Facultad de Informática
Universidad Nacional de La Plata

Tesis de Maestría en Cómputo de Altas Prestaciones

Un kernel diseñado para la virtualización

Lic. Matías Zabaljáuregui

Director: Lic. F. Javier Díaz

Índice de Capítulos

1. Introducción 2

2. Definiciones y clasificaciones 12

3. Virtualización clásica y la arquitectura x86 21

4. Traducción binaria 33

5. Paravirtualización 44

6. Virtualización de dispositivos 55

7. Virtualización asistida por hardware 65

8. Un kernel diseñado para la virtualización 77

9. Conclusiones y trabajo a futuro 87

A. Interrupciones y excepciones en x86 89

1. Introducción

1.1. Contexto del trabajo

En este trabajo, se denomina virtualización a la creación de abstracciones de dispositivos o recursos físicos con el fin de ofrecer sus servicios a uno o más entornos de ejecución. Concretamente, es una técnica que permite ocultar funcionalidades de un dispositivo (disco rígido, placa de red, memoria) o recurso (servidor, red, sistema operativo), sumar las de varios con el fin de presentarlos como otra entidad con capacidades diferentes o bien crear un equivalente virtual de los mismos.

En términos generales, la virtualización hace independientes a las instancias de recursos virtualizados del sustrato físico subyacente, presentándolas de manera transparente a los usuarios y aplicaciones que los utilizan sin distinguirlos de los reales.

La tendencia en infraestructura computacional: la necesidad de virtualizar recursos

La virtualización no es un concepto nuevo. Las primeras (y enormes) computadoras empresariales tipo *mainfraime*, diseñadas y construidas en la década de 1960, proveían una experiencia de computación compartimentalizada. Por diseño, estos mainfraimes originales podían generar completos espacios operativos virtuales distintos, con sistemas operativos y máquinas virtuales aisladas, que segregaban completamente los procesos y operaciones de un usuario con respecto a los de los demás. Esta arquitectura segura basada en virtualización se construía sobre anillos (*rings*) de protección que determinaban en qué nivel de privilegios se ejecutaban los procesos de usuario y del sistema.

El movimiento hacia la computación descentralizada se originó con el surgimiento de la computación personal. Para finales de la década de 1980, las computadoras eran considerablemente más potentes, numerosas y baratas que sus predecesoras, y la necesidad de multiplexar un equipo para varios usuarios se había eliminado mediante el uso de sistemas operativos multitarea y multiusuario. Las medianas y pequeñas organizaciones, que no tenían acceso a la infraestructura necesaria para implementar computación centralizada, aprovecharon la sofisticación de las pequeñas computadoras y sus aplicaciones.

El límite inherente de la computación descentralizada, sin embargo, es la escalabilidad. Resulta que hay límites para el número de servidores que una organización puede agregar sin incurrir en costos significativos por espacio físico en el centro de datos, por capacidades de recuperación de desastres, por mantenimiento, por licenciamiento y por el soporte necesario. El sostenimiento de la computación descentralizada parece estar volviéndose poco práctica y demasiado costosa.

Por otro lado, la computación centralizada no desapareció durante este proceso de tantos años de desarrollo de computación descentralizada. De hecho, ocurrió todo lo contrario. Los protagonistas actuales de la tecnología de virtualización mejoraron sus soluciones, teniendo a las grandes empresas como su porción de mercado principal. Sin embargo, en la actualidad han diseñado servicios centralizados que atraen a organizaciones de todos los tamaños y han desarrollado productos que apuntan al "problema de los servidores" con el que las organizaciones medianas y pequeñas deben luchar diariamente. Haciendo la virtualización tanto tecnológica como económicamente accesible a las pequeñas organizaciones, los proveedores de infraestructura virtualizada ofrecen mejores servicios tecnológicos con menores costos.

Al desacoplar el servidor-software del servidor-hardware, y similarmente al separar el escritorio-software del hardware de escritorio, las organizaciones pueden invertir menos en equipamiento informático. Esto significaría menos servidores *on-site*, clientes livianos en los escritorios, almacenamiento virtualizado, mejor manejo de licencias e incluso redes virtuales. El beneficio

más notable de la virtualización es la reducción significativa en el costo de la infraestructura informática para un entorno computacional dado.

