cambio de espacio de direcciones es de 15 ciclos en el Pentium y 39 ciclos en el 486.

Para entender que la restricción de un espacio global de 2^32 bytes no es crucial para el rendimiento, hay que mencionar que los espacios de direcciones que se utilizan solo durante períodos muy cortos y con conjuntos de trabajo pequeños son efectivamente muy pequeños en la mayoría de los casos, digamos 1 MB o menos para un controlador de dispositivos. Por ejemplo, podemos multiplexar un espacio de direcciones de usuario de 3 GB con 8 espacios de usuario de 64 MB y adicionalmente 128 espacios de usuario de 1 MB. El truco es compartir los espacios más pequeños con todos los grandes espacios de 3 GB. Entonces, cualquier cambio de espacio de direcciones a un espacio mediano o pequeño siempre es rápido. Cambiar entre dos grandes

espacios de direcciones es de todos modos no crítico, ya que cambiar entre dos grandes conjuntos de trabajo implica costos de fallo de TLB y caché, sin importar si los dos programas se ejecutan en el mismo o en diferentes espacios de direcciones.

Conclusión. Los cambios de espacio de direcciones correctamente construidos no son muy costosos, menos de 50 ciclos en procesadores modernos. En un procesador de 100 MHz, los costos heredados de los cambios de espacio de direcciones pueden ignorarse aproximadamente hasta 100,000 cambios por segundo. Optimizaciones especiales, como ejecutar servidores dedicados en el espacio del kernel, son superfluas. El cambio de contexto costoso en algunos micro-kernels existentes se debe a la implementación y no a problemas inherentes al concepto.

4.1.3 Cambios de Hilo e IPC

Ousterhout (1990) también midió el cambio de contexto en algunos sistemas Unix haciendo eco de un byte de ida y vuelta a través de tuberías entre dos procesos. Nuevamente normalizado a una máquina de 10 MIPS, la mayoría de los resultados están entre 400 y 800 µs por ping-pong, y uno fue de 1450 µs. Todos los micro-kernels existentes son al menos 2 veces más rápidos, pero está demostrado por construcción que 10 µs, es decir, un RPC de 40 a 80 veces más rápido, es alcanzable. La tabla 2 proporciona los costos de hacer eco de un byte por un RPC de ida y vuelta, es decir, dos operaciones IPC.

Todos los tiempos son de usuario a usuario, cruzando espacios de direcciones. Incluyen el costo de la llamada al sistema, copia de argumentos, cambio de pila y cambio de espacio de direcciones. Exokernel, Spring y L3 demuestran que la comunicación puede implementarse bastante rápido y que los costos están fuertemente influenciados por la arquitectura del procesador: Spring en Sparc tiene que lidiar con ventanas de registro, mientras que L3 se ve lastrado por el hecho de que una trampa 486 es 100 ciclos más costosa que una trampa Sparc.

El efecto de usar el cambio de espacio de direcciones basado en segmentos en Pentium se muestra en la figura 2. Una aplicación de larga duración con un conjunto de trabajo estable (de 2 a 64 páginas de datos) ejecuta un RPC corto a un servidor con un pequeño conjunto de trabajo (2 páginas). Después del RPC, la aplicación vuelve a acceder a todas sus páginas. La medición se realiza mediante 100,000 repeticiones y comparando cada ejecución contra la ejecución de la aplicación (accediendo 100,000 veces a todas las páginas) sin RPC. Los tiempos dados son tiempos de RPC de ida y vuelta, de usuario a usuario, más el tiempo requerido para restablecer el conjunto de trabajo de la aplicación.

Conclusión. El IPC puede implementarse lo suficientemente rápido como para manejar también las interrupciones de hardware mediante este mecanismo.

4.2 Efectos de la Memoria

Chen y Bershad (1993) compararon el comportamiento del sistema de memoria de Ultrix, un gran sistema Unix monolítico, con el del micro-kernel Mach complementado con un servidor Unix. Midieron la sobrecarga de ciclos de memoria por instrucción (MCPI) y encontraron que los programas que se ejecutan bajo Mach + servidor Unix tenían un MCPI sustancialmente más alto que los mismos programas bajo Ultrix. Para algunos programas, las diferencias eran de hasta 0.25 ciclos por instrucción, promediados sobre el programa completo (usuario + sistema). Una degradación similar del sistema de memoria de Mach versus Ultrix es notada por otros [Nagle et al. 1994]. El punto crucial es si este problema se debe a la forma en que está construido Mach, o si es causado por el enfoque del micro-kernel.

Chen y Bershad (1993, p. 125) afirman: "Esto sugiere que las optimizaciones del micro-kernel que se centran exclusivamente en el IPC, sin considerar otras fuentes de sobrecarga del sistema como MCPI, tendrán un impacto limitado en el rendimiento general del sistema." Aunque uno podría suponer un impacto principal de la arquitectura del sistema operativo, el artículo mencionado presenta exclusivamente hechos "tal como son" sobre una implementación específica sin analizar las razones de la degradación del sistema de memoria.

