On μ−Kernel Construction

CARLOS MARTÍN FLORES GONZÁLEZ, Carné: 2015183528

Instituto Tecnológico de Costa Rica
Maestría en Computación
Sistemas Operativos Avanzados
Profesor: Francisco Torres Rojas, Ph.D

Los sistemas basados en μ−kernel han sido construidos desde antes que el término fuera introducido. Tradicionalmente, la palabra "kernel" se usa para denotar la parte el sistema operativo es necesaria y común para todo el otro software. La idea básica del enfoque μ−kernel es el de minimizar esta parte, por ejemplo, implementar fuera del kernel lo que sea posible.

Aunque se ha invertido mucho esfuerzo en la construcción de μ−kernel, el enfoque no ha sido aún aceptado de forma general. Esto debido al hecho de que la mayoría de μ−kernels no tiene un rendimiento suficientemente bueno. La ausencia de eficiencia también reestringue la flexibilidad, dado que mecanismos y princpios importantes no se pueden poner en práctica debido al pobre rendimiento. Se cree que el modo de *user-kernel* aumentado y los cambios en el espacio de direcciones con los responsables.

Ventajas del enfoque μ−kernel

- Una clara interfase μ−kernel impone una estructura de sistemas modular.
- Servidores pueden usar los mecanismos provistos por μ-kernel tal y como si fuera otro programa de usuario.
- El sistema es más flexible y personalizable. Diferentes estrategias y APIs, implementadas por diferentes servidores, pueden conexistir en el sistema.

Some μ−Kernel Concepts. Un criterio puede es tolerado dentro de μ−kernel solo si al moverlo fuera del kernel, permitiendo que las implementaciones compitan, evitaría la implementación de una funcionalidad requerida por el sistema. Un requerimiento inevitable para tales sistemas es que un programador debe ser capaz de implementar un sub-sistema S de tal forma que no pueda ser perturbado o corrompido por otros sub-sistemas S'. Principio de independencia: S puede garantizar independencia de S'. El segundo requerimiento es que otros sub-sistemas puedan confiar de estar garantías → Principio de integridad: tiene que haber una forma para que S_1 se puede dirigir a S_2 y establecer un canal de comunicación que no pueda ser corrompido o interferido por S'. **Address Spaces:** a nivel del *hardware* un espacio de direcciones es un mapeo que asocia cada página virtual con un marco de página física o bien la marca como "no accesible". Los mapeos son implementados por *hardware* TLB y tablas de páginas. μ−kernel tiene que ocultar el concepto de *hardware* de espacio de direcciones sino de otro forma , la implementación de protección seria imposible. Se debe permitir la implementación de esquemas arbitrarios de protección (y de no-protección) por encima de μ−kernel. Debería ser simple y similar al concepto de *hardware*. La idea básica es la de brindar construcciones recursivas de espacios de direcciones por fuera del kernel. Existe un espacio direcciones σ_0 que representa la memoria físicay que es controlada por el primer sub-sistema S_0. Cuando el sistema inicia todos los otros espacios de memoria están vacíos. Para construir y mantener más espacios de memoria por encima de σ_0, μ−kernel provee estas tres operaciones: (1) *Grant*: el dueño de un espacio