Cloud y Grid Computing

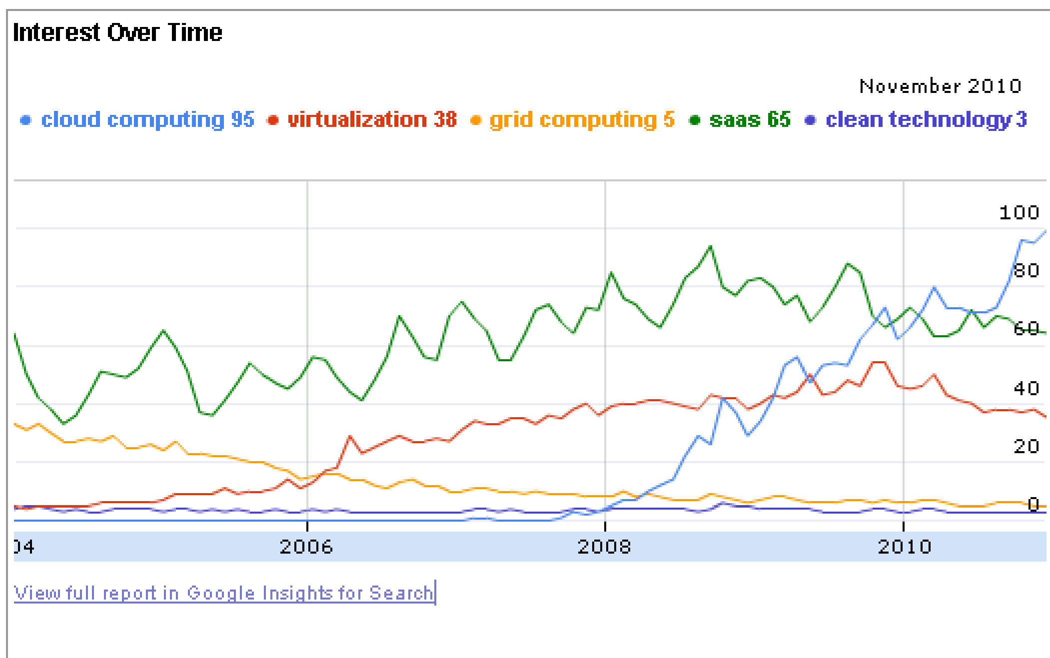
Por los motivos expuestos, hemos presenciado, en los últimos años, la aparición de nuevas formas de acceder al poder de cálculo y capacidad de almacenamiento de grandes infraestructuras de cómputo. El fácil acceso a los recursos distribuidos ha permitido la concepción de nuevos paradigmas de computación, entre los que podemos encontrar *Grid Computing, Cloud Computing,* Arquitecturas Orientadas a Servicios, Infraestructura como Servicio, Software como Servicio, *Utility Computing*, etc.

Parece ser irreversible la tendencia a delegar 'el problema de la continuidad de la infraestructura', confiando en una infraestructura común particionada (posiblemente externa) y reconfigurable (casi instantáneamente) para satisfacer las necesidades exactas en un momento dado. Agregar más espacio de almacenamiento ya no significará agregar más discos, y los recursos no estarán asociados a una función particular del cálculo, proceso o negocio.

Sin intenciones de definir los conceptos de *Cloud Computing* o *Grid Computing*, que ya han sido extensivamente documentados en otros trabajos, se hace una breve reseña de su aparición en la escena tecnológica, para comprender la importancia de la virtualización como tecnología central en las soluciones de infraestructuras actuales.

Cloud Computing

Cloud Computing parece haberse vuelto el concepto más popular, en términos de interés general, como lo demuestra el estudio [TWEAK] realizado por el sitio *cloudtweaks.com* en base a las búsquedas realizadas en el motor de búsquedas Google:



En el gráfico se observa que el interés en temas relacionados con *Cloud Computing* comienza a crecer rápidamente a partir del año 2008, superando, a mediados del año 2010, al interés demostrado por las técnicas alternativas y/o complementarias.

Por otro lado, Gartner, la mayor y más prestigiosa empresa de Investigación y Asesoramiento en Informática y Telecomunicaciones del mundo, posiciona en primer lugar a las soluciones

basadas en *Cloud Computing* en su última edición de las diez tecnologías estratégicas reconocidas como más relevantes para el año 2011 [GART11].

Aunque este dato aislado es importante, también resulta interesante observar y comparar el pronóstico de Gartner para el año 2011 con respecto a los realizados para los años 2010 [GART10] y 2009 [GART09]. En estos dos últimos la virtualización se presentaba, como técnica separada de *Cloud Computing*, liderando la tabla en el pronóstico para el año 2009 y apareciendo dos veces en la lista del pronóstico para el año 2010.

No resulta extraño que, a pesar de esto, la virtualización no sea siquiera mencionada en la tabla de tecnologías más importantes para el año 2011. Ya en septiembre del 2008, un grupo de expertos en virtualización y *Cloud Computing* reunidos en la conferencia de Tecnologías Emergentes en el Instituto de Tecnología de Massachusetts (MIT, por sus siglas en inglés) advirtieron [MITEXP]:

> *"Las buenas noticias son que la virtualización se convertirá en la parte crítica de algo mucho más grande en la mayoría de las infraestructuras tecnológicas, a medida que el tiempo pase. Las malas noticias son que lo hará como parte de un movimiento más importante hacia cloud computing y, en gran medida, aquella desaparecerá como una disciplina separada"*

Cloud computing puede ser caracterizada como la "evolución natural por la amplia adopción de virtualización de recursos, Arquitecturas Orientadas a Servicios y *Utility Computing*" **[WIKICL]**. Es decir, en el contexto de este trabajo, se considera a la virtualización de recursos como la tecnología clave sobre la cual se construye el modelo de provisión, consumo y entrega de servicios de infraestructura informática que se prevé será dominante en el futuro inminente.

Grid Computing

Un término que acompañó al desarrollo de la computación distribuida en los últimos años, principalmente en los entornos científicos, es el de *Grid Computing*. Aunque el desenfrenado interés mostrado por la comunidad en el pasado reciente parece haberse trasladado a las soluciones basadas en *Cloud Computing*, parecería injusto e incorrecto pensar que la idea fundamental que gestó el movimiento de *Grid Computing* haya sido errónea o haya quedado obsoleta. Al igual que con la virtualización, la idea de *Grid Computing* fue el paso final en la evolución que derivó en la concepción de *Cloud Computing*.