Por lo tanto, se requiere un análisis cuidadoso de los resultados. Según el artículo original, comprendemos bajo 'sistema' ya sea todas las actividades de Ultrix o las actividades combinadas del micro-kernel Mach, la biblioteca de emulación Unix y el servidor Unix. El caso de Ultrix está denotado por U, el caso de Mach por M. Restringimos nuestro análisis a las muestras que muestran una diferencia significativa de MCPI para ambos sistemas: sed, egrep, yacc, gcc, compress, espresso y el benchmark andrew ab.

En la figura 3, presentamos los resultados de la figura 2-1 de Chen de una manera ligeramente reordenada. Hemos coloreado en negro los MCPI que se deben a fallos de la caché de instrucciones del sistema o de caché de datos. Las barras blancas comprenden todas las demás causas, paradas del búfer de escritura del sistema, lecturas no cacheadas del sistema, fallos de caché de instrucciones y de datos del usuario y paradas del búfer de escritura del usuario. Es fácil ver que las barras blancas no difieren significativamente entre Ultrix y Mach; la diferencia promedio es 0.00, la desviación estándar es 0.02 MCPI.

Concluimos que las diferencias en el comportamiento del sistema de memoria son esencialmente causadas por un aumento en los fallos de caché del sistema para Mach. Podrían ser fallos de conflicto (el sistema medido usaba cachés mapeadas directamente) o fallos de capacidad. Un gran número de fallos de conflicto sugeriría un problema potencial debido a la estructura del sistema operativo.

Chen y Bershad midieron conflictos de caché comparando el mapeo directo con una caché simulada de 2 vías. Encontraron que la auto-interferencia del sistema es más importante que la interferencia usuario/sistema, pero los datos también muestran que la proporción de fallos de conflicto a fallos de capacidad en Mach es menor que en Ultrix. La tabla 4 muestra los fallos de caché del sistema por conflicto (negro) y capacidad (blanco) tanto en una escala absoluta (izquierda) como en proporción (derecha).

De esto podemos deducir que el aumento de los fallos de caché es causado por un mayor consumo de caché del sistema (Mach + biblioteca de emulación + servidor Unix), no por conflictos inherentes a la estructura del sistema. La próxima tarea es encontrar el componente responsable del mayor consumo de caché. Suponemos que el servidor Unix único utilizado se comporta de manera comparable a la parte correspondiente del kernel de Ultrix. Esto se apoya en el hecho de que las muestras pasaron incluso menos instrucciones en el servidor Unix de Mach que en las rutinas correspondientes de Ultrix. También excluimos la biblioteca de emulación de Mach, ya que Chen y Bershad informan que solo el 3% o menos de la sobrecarga del sistema es causada por ella.

Lo que queda es Mach mismo, incluyendo el manejo de trampas, IPC y gestión de memoria, que por lo tanto debe inducir casi todos los fallos de caché adicionales. Por lo tanto, las mediciones mencionadas sugieren que la degradación del sistema de memoria es causada

únicamente por un alto consumo de caché del micro-kernel. En otras palabras: reducir drásticamente el conjunto de trabajo de caché de un micro-kernel resolverá el problema.

Dado que un micro-kernel es básicamente un conjunto de procedimientos que son invocados por hilos a nivel de usuario o hardware, un alto consumo de caché solo puede explicarse por un gran número de operaciones del micro-kernel muy frecuentemente utilizadas o por grandes conjuntos de trabajo de caché de algunas operaciones frecuentemente utilizadas. Según la sección 2, el primer caso debe considerarse como un error conceptual. Los grandes conjuntos de trabajo de caché tampoco son una característica inherente de los micro-kernels. Por ejemplo, L3 requiere menos de 1 K para IPC corto.

Mogul y Borg (1991) informaron un aumento en los fallos de caché después de cambios de contexto programados de manera preventiva entre aplicaciones con grandes conjuntos de trabajo. Esto depende principalmente de la carga de la aplicación y del requisito de ejecución entrelazada (tiempo compartido). El tipo de kernel es casi irrelevante. Mostramos (sección 4.1.2 y 4.1.3) que los cambios de contexto del micro-kernel no son costosos en el sentido de que no hay mucha diferencia entre ejecutar la aplicación + servidores en uno o en múltiples espacios de direcciones.

Conclusión. La hipótesis de que las arquitecturas de micro-kernel conducen inherentemente a la degradación del sistema de memoria no está fundamentada. Por el contrario, las mediciones citadas apoyan la hipótesis de que los micro-kernels correctamente construidos evitarán automáticamente la degradación del sistema de memoria medida para Mach.

5 No portabilidad

Los micro-kernels antiguos se construyeron de manera independiente de la máquina sobre una pequeña capa dependiente del hardware. Este enfoque tiene grandes ventajas desde el punto de vista tecnológico del software: los programadores no necesitaban saber mucho sobre los procesadores y los micro-kernels resultantes podrían ser fácilmente portados a nuevas máquinas. Desafortunadamente, este enfoque impidió que estos micro-kernels alcanzaran el rendimiento necesario y, por lo tanto, la flexibilidad.