de memoria puede otorgar (*grant*) una de sus páginas a otro espacio, siempre y cuando el destinatario acepte. La página se remueve del espacio de direcciones del que otorga[1] y se le asigna al destinatario. (2) *Map*: El dueño de un espacio de direcciones puede mapear cualquiera de sus páginas[2] dentro de otro espacio de direcciones, si el destinatario acepta. Al final, la página puede ser accedida en ambos espacios de direcciones. (3) *Flush*: el dueño de un espacio de memoria puede liberar (*flush*) cualquiera de sus páginas. La página que se libera se mantiene accesible in el espacio de direcciones de quién la liberó pero es removida de todo los otros espacios de direcciones que hayan recibidio directa o indirectamente la página de quién la liberó (el *flusher*). I/O: el control de los derechos de acceso de I/O y controladores de dispositivos se lleva a cabo por gestionadores de memoria y *pagers* por encima de μ−kernel. **Threads and IPC:** Un hilo es una actividad que se ejecuta dentro un espacio de direcciones. Un hilo τ se caracteriza por un conjunto de registros, un puntero de pila e información de estado. El estado del hilo también incluye el espacio de memoria $\sigma^{(\tau)}$ en donde τ se está ejecutando. Esta asociación dinámica o estática a un espacio de direcciones es la razón decisiva para incluir el concepto de hilos en μ−kernel. Para prevenir espacios de direcciones corrompidos, todos los cambios en el espacio de direcciones de un hilo debe ser controladas por el kernel. Esto implica que μ−kernel incluye la noción de algún τ el cual representa la actividad mencionada anteriormente. La comunicación entre espacios de direcciones (IPC) debe de ser soportada por μ−kernel. El método clásico es transferir mensajes entre hilos por medio de μ−kernel. Interrupciones: abstracción: mensajes IPC. El *hardware* es considerado como un conjunto de hilos que tienen *ids* especiales y envían mensajes vacíos hacia hilos de *software* asociados. Transformar interrupciones a mensajes lo realiza el kernel pero μ−kernel no se especializa en interrupciones específicas de dispositivos. **Unique Identifiers**: μ−kernel provee identificadores únicos para hilos, tareas o canales de comunicación. *Uids* son requeridos para una comunicación local confiable y eficiente.

Flexibility. Algunas aplicaciones que típicamente pertenecen al SO básico pero que pueden ser implementas por encima de μ−kernel son: Gestores de memoria, *pagers*, asignación de recursos multimedia, controladores de dispositivos, cache de segundo nivel y TLB, comunicaciones remotas, servidores Unix. Lo único que no se puede implementar por encima de estas abstracciones es la arquitectura del procesador, registros, cache de primer nivel y TLBs de primer nivel.

Performance, Facts & Rumors. *Kernel-user switches:* comparado con el mínimo teórico, *user-kernel mode switches* son costoso en algunos procesadores. Por otro lado, en comparación con otros kernels existentes se pueden mejorar de 6 a 10 veces por medio de una construcción apropiada de μ−kernel. *Kernerl-user mode switches* no son un problema conceptual serio pero si uno de implementación. *Address Space Switches:* si se construyen adecuadamente no soy muy costosos, menos de 50 ciclos en procesadores modernos. En un procesador de 100 MHz, el costo heredado de *switches* de espacios de direcciones puede ser ignorado aproximandamente hasta 100,000 *switches* por segundo. Optimizaciones especiales, como ejecución de servidores dedicados en espacios de kernel, son superfluos. Este contexto costoso de *switching* en algunos μ−kernels existentes se debe a la implementación y no es causado por problemas inherentes con el concepto. *Thread Switches and IPC:* IPC puede ser implementado lo suficientemente rápido para manejar también interrupciones de *hardware* por medio de este mecanismo. **Memory Effects:** la hipótesis que la arquitectura μ−kernel inherentemente conduce a degradación de la memoria del sistema no está justificado. Por el contrario, las mediciones mencionadas apoyan la hipótesis que μ−kernels construidos apropiadamente van a evitar automáticamente la degradación de la memoria del sistema en Mach.[3]

[1] Restricción: Solamente puede otorgar páginas a las cuales tiene acceso
[2] Restricción: solo puede mapear páginas a las que tenga acceso.
[3] Implementación de un μ−kernel.

Non-Portability. μ–kernel antiguos fueron construidos indpendientes de la máquina por encima de una pequeña capa dependiente del *hardware*. Esto ayudaba a que los programadores no tuvieran que saber mucho acerca de procesadores. Desafortunadamente este enfoque evitaba a estos μ–kernels lograr el rendimiento y flexibilidad necesaria. Construir un μ–kernel por encima de *hardware* abstracto tiene las serias implicaciones:

- No puede tomar ventaja de *hardware* específico.
- No puede tomar precauciones o evitar problemas de rendimiento en *hardware* específico.
- La capa adicional por si misma impacta en el rendimiento.