De hecho, según cierta literatura [IBMCLO], para imaginar una implementación de *Cloud Computing*, necesitamos tres elementos básicos: alguna forma de cliente liviano o aplicación *front-end* (que podría ser un navegador Web), una solución similar o igual a un *middleware* de *Grid Computing* y alguna solución de *Utility Computing*.

Grid Computing agrupa recursos distribuidos para formar una gran infraestructura virtual. *Utility Computing* permite facturar en términos proporcionales a los recursos que el usuario usa en la infraestructura compartida como se cobra por un servicio público, como la electricidad. *Cloud Computing* va un paso más adelante con la posibilidad de aprovisionamiento de recursos de manera dinámica, bajo demanda.

Ya en el año 2007, cuando aún no se utilizaba el término *Cloud Computing*, algunos analistas tenían una visión bien clara de cómo las distintas tecnologías que se presentaban como alternativas entre si, en realidad eran manifestaciones de un mismo movimiento:

> *"A medida que la tecnología Grid sea incorporada por las infraestructuras, podría volverse inseparable de tecnologías tales como la virtualización y arquitecturas orientadas a servicios (SOA), y por lo tanto la creación de utilidades"* [GRID]

Una consecuencia, según diagnosticaban los expertos en ese año, es que "el término *Grid Computing* se volverá más relevante pero menos utilizado". Aunque dejando la terminología de lado, parecía que la virtualización podía "ofrecer el sueño de *Grid Computing*, asumiendo que el sueño es la provisión de recursos computacionales, bajo demanda, desde fuentes

distribuidas". Un análisis realmente acertado en aquella época, como se comprobó con el tiempo.

Evidentemente, en ese momento se presentía una tendencia a la convergencia entre estas tecnologías. Para correr un trabajo (*job*) en una infraestructura *Grid,* un usuario debía identificar un conjunto de plataformas capaces de correr ese trabajo, con el sistema operativo y librerías apropiadas, entre otras cosas. En cambio, la virtualización introducía una capa de abstracción, por lo que en lugar de tener que buscar los recursos disponibles e intentar adaptar el problema a ellos, un usuario podría describir un entorno de ejecución o *workspace* y sólo esperar a que éste sea implantado en la Grid, bajo demanda. En una descripción muy similar a lo que hoy en día se entiende como Infraestructura como Servicio, los expertos anticipaban:

"Las máquinas virtuales y las appliances virtuales, junto con el almacenamiento distribuido y las redes overlays, posibilitarán el mapeo de este tipo de workspaces virtuales en recursos físicos. Incluso se espera que sean fácil de definir, testear, instalar, transportar y ajustarlos bajo demanda." [GRID].

Entonces, ya sea, porque fue el paso inmediatamente previo en la evolución hacia *Cloud Computing* o porque efectivamente podemos interpretarlo como parte integrante del conjunto de tecnologías que permiten la idea actual de *Cloud Computing*, también se presenta el concepto de grid computing como marco conceptual en el cuál definimos a la virtualización.

1.2. Objetivos de la tesis

Proponer un nuevo diseño de kernel-framework para ser usado en un entorno de virtualización

Los sistemas operativos actuales fueron desarrollados originalmente para ser ejecutados en una o varias arquitecturas de hardware nativo. Esto implica que gran parte de la complejidad del diseño y la mayor parte de las líneas de código que lo implementan son, en realidad, necesarias sólo para soportar la gran cantidad de plataformas de hardware y dispositivos disponibles, mientras que una minoría del código implementa las funcionalidades de alto nivel como la planificación de la CPU, los protocolos de red, los sistemas de archivos, etc.

Por ejemplo, el kernel Linux soporta 24 arquitecturas de hardware y posee drivers para más dispositivos de hardware que ningún otro sistema operativo en la historia. Esta característica lo hace único, pero a la vez implica que más del 70% de su código fuente está destinado a implementar este soporte.

La nueva oleada de técnicas de virtualización para la arquitectura x86 suele tener como sistemas operativos guests a los mismos que corren sobre el hardware nativo. En este trabajo se asume como irreversible la tendencia a la omnipresencia de plataformas virtualizadas, por lo tanto se presenta la posibilidad de reestructurar los sistemas operativos para adaptarlos a estos nuevos entornos. Es decir, a medida que la estandarización en la tecnología de virtualización avance y se termine de afianzar como una capa más de las arquitecturas de cómputo (justo por encima del hardware tradicional) los sistemas operativos que se ejecuten como guests podrán dejar de incluir toda la complejidad asociada al soporte del hardware nativo, ya que de cualquier modo no lo necesitarán.

Al momento de definir la propuesta para este trabajo, no existe una iniciativa de kernel especializado para actuar exclusivamente como guest de un VMM para una arquitectura x86 y este vacío es el que define el objetivo central de esta tesis: proponer un diseño que, basado en las técnicas de virtualización, ofrezca una plataforma de construcción de kernels guest, de manera simplificada.