En retrospectiva, no deberíamos sorprendernos, ya que construir un micro-kernel sobre hardware abstracto tiene implicaciones serias:

- Un micro-kernel de este tipo no puede aprovechar hardware específico.

- No puede tomar precauciones para eludir o evitar problemas de rendimiento de hardware específico.

- La capa adicional por sí misma cuesta rendimiento.

Los micro-kernels forman la capa más baja de los sistemas operativos más allá del hardware. Por lo tanto, deberíamos aceptar que son tan dependientes del hardware como los generadores de código de optimización. Hemos aprendido que no solo la codificación, sino incluso los algoritmos utilizados dentro de un micro-kernel y sus conceptos internos son extremadamente dependientes del procesador.

5.1 Procesadores compatibles

Para ilustrar, describiremos brevemente cómo un micro-kernel debe ser conceptualmente modificado incluso cuando se "porta" de un 486 a un Pentium, es decir, a un procesador compatible.

Aunque el procesador Pentium es compatible en binario con el 486, hay algunas diferencias en la arquitectura interna del hardware (ver tabla 3) que influyen en la arquitectura interna del micro-kernel:

- Implementación de espacio de direcciones de usuario. Como se mencionó en la sección 4.1.2, un micro-kernel Pentium debería usar registros de segmento para implementar espacios de direcciones de usuario de modo que cada espacio de direcciones de hardware de 232 bytes comparta todos los espacios de usuario pequeños y uno grande. Recuerde que esto puede implementarse de manera transparente para el usuario.

Ford (1993) propuso una técnica similar para el 486, y la tabla 1 también la sugiere para el 486. Sin embargo, el cambio convencional de espacio de direcciones de hardware es preferible en este procesador. Las cargas costosas de registros de segmento y las instrucciones adicionales en varios lugares del kernel suman aproximadamente 130 ciclos requeridos adicionales. Ahora miremos la situación relevante: un cambio de espacio de direcciones de un espacio grande a uno pequeño y de vuelta al grande. Asumiendo aciertos de caché, los costos del modelo de registro de segmento serían (130 + 39) 2 = 338 ciclos, mientras que el modelo de espacio de direcciones convencional requeriría 28 _ 9 + 36 = 288 ciclos en el caso teórico de uso del 100% de TLB, 14 _ 9 + 36 = 162 ciclos para el caso más probable de que el espacio de direcciones grande use solo el 50% de la TLB y solo 72 ciclos en el mejor caso. En total, el método convencional gana.

En el Pentium, sin embargo, el método de registro de segmento vale la pena. Las razones son varias: (a) Las cargas de registros de segmento son más rápidas. (b) Las instrucciones rápidas son más baratas, mientras que la sobrecarga por trampas y fallos de TLB permanece casi

constante. (c) Los fallos de caché por conflicto (que, en relación con la ejecución de instrucciones, son de todos modos más costosos) son más probables debido a la reducida asociatividad. Evitar fallos de TLB también reduce los conflictos de caché. (d) Debido al TLB tres veces más grande, los costos de vaciado pueden aumentar sustancialmente. Como resultado, en el Pentium, el método de registro de segmento siempre vale la pena (ver figura 2).

Como consecuencia, tenemos que implementar un multiplexor de espacio de direcciones de usuario adicional, tenemos que modificar las rutinas de cambio de espacio de direcciones, el manejo de direcciones suministradas por el usuario, los bloques de control de hilos, los bloques de control de tareas, la implementación de IPC y la estructura de espacio de direcciones tal como la ve el kernel. En total, los cambios mencionados afectan a algoritmos en aproximadamente la mitad de todos los módulos del micro-kernel.

Implementación de IPC. Debido a la reducida asociatividad, las cachés del Pentium tienden a exhibir un aumento en fallos por conflicto. Una forma simple de mejorar el comportamiento de la caché durante el IPC es reestructurando los datos del bloque de control de hilos de manera que se beneficie del tamaño duplicado de línea de caché. Esto puede adaptarse al kernel del 486, ya que no tiene efecto en el 486 y puede implementarse de manera transparente para el usuario.

En el kernel del 486, los bloques de control de hilos (incluidos los stacks del kernel) estaban alineados con la página. El IPC siempre accede a 2 bloques de control y stacks del kernel simultáneamente. El hardware de caché mapea los datos correspondientes de ambos bloques de control a direcciones de caché idénticas. Debido a su asociatividad de 4 vías, este problema podría ignorarse en el 486. Sin embargo, la caché de datos del Pentium es solo asociativa de 2 vías. Una buena optimización es alinear los bloques de control de hilos no más en 4K sino en límites de 1K. (1K es el límite inferior por razones internas.) Entonces hay un 75% de probabilidad de que dos bloques de control seleccionados al azar no compitan en la caché.