Más allá del *hardware*, los μ–kernels forman la capa más baja de un SO. Por tal motivo, se acepta que son dependientes del *hardware*. Se ha aprendido de que no solamente el código sino que tambien los algoritmos utilizados dentro de μ–kernel son extremadamente dependientes del procesador. Diferentes arquitecturas requiered optimizaciones específicas de procesador que podrían afectar la estructura global de $\mu - kernel$. Los μ–kernels no son portables.

1. ¿CUÁL ES EL PROBLEMA QUE PLANTEA EL *PAPER*?

Se cree ampliamente que los sistemas μ–kernel son inherentemente ineficientes y que no son lo sufientemente flexibles.

2. ¿POR QUÉ EL PROBLEMA ES INTERESANTE O IMPORTANTE?

Porque contrario a la creencia, se provee evidencia en la cual la ineficiencia y la no flexibilidad de los μ–kernels reales no viene heredada de la idea básica (los conceptos detrás de μ–kernel) sino más bien de sobrecargas al kernel y/o implementaciones inapropiadas.

3. ¿QUÉ OTRAS SOLUCIONES SE HAN INTENTADO PARA RESOLVER ESTE PROBLEMA?

- Synthesis: kernel de alto rendimiento no portable. Su característica distintiva fue un "compilador" integrado al kernel que generaba código de kernel en tiempo de ejecución.
- SPIN: intenta extender la idea de Synthesis: algoritmos proporcionados por el usuario son traducidos por un compilador de kernel y añadidos al kernel, por ejemplo, el usuario podría escribir un nuevo *system call*. Al controlar ramas (*branches*) y referencias de memoria, el compilador se asegura que el código recién generado no va a impactar al kernel o a la integridad del usuario.
- DP-Mach: implementa múltiples dominios de protección dentro dentro de un espacio de direcciones de un usuario y ofrece llamados inter-dominio.
- Panda: es otro ejemplo más de un kernel pequeño que delega a los espacios de usuario tanto como se pueda.
- Cache-Kernel: otro ejemplo de un μ–kernel pequeño y dependiente del *hardware*. En contraste con Exokernel, depende de una pequeña máquina virtual no extendible. *Cachea* hilos, espacios de direcciones y mapeos de kernel.
- Exokernel: es un pequeño μ–kernel dependiente del *hardware*.

4. ¿CUÁL ES LA SOLUCIÓN PROPUESTA POR LOS AUTORES?

Un μ–kernel puede proveer capas más altas con un conjunto mínimo de abstracciones apropiadas que son lo suficientemente flexibles para permitir la implementación de SO arbitrarios y permitir la explotación de un amplio rango de *hardware*. Los mecanismos presentados (operaciones de espacio de direcciones con *map, flush* y *grant*, hilos con IPC e identificadores únicos) forman esa base. Seguridad de sistemas multi-nivel podria necesitar adicionalmente *clans* o un concepto de referencia similar para monitoreo. Escoger la abstracción apropiada es crucial para la flexibilidad y el rendimiento.

Decisiones de implementación básicas, la mayoría de los algoritmos y las estructuras de datos que viven dentro de μ−kernel son dependientes del procesador. Su diseño tiene que guiarse por predicción de rendimiento y análisis. El diseño que se presenta muestra que es posible lograr un μ−kernel de buen rendimiento a través de implementaciones específicas de procesador de abstracciones independientes del procesador

5. ¿QUÉ TAN EXITOSA ES ESTA SOLUCIÓN?

En el artículo se menciona de que hipótesis que se sostienen acerca de el mal rendimiento de los sistemas μ−kernel no está justificada, pero por otro lado se deja claro de que el éxito y/o fracaso dependerá en la medida en que las implementaciones de las abstracciones y el soporte especifíco al procesador sea el adecuado.