Escribir un prototipo como prueba de concepto que demuestre la efectividad de esta idea

Con la gran cantidad de técnicas de virtualización emergentes, sus posibles combinaciones y distintos desarrolladores que las implementan, hace tiempo que resulta evidente una imperiosa necesidad de estandarizar, al menos, algunas de las interfaces intervinientes. El universo técnico conformado por ingenieros, investigadores y desarrolladores del kernel Linux ha sido precursor en intentar resolver la complejidad que se genera al combinar las distintas implementaciones de guests, hipervisores, drivers y dispositivos virtuales.

En este esfuerzo por la estandarización y la reducción de complejidad se destacan dos aportes fundamentales en la arquitectura de virtualización de Linux, principalmente orientados a implementar técnicas de paravirtualización:

- paravirt_ops: una interfase que permite conectar distintos guests a distintos hipervisores. [PVOPS]
- virtio: abstracción de un mecanismo de transporte que permite que los guests accedan a los dispositivos virtuales. [VIRTIO]

El segundo objetivo principal consiste en construir, sobre ambas abstracciones mencionadas anteriormente, un sistema operativo tipo framework, un template, implementado como un guest concebido desde el principio para ser ejecutado en un entorno paravirtual. Esto simplifica enormemente el desarrollo del prototipo y permite que nuestro guest se ejecute en tantas arquitecturas como soporte el hipervisor.

Por lo tanto, es necesario el estudio detallado de las técnicas actuales implementadas en el kernel Linux y el análisis de posibilidades de reusar ciertas partes de código o, al menos, exportar ciertos diseños para ser utilizados en la construcción del framework propuesto. En particular, es de especial interés el hipervisor Lguest. La mayor parte de nuestro trabajo se basa en este proyecto de software libre que implementa la técnica de paravirtualización pura dentro del kernel Linux

Entonces, como objetivos secundarios, pueden mencionarse los de familiarizarnos con el diseño y la implementación de este hipervisor, de manera que cualquier idea nueva sea fácilmente prototipada usando a Lguest como plataforma. Además de los desafíos técnicos mencionados, durante el período de investigación se formalizó una colaboración con Rusty Russel, el creador de Lguest y un importante actor en el desarrollo del kernel desde sus inicios.

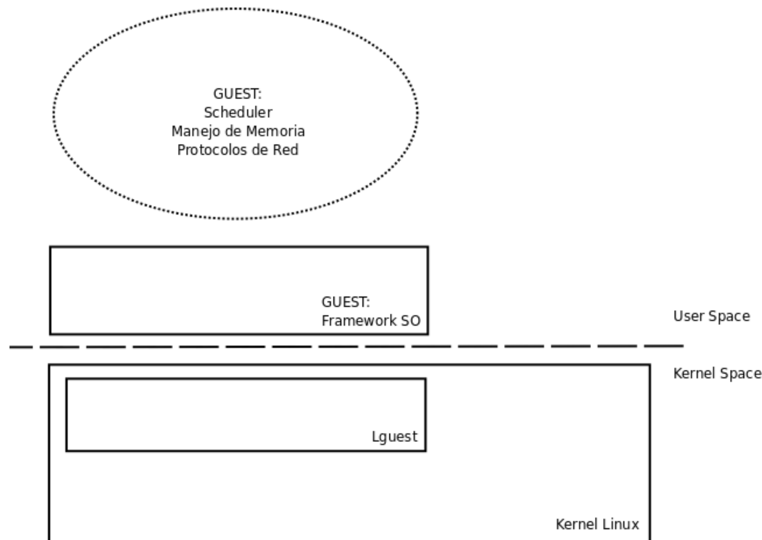
1.3. Aportes de esta tesis

Descripción de un nuevo diseño de kernel-framework para entornos virtualizados

Como se mencionó anteriormente, generalmente un guest es un sistema operativo, originalmente diseñado e implementado para ser ejecutado sobre hardware, que luego es modificado para poder ser ejecutado en un entorno virtual.

El diseño del framework descripto en este trabajo plantea la construcción sencilla de guests especializados, preparados para ejecutarse exclusivamente sobre un hipervisor. Éste representa el aporte fundamental de esta tesis.

Como se muestra en la figura, nuestro framework se construye como un guest sobre un host Linux.



Creación del código fuente del prototipo bajo una licencia de software libre y su publicación en Internet como material didáctico y para la promoción de la colaboración

El código completo del trabajo realizado al implementar el diseño propuesto fue liberado bajo una licencia GPL (General Public License) con la sincera intención de promover el estudio, la modificación, la colaboración y la corrección de este proyecto, por parte de cualquier interesado en el tema.

Para su fácil acceso, se utilizó un repositorio público, que suele albergar proyectos de software libre: GITHUB [GITHUB]. Puede accederse al código a través del proyecto denominado OSom, ubicado en [OSOM].

GitHub funciona sobre el sistema de versiones GIT (el mismo utilizado para el desarrollo del kernel Linux), ideal para proyectos colaborativos de gran escala y con desarrolladores distribuidos geográficamente.

Colaboración con el proyecto Lguest, aportando implementaciones que fueron incorporadas al kernel Linux.