Sorprendentemente, esto afecta la estructura interna de bits de los identificadores únicos de hilos suministrados por el micro-kernel (ver Liedtke 1993 para detalles). Por lo tanto, el nuevo kernel no puede simplemente reemplazar al antiguo, ya que los programas de usuario (persistentes) ya poseen uids que se volverían inválidos.

5.2 Procesadores Incompatibles

Los procesadores de familias competidoras difieren en conjunto de instrucciones, arquitectura de registros, manejo de excepciones, arquitectura de caché/TLB, protección y modelo de memoria. Especialmente estos últimos influyen radicalmente en la estructura del micro-kernel. Hay sistemas con:

- Tablas de páginas multinivel,

- Tablas de páginas hasheadas,

- (sin) bits de referencia,

- (sin) protección de página, protección de página extraña,

- Tamaños de página únicos/múltiples,

- Espacios de direcciones de 2^32, 2^43, 2^52 y 2^64 bytes,

- Espacios de direcciones planos y segmentados,

- Varios modelos de segmentos,

- TLBs etiquetadas/no etiquetadas,

- Cachés etiquetadas virtualmente/físicamente.

Las diferencias son órdenes de magnitud mayores que entre el 486 y el Pentium. Debemos esperar que un nuevo procesador requiera un nuevo diseño de micro-kernel.

Para ilustrar, comparamos dos kernels diferentes en dos procesadores diferentes: el Exokernel [Engler et al. 1995] funcionando en un R2000 y L3 funcionando en un 486. Aunque esto es similar a comparar manzanas con naranjas, un análisis cuidadoso de las diferencias de rendimiento ayuda a comprender los factores determinantes del rendimiento y a ponderar las diferencias en la arquitectura del procesador. Finalmente, esto resulta en diferentes arquitecturas de micro-kernels.

Comparamos la transferencia de control protegida (PCT) de Exokernel con el IPC de L3. Exo-PCT en el R2000 requiere aproximadamente 35 ciclos, mientras que L3 toma 250 ciclos en un procesador 486 para una transferencia de mensajes de 8 bytes. Si esta diferencia no puede ser explicada por diferente funcionalidad y/o rendimiento promedio del procesador, debe haber una anomalía relevante para el diseño del micro-kernel.

Exo-PCT es un "sustrato para implementar mecanismos eficientes de IPC. Cambia el contador de programa a un valor acordado en el llamado, dona el segmento de tiempo actual al entorno del procesador del llamado, e instala los elementos requeridos del contexto del procesador del llamado." El IPC de L3 se utiliza para la comunicación segura entre socios potencialmente no confiables, por lo tanto, verifica adicionalmente el permiso de comunicación (si el socio está dispuesto a recibir un mensaje del emisor y si no se cruza un límite de clan), sincroniza ambos hilos, soporta la recuperación de errores por tiempos de espera de envío y recepción, y permite mensajes complejos para reducir los costos de marshaling y la frecuencia de

IPC. Desde nuestra experiencia, extender Exo-PCT de manera correspondiente debería requerir no más de 30 ciclos adicionales. (Nota que usar PCT para un LRPC de confianza ya cuesta 18 ciclos adicionales, ver tabla 2). Por lo tanto, asumimos que un "Exo-IPC" equivalente a L3 costaría alrededor de 65 ciclos en el R2000. Finalmente, debemos tener en cuenta que los ciclos de ambos procesadores no son equivalentes en cuanto a las instrucciones más frecuentemente ejecutadas se refiere. Basado en SpecInts, aproximadamente 1.4 ciclos de 486 parecen hacer tanto trabajo como un ciclo de R2000, comparando las cinco instrucciones más relevantes en este contexto (2-op-alu, 3-op-alu, carga, bifurcación tomada y no tomada) da 1.6 para código bien optimizado. Así estimamos que el Exo-IPC costaría hasta aproximadamente 100 ciclos de 486, siendo definitivamente menos que los 250 ciclos de L3.

Esta sustancial diferencia en tiempo indica una diferencia aislada entre ambas arquitecturas de procesadores que influye fuertemente en el IPC (y quizás otros mecanismos del micro-kernel), pero no en programas promedio.

De hecho, el procesador 486 impone una alta penalización al entrar/salir del kernel y requiere un vaciado de TLB por IPC debido a su TLB no etiquetada. Esto cuesta al menos 107 + 49 = 156 ciclos. Por otro lado, el R2000 tiene una TLB etiquetada, es decir, evita el vaciado de la TLB, y necesita menos de 20 ciclos para entrar y salir del kernel.

Del ejemplo anterior, aprendemos dos lecciones:

- Para micro-kernels bien diseñados en diferentes arquitecturas de procesador, en particular con diferentes sistemas de memoria, deberíamos esperar diferencias aisladas de tiempo que no están relacionadas con el rendimiento general del procesador.

- Diferentes arquitecturas requieren técnicas de optimización específicas del procesador que incluso afectan la estructura global del micro-kernel.

Para entender el segundo punto, recuerde que el vaciado obligatorio de TLB del 486 requiere minimizar el número de fallos de TLB subsecuentes. Las técnicas relevantes [Liedtke 1993, pp. 179,182-183] están basadas principalmente en una construcción adecuada del espacio de direcciones: concentrando tablas internas del procesador y datos del kernel muy utilizados en una página (no hay memoria no mapeada en el 486), implementando bloques de control y pilas del kernel como objetos virtuales, programación perezosa. En total, estas técnicas ahorran 11 fallos de TLB, es decir, al menos 99 ciclos en el 486 y por lo tanto son inevitables.

Debido a su facilidad de memoria no mapeada y TLB etiquetada, la restricción mencionada desaparece en el R2000. En consecuencia, la estructura interna (estructura del espacio de direcciones, manejo de fallos de página, quizás acceso a bloques de control y programación) de un kernel correspondiente puede diferir sustancialmente de un kernel de 486.

Si otros factores también implican implementar bloques de control como objetos físicos, incluso los identificadores únicos diferirán entre el R2000 (no tamaño de puntero + x) y el kernel de 486 (no tamaño de bloque de control + x).

Conclusión. Los micro-kernels forman el enlace entre un conjunto mínimo de abstracciones y el procesador desnudo. Las demandas de rendimiento son comparables a las de la microprogramación anterior. Como consecuencia, los micro-kernels no son inherentemente portables. En su lugar, son la base dependiente del procesador para sistemas operativos portables.

6 Síntesis, Spin, DP-Mach, Panda, Cache y Exokernel

Síntesis. El sistema operativo Synthesis de Henry Massalin [Pu et al. 1988] es otro ejemplo de un kernel de alto rendimiento (y no portátil). Su característica distintiva era un "compilador" integrado en el kernel que generaba código del kernel en tiempo de ejecución. Por ejemplo, al emitir una llamada al sistema para leer un tubo, el kernel de Synthesis generaba un código especializado para leer de este tubo y modificaba la invocación respectiva. Esta técnica fue muy exitosa en el 68030. Sin embargo (un buen ejemplo de la no portabilidad), probablemente ya no sería rentable en procesadores modernos, porque (a) la inflación de código degradaría el rendimiento de la caché y (b) la generación frecuente de pequeños fragmentos de código contaminaría la caché de instrucciones.

Spin. Spin [Bershad et al. 1994, Bershad et al. 1995] es un desarrollo nuevo que intenta extender la idea de Synthesis: los algoritmos suministrados por el usuario son traducidos por un compilador del kernel y añadidos al kernel, es decir, el usuario puede escribir nuevas llamadas al sistema. Controlando las ramificaciones y las referencias de memoria, el compilador asegura que el código recién generado no viole la integridad del kernel o del usuario. Este enfoque reduce los cambios de modo kernel-usuario y, a veces, los cambios de espacio de direcciones. Spin se basa en Mach y por lo tanto puede heredar muchas de sus ineficiencias, lo que dificulta evaluar los resultados de rendimiento. Es necesario reescalarlos a un micro-kernel eficiente con cambios rápidos de modo kernel-usuario y IPC rápido. Sin embargo, el problema más crucial es la estimación de cómo una arquitectura de micro-kernel optimizada y los requisitos de un compilador de kernel interfieren entre sí.

Utah-Mach. Ford y Lepreau [1994] modificaron la semántica IPC de Mach para migrar RPC basado en la migración de hilos entre espacios de direcciones, similar al modelo de Clouds [Bernabeu-Auban et al. 1988]. Se logró una ganancia de rendimiento sustancial, un factor de 3 a 4.

DP-Mach. DP-Mach [Bryce y Muller 1995] implementa múltiples dominios de protección dentro de un espacio de direcciones de usuario y ofrece una llamada interdominio protegida. Los resultados de rendimiento (ver tabla 2) son alentadores. Sin embargo, aunque esta

llamada interdominio es altamente especializada, es el doble de lenta que lo que podría lograrse con un mecanismo RPC general. De hecho, una llamada interdominio necesita dos llamadas al kernel y dos cambios de espacio de direcciones. Un RPC general requiere dos cambios de hilo adicionales y transferencias de argumentos. Aparentemente, los costos de la llamada al kernel y el cambio de espacio de direcciones dominan.

Panda. El micro-kernel de Panda [Assenmacher et al. 1993] es otro ejemplo de un kernel pequeño que delega tanto como sea posible al espacio de usuario. Además de sus dos conceptos básicos, dominio de protección y procesador virtual, el kernel de Panda solo maneja interrupciones y excepciones.