"Linux parece haberse convertido en el estándar de-facto para la investigación académica relacionada con sistemas operativos. Por ejemplo, el 80% de los artículos publicados en la conferencia "21st Symposium on Operating Systems Principles, ACM" involucraron experimentación con el kernel Linux. Desafortunadamente, sólo el 1% de las contribuciones hechas al kernel en los últimos tres años pueden ser atribuidas a la comunidad académica.". Minding the Gap: R&D in the Linux Kernel, ACM Operating Systems Review. [MINGAP]

Lguest se presenta como una herramienta ideal para la investigación y la enseñanza de arquitectura de computadoras y sistemas operativos. Además se implementa como un módulo del kernel Linux que interactúa con las partes más importantes de éste, como el subsistema de memoria, el planificador y las nuevas abstracciones de Entrada/Salida, ofreciendo la posibilidad de entender en detalle el funcionamiento de éstos y su interrelación.

Pero, principalmente, Lguest es un proyecto que tiene como finalidad facilitar el estudio, investigación y modificación de un hipervisor real. Por este motivo parece alinearse perfectamente con los objetivos de nuestro proyecto. En algunos cursos de sistemas operativos, utilizamos Lguest como material de estudio [LINTI].

El autor de la tesis colabora con el proyecto Lguest desde el año 2007, corrigiendo errores y aportando nuevas funcionalidades al código del hipervisor [KERNEL]. Entre los aportes más importantes aceptados en el repositorio oficial, el soporte para PAE y PSE y la migración del

mecanismo de *hypercalls* a una implementación compatible con KVM resultaron ser funcionalidades importantes para el proyecto.

Por ejemplo, el soporte de PAE para Lguest fue fundamental, según el ingeniero responsable (y un colaborador de Lguest), Ron Minnich, en el experimento a gran escala realizado en Laboratorio Nacional Sandia, uno de los grandes laboratorios nacionales de Estados Unidos. El experimento consistió en activar un millón de kernels Linux, basándose en máquinas virtuales implementadas con Lguest con el soporte para PAE agregado. Se utilizó semejante infraestructura virtual para simular el comportamiento de una red de redes, como Internet [MINNICH].

Creación de material didáctico, utilizado en cursos de grado y postgrado.

Los libros habituales de consulta para los profesores de Sistemas Operativos incluyen el concepto de máquinas virtuales con menor o mayor profundidad. El docente se enfrenta al desafío de definir un temario donde deben convivir conceptos básicos imprescindibles y las tendencias actuales.

El primer caso de uso para el prototipo que surgió naturalmente fue la enseñanza/investigación de temas afines. Dado que el alumno no debe lidiar con los detalles del acceso al hardware, se hace viable la opción de aprender sistemas operativos escribiendo módulos de un guest: planificador, memoria, protocolos de comunicación, etc. Y se suma la posibilidad de enseñar las nuevas técnicas de virtualización a través de una implementación real de un hipervisor y un guest.

En el material generado durante estos años de investigación, incluida esta tesis, se revisaron, comprendieron y aplicaron las nuevas técnicas utilizadas en diseño e implementación de arquitecturas y sistemas operativos modernos, relacionadas con las tecnologías de virtualización. Todo el material se encuentra disponible en un sitio destinado a la difusión de estos temas [LINTI].

El material fue utilizado con éxito en la materia 'Sistemas Operativos' perteneciente a la Maestría en Redes de Datos de la Facultad de Informática de La Universidad Nacional de La Plata y en los cursos de grado Sistemas Operativos I y Sistemas Operativos II, pertenecientes a la Escuela de Tecnología de la Universidad Nacional del Noroeste de la Provincia de Buenos Aires.

1.4. Materiales y Métodos

Al ser un proyecto principalmente de software, sólo se requirió el entorno apropiado para el desarrollo, depuración y perfilamiento del código implementado. La GNU *toolchain* incluye todas las herramientas necesarias para el desarrollo de código para el kernel Linux, y goza de licencias libres.

Con respecto a la metodología, se pueden mencionar dos cuestiones que en un principio condicionaron el rápido avance en el tema de estudio, pero que finalmente resultaron ser experiencias enriquecedoras, cada una a su manera: participar en el proceso de desarrollo del kernel Linux y comprender el modelo del sistema de control de versiones llamado GIT [GIT].

Resultó ser de gran interés interactuar con la comunidad open source que implementa el kernel. La gran cantidad de desarrolladores involucrados (más de mil) obliga a una organización jerárquica de revisores de código que aseguran la calidad de los parches agregados al kernel. Esto hace que sumar un parche al kernel oficial sea un proceso técnicamente complejo y demandante en tiempo, en el cuál el autor normalmente debe defender sus diseños y decisiones de implementación en discusiones con un gran nivel de exigencia técnica.

Otra experiencia importante fue la de incorporar el modelo de desarrollo utilizando GIT, el cual ofrece una interfaz algo compleja, pero su velocidad de funcionamiento y su modelo de distribución permite al desarrollador trabajar *offline* la mayor parte del tiempo,

con la seguridad y eficiencias necesarias. Se extendió el uso de esta herramienta al código generado en el prototipo presentado.