Cache-Kernel. El Cache-kernel [Cheriton y Duda 1994] también es un micro-kernel pequeño y dependiente del hardware. A diferencia del Exokernel, se basa en una máquina virtual pequeña y fija (no extensible). Caché de kernels, hilos, espacios de direcciones y mapeos. El término "caché" se refiere al hecho de que el micro-kernel nunca maneja el conjunto completo de, por ejemplo, todos los espacios de direcciones, sino solo un subconjunto seleccionado dinámicamente. Se esperaba que esta técnica condujera a una interfaz de micro-kernel más pequeña y también a menos código de micro-kernel, ya que ya no tiene que lidiar con casos especiales pero infrecuentes.

Exokernel. A diferencia de Spin, el Exokernel [Engler et al. 1994, Engler et al. 1995] es un micro-kernel pequeño y dependiente del hardware. De acuerdo con nuestra tesis de dependencia del procesador, el exokernel está adaptado al R2000 y obtiene excelentes valores de rendimiento para sus primitivas. A diferencia de nuestro enfoque, se basa en la filosofía de que un kernel no debe proporcionar abstracciones sino solo un conjunto mínimo de primitivas. Consecuentemente, la interfaz de Exokernel es dependiente de la arquitectura, en particular dedicada a TLBs controladas por software. Una diferencia adicional con nuestro enfoque de micro-kernel sin controladores es que Exokernel parece integrar parcialmente controladores de dispositivos, en particular para discos, redes y búferes de cuadros.

Creemos que abandonar el enfoque abstracto solo podría justificarse por ganancias sustanciales de rendimiento. Si estas se pueden lograr permanece abierto (ver discusión en la sección 5.2) hasta que tengamos micro-kernels exo y abstractos bien diseñados en la misma plataforma de hardware. Podría resultar entonces que las abstracciones correctas son incluso más eficientes que la multiplexación segura de primitivas de hardware o, por otro lado, que las abstracciones son demasiado inflexibles. Deberíamos tratar de decidir estas preguntas construyendo micro-kernels comparables en al menos dos plataformas de referencia. Tal co-construcción probablemente también llevará a nuevos conocimientos para ambos enfoques.

7 Conclusiones

Un micro-kernel puede proporcionar a las capas superiores un conjunto mínimo de abstracciones apropiadas que son lo suficientemente flexibles para permitir la implementación de sistemas operativos arbitrarios y permitir la explotación de una amplia gama de hardware. Los mecanismos presentados (espacio de direcciones con operaciones de mapeo, limpieza y concesión, hilos con IPC e identificadores únicos) forman tal base. Los sistemas de seguridad multinivel pueden necesitar adicionalmente clanes o un concepto similar de monitor de referencia. Elegir las abstracciones correctas es crucial tanto para la flexibilidad como para el rendimiento. Algunos micro-kernels existentes eligieron abstracciones inapropiadas, o demasiadas, o demasiado especializadas e inflexibles.

Similar a los generadores de código de optimización, los micro-kernels deben construirse por procesador y no son inherentemente portátiles. Las decisiones básicas de implementación, la mayoría de los algoritmos y las estructuras de datos dentro de un micro-kernel son dependientes del procesador. Su diseño debe ser guiado por la predicción y análisis del rendimiento. Además de abstracciones básicas inapropiadas, los errores más frecuentes provienen de una comprensión insuficiente del sistema combinado hardware-software o de una implementación ineficiente. El diseño presentado muestra que es posible lograr micro-kernels de buen rendimiento a través de implementaciones específicas del procesador de abstracciones independientes del procesador.

Disponibilidad

El código fuente del micro-kernel L4, sucesor del micro-kernel L3, está disponible para examen y experimentación a través de la web:

Agradecimientos

Muchas gracias a Hermann H!artig por la discusión y a Rich Uhlig por la corrección de pruebas y ayuda estilística. Más agradecimientos por comentarios de revisión a Dejan Milojicic, algunos árbitros anónimos y a Sacha Krakowiak por su guía.

A Espacios de Direcciones

Un Modelo Abstracto de Espacios de Direcciones

Describimos los espacios de direcciones como mapeos. 0 : V -> R U {ø} es el espacio de direcciones inicial, donde V es el conjunto de páginas virtuales, R el conjunto de páginas físicas (reales) disponibles y ø la página nula que no se puede acceder. Otros espacios de direcciones se definen recursivamente como mapeos: : V -> (& V) U {ø}, donde & es el conjunto de espacios de direcciones. Es conveniente considerar cada mapeo como una tabla de una columna que contiene (v) para todo v ∈ V y puede ser indexada por v. Denotamos los elementos de esta tabla por v.

Todas las modificaciones de espacios de direcciones se basan en la operación de reemplazo: escribimos v -> x para describir un cambio de en v, precisamente:

$$\text{flush } (\sigma, v) \quad ; \quad \sigma_v := x \quad .$$

Una página potencialmente mapeada en v en se limpia, y el nuevo valor x se copia en v. Esta operación es interna al micro-kernel. La usamos solo para describir las tres operaciones exportadas.

Un subsistema S con espacio de direcciones puede conceder cualquiera de sus páginas v a un subsistema S0 con espacio de direcciones 0 siempre que S0 esté de acuerdo:

$$\sigma'_{v'} \leftarrow \sigma_v \quad , \quad \sigma_v \leftarrow \phi$$

Nótese que S determina qué páginas suyas deben ser concedidas, mientras que S0 determina en qué dirección virtual debe mapearse la página concedida en 0. La página concedida se transfiere a 0 y se elimina de .

Un subsistema S con espacio de direcciones puede mapear cualquiera de sus páginas v a un subsistema S0 con espacio de direcciones 0 siempre que S0 esté de acuerdo:

$$\sigma'_{v'} \leftarrow (\sigma, v) \quad .$$

A diferencia de conceder, la página mapeada permanece en el espacio del mapeador y un enlace a la página en el espacio de direcciones del mapeador (v) se almacena en el espacio de direcciones receptor 0, en lugar de transferir el enlace existente de v a v0. Esta operación permite construir espacios de direcciones recursivamente, es decir, nuevos espacios basados en los existentes.

El vaciado, la operación inversa, puede ejecutarse sin el acuerdo explícito de los mapeados, ya que estos acordaron implícitamente al aceptar la operación de mapeo anterior. S puede limpiar cualquiera de sus páginas:

$$\forall_{\sigma'_{v'} = (\sigma, v)} : \sigma'_{v'} \leftarrow \phi$$

Nótese que map y flush se definen recursivamente. El vaciado recursivo también afecta todos los mapeos que se derivan indirectamente de v.

No se pueden establecer ciclos por estas tres operaciones, ya que flush limpia el destino antes de copiar.

Implementando el Modelo

A primera vista, derivar la dirección física de la página v en el espacio de direcciones parece ser bastante complicado y costoso:

$$\sigma(v) = \begin{cases} \sigma'(v') & \text{if } \sigma_v = (\sigma', v') \\ r & \text{if } \sigma_v = r \\ \phi & \text{if } \sigma_v = \phi \end{cases}$$

Afortunadamente, una evaluación recursiva de (v) nunca es requerida. Las tres operaciones básicas garantizan que la dirección física de una página virtual nunca cambie, excepto por el vaciado. Para la implementación, por lo tanto, complementamos cada con una tabla adicional P, donde Pv corresponde a v y contiene ya sea la dirección física de v o ø. El mapeo y la concesión entonces incluyen

$$P'_{v'} := P_v$$

y cada reemplazo $\sigma_v \leftarrow \phi$ invocado por una operación de vaciado incluye

$$P_v := \phi \quad .$$

Pv siempre puede usarse en lugar de evaluar (v). De hecho, P es equivalente a una tabla de páginas de hardware. Los espacios de direcciones del micro-kernel pueden implementarse directamente mediante las facilidades de traducción de direcciones de hardware.

La implementación recomendada de es usar un árbol de mapeo por marco de página física que describa todos los mapeos actuales del marco. Cada nodo contiene (Pv), donde v es la página virtual correspondiente en el espacio de direcciones que está implementado por la tabla de páginas P.

Supongamos que una operación de concesión, mapeo o vaciado trata con una página v en el espacio de direcciones al que está asociada la tabla de páginas P. En un primer paso, la operación selecciona el árbol correspondiente por Pv, la página física. En el siguiente paso, selecciona el nodo del árbol que contiene (Pv). (Esta selección puede hacerse analizando el árbol o en un solo paso, si Pv se extiende con un enlace al nodo). Conceder simplemente reemplaza los valores almacenados en el nodo y el mapeo crea un nuevo nodo hijo para almacenar (P0 v0). El vaciado deja el nodo seleccionado sin efecto pero analiza y borra el subárbol completo, donde Pv0 := se ejecuta para cada nodo (P0 v0) en el subárbol.

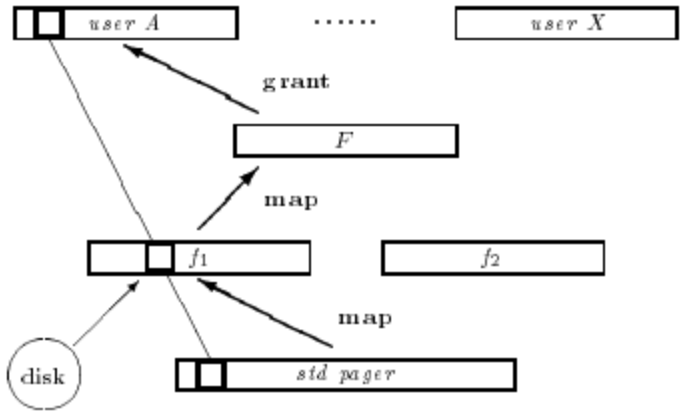

Figure 1: *A Granting Example.*

```
driver thread:
    do
        wait for (msg, sender) ;
        if sender = my hardware interrupt
            then  read/write io ports ;
                  reset hardware interrupt
            else  ...
        fi
    od .
```

	TLB entries	TLB miss cycles	Page Table switch cycles	Segment
486	32	9...13	36...364	39
Pentium	96	9...13	36...1196	15
PowerPC 601	256	?	?	29
Alpha 21064	40	20...50[a]	80...1800	n/a
Mips R4000	48	20...50[a]	0[b]	n/a

[a] Alpha and Mips TLB misses are handled by software.
[b] R4000 has a tagged TLB.