1.5. Publicaciones relacionadas con esta tesis

Fueron publicados los siguientes reportes técnicos en el sitio creado para difundir los resultados que resultaron de esta investigación: **linux.linti.unlp.edu.ar**

Middlewares and New Virtualization Techniques
http://linux.linti.unlp.edu.ar/kernel/images/1/1f/Grid-virtualization.pdf

Lguest Hypervisor
http://linux.linti.unlp.edu.ar/kernel/images/1/14/Lguest.pdf

Hardware Assisted Virtualization - Intel Virtualization Technology
http://linux.linti.unlp.edu.ar/kernel/images/f/f1/Vtx.pdf

El siguiente artículo fue aceptado en un congreso nacional:

Desarrollo de un framework para la enseñanza de los Sistemas Operativos usando Lguest
Lic Javier Díaz, Mg. Lía Molinari, Lic. Matías Zabaljáuregui
LINTI, Laboratorio de Investigación en Nuevas Tecnologías Informáticas
Universidad Nacional de La Plata. La Plata, Argentina. Aceptado para exposición en el V Congreso de Tecnología en Educación y Educación en Tecnología que se realizó en la Ciudad de El Calafate, provincia de Santa Cruz, los días 6 y 7 de Mayo de 2010.

1.6. Organización del documento

La tesis se presenta en 9 capítulos que introducen al lector los principales términos, conceptos, técnicas e implementaciones concretas de las más recientes propuestas de virtualización, originadas tanto en el mundo académico como en la industria. Se considera muy relevante el contexto histórico, ya que como se aclaró previamente, las nuevas implementaciones no son más que adaptaciones de antiguas ideas. Y, principalmente, se asume cierta preparación previa del lector en temas centrales de sistemas operativos y arquitectura de computadoras, teniendo en cuenta que fue imposible incluir la explicación de cada concepto involucrado, por cuestiones de espacio.

Se decidió presentar la bibliografía de manera fragmentada al final de cada capítulo. Esta forma, inspirada por algunos libros de texto, resultó más fácilmente manejable y modificable, y se adapta al hecho de que cada capítulo, más allá de su evidente interrelación con el resto del trabajo, aborda un tema discreto y prácticamente autocontenido, por lo que no se descarta la posibilidad de que esta tesis sea utilizada como material de referencia para el tema completo o para los temas particulares encarados en cada capítulo.

El capítulo 2 muestra un primer acercamiento a los usos de la virtualización en su concepción más genérica. Simplemente como capa de abstracción entre usuarios y recursos compartidos, la virtualización se hace presente en cada nivel de la pila arquitectónica de una plataforma de cómputos. También se presenta una primera clasificación: virtualización horizontal y vertical, brindando ejemplos de cada una de estas categorías. El capítulo concluye con las definiciones fundamentales para el resto del trabajo, las distintas formas de virtualización vertical y los beneficios que suelen ser citados en la mayor parte de la literatura relacionada con virtualización.

El capítulo 3 comienza con una interesante revisión histórica que inicia en la década de 1960 y en la cuál se comprueba que las técnicas usadas actualmente fueron concebidas principalmente en el contexto de computación centralizada de aquella época. Luego se introducen los requerimientos de virtualización publicados por Popek y Goldberg en el año 1974. Tanto las condiciones suficientes como sus dos teoremas serán mencionados en el resto

del trabajo. Se define exactamente lo que se entiende por virtualización clásica, describiendo cada uno de los mecanismos que permiten la histórica aproximación de virtualización por *trap-and-emulate,* principalmente el de-privileging y las estructuras shadow. Ambos conceptos también son centrales en el resto del trabajo. La segunda parte del capítulo introduce cuestiones relacionadas con la creación de VMMs para x86 y, por lo tanto, los obstáculos de una arquitectura que no es clásicamente virtualizable. Finaliza con una escueta introducción a los refinamientos de la virtualización clásica y una mención a las nuevas técnicas de virtualización para x86.

En el capítulo 4 se inicia el estudio de la primera de las tres técnicas que se exploran en profundidad en este trabajo: la traducción binaria. La primer parte está dedicada a las definiciones previas y complementarias, en particular aquellas relacionadas con compilación dinámica en general. Luego se analiza la solución ofrecida por VMware, por ser la más madura del mercado y una pionera en esta técnica para x86. Luego de evaluar los problemas de performance, se mencionan los mecanismos avanzados de la traducción binaria adaptativa.

El capítulo 5 aborda la paravirtualización, el tema central de esta tesis. Luego de una introducción conceptual, el texto describe el componente, llamado *paravirt_ops,* que es fundamental en el soporte de paravirtualización en el kernel Linux y una pieza clave para el desarrollo del prototipo presentado en este trabajo. Luego se hace una introducción al proyecto Lguest. Se describen ciertos detalles de implementación del VMM, del mecanismo de hypercalls y del proceso de inicialización de un guest. Finalmente, se mencionan las modificaciones realizadas al código de Lguest durante el trabajo de investigación realizado para esta tesis.

La virtualización de dispositivos de entrada/salida es el tema estudiado en el capítulo 6. Sus tres formas principales, emulación, paravirtualización y asignación directa, son explicadas en detalle y comparadas entre sí para obtener ciertas conclusiones. Por ejemplo, qué técnica se aplica mejor a determinados escenarios. La segunda parte del capítulo presenta a *virtio*. El segundo componente esencial para el diseño del prototipo presentado en este trabajo. Virtio es un componente del kernel Linux que abstrae a los desarrolladores de virtualización de los problemas asociados con la paravirtualización de dispositivos.