Table 1: *Address Space Switch Overhead*

System	CPU, MHz	RPC time (round trip)	cycles/IPC (oneway)
full IPC semantics			
L3	486, 50	10 μs	250
QNX	486, 33	76 μs	1254
Mach	R2000, 16.7	190 μs	1584
SRC RPC	CVAX, 12.5	464 μs	2900
Mach	486, 50	230 μs	5750
Amoeba	68020, 15	800 μs	6000
Spin	Alpha 21064, 133	102 μs	6783
Mach	Alpha 21064, 133	104 μs	6916
restricted IPC semantics			
Exo-tlrpc	R2000, 16.7	6 μs	53
Spring	SparcV8, 40	11 μs	220
DP-Mach	486, 66	16 μs	528
LRPC	CVAX, 12.5	157 μs	981

Table 2: *1-byte-RPC performance*

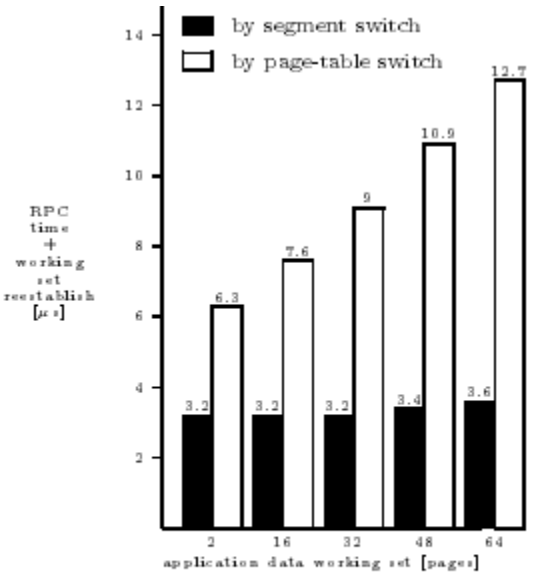

Figure 2: *Segmented Versus Standard Address-Space Switch in L4 on Pentium, 90 MHz.*

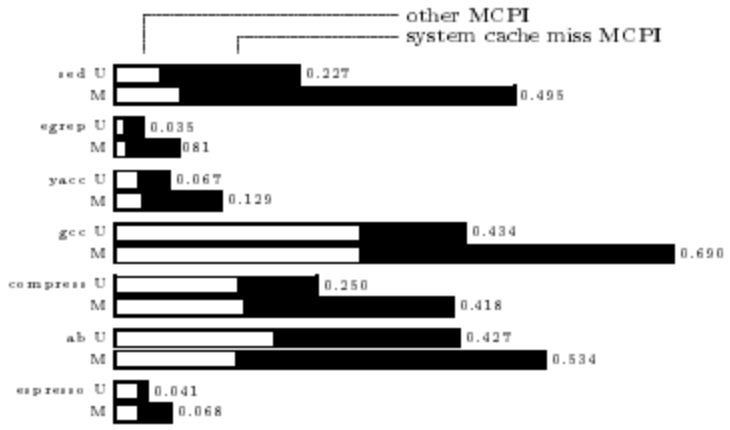

Figure 3: *Baseline MCPI for Ultrix and Mach.*

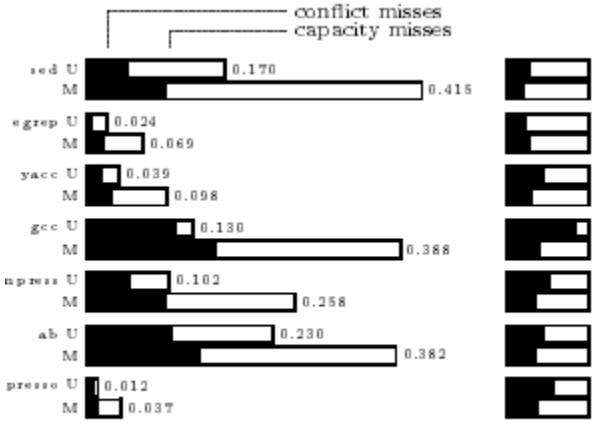

Figure 4: *MCPI Caused by Cache Misses.*

	486	Pentium
TLB entries, ways	$32_{(u)}$ 4×	$32_{(i)} + 64_{(d)}$ 4×
Cache size, ways	$8K_{(u)}$ 4×	$8K_{(i)} + 8K_{(d)}$ 2×
line, write	16B through	32B back
fast instructions	1 cycle	0.5–1 cycle
segment register	9 cycles	3 cycles
trap	107 cycles	69 cycles

Table 3: *486 / Pentium Differences*