El capítulo 7 explica la última gran extensión a la arquitectura que los principales fabricantes de procesadores x86 han introducido en los últimos años. La virtualización asistida por hardware, aunque tiene reminiscencias de tecnologías antiguas de mainframes, sigue siendo una novedad importante por las alternativas arquitectónicas que ofrece a los nuevos servidores de cómputo y almacenamiento. Habiendo pasado ya dos generaciones de estos nuevos procesadores, los cambios al hardware son importantes y las posibilidades que se abren deben ser estudiadas con cuidado. La primera parte explica los cambios originales a los mecanismos de protección tradicionales de x86, para permitir un nuevo modo de ejecución para el VMM. Luego se estudian las extensiones agregadas en la segunda generación, principalmente orientadas a la virtualización del MMU y de la Entrada/Salida.

El prototipo de la propuesta realizada se presenta en el capítulo 8. Luego de una introducción que describe la idea, se presenta brevemente la arquitectura y la implementación del kernel framework diseñado exclusivamente para ejecutarse como un guest de Lguest. Con cierto nivel de detalle se aclaran algunos mecanismos, pero se intenta no confundir al lector con demasiado código fuente. Por eso, se ofrece acceso público al código fuente del proyecto alojado en un repositorio GIT en Internet y licenciado GPL.

Finalmente, el capítulo 9 presenta las conclusiones finales del trabajo y menciona los posibles trabajos relacionados y líneas de trabajo futuras.

Bibliografía

[TWEAK] Interés general en cloud computing vs virtualización vs saas
http://www.cloudtweaks.com/2010/01/mega-trends-cloud-vs-virtualization-vs-saas-so-who-wins/

[GART11] Gartner Identifies the Top 10 Strategic Technologies for 2011
http://www.gartner.com/it/page.jsp?id=1454221

[GART10] Gartner Identifies the Top 10 Strategic Technologies for 2010
http://www.gartner.com/it/page.jsp?id=1210613

[GART09] Gartner Identifies the Top 10 Strategic Technologies for 2009
http://www.gartner.com/it/page.jsp?id=777212

[MITEXP] The Mix and Match Trouble with Virtualization and Cloud Computing
http://www.cio.com/article/451469/The_Mix_and_Match_Trouble_with_Virtualization_and_Cloud_Computing_

[WIKICL] http://en.wikipedia.org/wiki/Cloud_computing

[IBMCLO] http://www.ibm.com/developerworks/web/library/wa-cloudgrid/
(http://www.cmather.com/blog/cloud-computing-virtualisation-infrastructure/)

[GRID] http://www.gridcomputingplanet.com/news/article.php/3651536

[PVOPS] Connecting Linux to hypervisors. http://lwn.net/Articles/194543/

[LGUEST] Lguest: A simple virtualization platform for Linux.
http://www.linux.com/archive/feature/126293

[VIRTIO] virtio: towards a de-facto standard for virtual I/O devices. Rusty Russell IBM OzLabs, Canberra, Australia. Newsletter ACM SIGOPS Operating Systems Review - Research and developments in the Linux kernel archive Volume 42 Issue 5, July 2008

[RUSTY] http://en.wikipedia.org/wiki/Rusty_Russell

[KERNEL] http://git.kernel.org/?p=linux%2Fkernel%2Fgit%2Ftorvalds%2Flinux-2.6.git&a=search&h=HEAD&st=commit&s=zabaljauregui

[LINTI] http://linux.linti.unlp.edu.ar/

[GIT] http://git-scm.com/

[TEYET] *Desarrollo de un framework para la enseñanza de los Sistemas Operativos usando Lguest.* Lic Javier Díaz, Mg. Lía Molinari, Lic. Matías Zabaljáuregui
LINTI, Universidad Nacional de La Plata. La Plata, Argentina. V Congreso de Tecnología en Educación y Educación en Tecnología que se realizó en la Ciudad de El Calafate, provincia de Santa Cruz, los días 6 y 7 de Mayo de 2010.

[GITHUB] https://github.com/

[OSOM] https://github.com/dc740/OSom

[MINGAP] Muli Ben-Yehuda, Eric Van Hensbergen, Marc Fiuczynski (2008) Minding the gap: R&D in the Linux kernel, 1-3. In *ACM SIGOPS Oper. Sys. Rev.* 42 (5).

[MINNICH] https://share.sandia.gov/news/resources/news_releases/sandia-computer-scientists-successfully-boot-one-million-linux-kernels-as-virtual-machines/

2. Definiciones y clasificaciones

En este capítulo se introducen definiciones más precisas y algunas clasificaciones que ordenan las distintas formas de virtualización existentes en la actualidad. Desde una primera exposición genérica de la virtualización, interpretada como capa de abstracción aplicable a distintos niveles en una infraestructura computacional, hasta la clasificación de las distintas formas de arquitectura de implementaciones de virtualización, se recorren ciertos conceptos y taxonomías vigentes, aunque aún no del todo estandarizadas.

Se hace una revisión de una aproximación organizada en pila, identificando las capas de abstracción involucradas en estas técnicas y se introduce una nueva forma de entender la virtualización a través de dos dimensiones: vertical y horizontal.

Finalmente se presentan los que suelen ser mencionados como beneficios comprobados de aplicar las técnicas de virtualización a una infraestructura de cómputo y/o almacenamiento.

2.1 Virtualización genérica

En todo escenario en el que se utiliza la virtualización existen distintos actores con roles bien definidos: los usuarios que utilizan los servicios ofrecidos, los recursos compartidos que implementan la funcionalidad real y la capa de virtualización que mapea las instancias de estos recursos con las invocaciones que hacen los usuarios.

Los usuarios acceden a los servicios ofrecidos por los recursos a través de una interfaz virtual, implementada por la capa de virtualización, la cual a su vez accede a los recursos utilizando una interfaz real o física.

Supongamos una aplicación desarrollada para ejecutarse en una infraestructura de Grid Computing. Cada vez que el desarrollador desea acceder a los datos distribuidos, debe hacerlo utilizando las llamadas a funciones del *middleware* de *Grid* que oculta la complejidad de la distribución de los datos en la infraestructura, ofreciendo generalmente una visión jerárquica de los archivos que suelen estar almacenados en distintos servidores de datos. El implementador del *middleware*, a su vez, debe traducir estos accesos virtuales a accesos reales a los archivos físicos almacenados en los discos distribuidos.

Ésta es una forma sobre-simplificada de plantear la idea, pero nos permite comprender las interacciones entre las diversas capas de la arquitectura de un centro de cómputo moderno y por lo tanto parece ser una buena forma de plantear algunos de los muchos casos de uso que existen en la actualidad para esta idea genérica, como muestra la figura 2.1.

Es importante destacar desde un principio que en esta definición genérica se han combinado dos grandes formas de virtualización: la virtualización horizontal y la virtualización vertical [BMAR]. La virtualización horizontal es aquella virtualización que abstrae a los usuarios de la complejidad inherente de las infraestructuras de recursos distribuidos. Como se mencionó anteriormente, las distintas formas de Grid Computing y Cloud Computing son ejemplos de virtualización en esta dimensión.

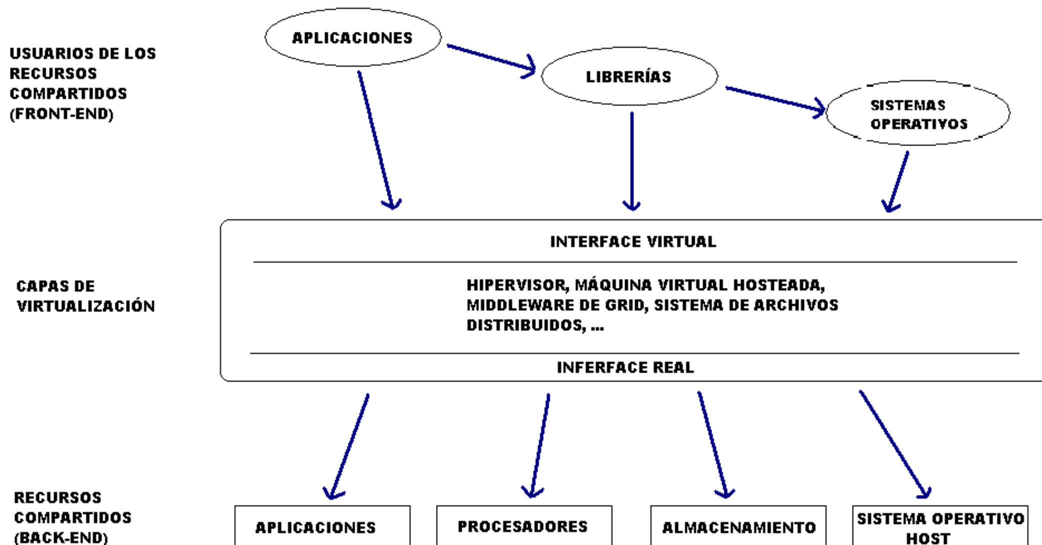


Figura 2.1 - Virtualización Genérica

Por otro lado, la virtualización vertical es la idea de virtualización aplicada entre las capas arquitectónicas de una infraestructura computacional. Los ejemplos más comunes de este tipo de virtualización son aquellos ofrecidos por hipervisores, emuladores de hardware, emuladores de entornos de ejecución, etc. Como se detalla más adelante, este trabajo se centra principalmente en las formas más relevantes de virtualización vertical de la actualidad, que son, como veremos, herencia directa de ideas gestadas en la década de 1960.

Casos de uso de la virtualización en una infraestructura moderna

Un primer método que puede resultar adecuado para entender las variantes de las técnicas de virtualización es recorrer la pila arquitectónica de la infraestructura computacional moderna. La figura 2.2 muestra las capas de virtualización que suelen coexistir. Las funcionalidades de cada una de las capas pueden ser ofrecidas de manera transparentes a las capas superiores. A la izquierda del diagrama pueden verse los ejemplos para cada capa propuesta.

Como puede observarse, los estratos de la infraestructura se apilan uno sobre el otro para ir sumando en nivel de abstracción y ofrecer finalmente funcionalidades de software como un servicio, siguiendo el modelo de *Software as a Service*. A continuación realizamos un análisis descendente de estas capas para mencionar los usuarios, recursos compartidos y capas de virtualización involucradas en cada nivel.

EJEMPLOS DE CAPAS DE VIRTUALIZACIÓN

Software as a Service	SERVICIOS
Utility Computing	APLICACIONES
Grid de datos	FUENTES DE DATOS
Máquina Virtual	SISTEMAS OPERATIVOS
Hipervisor	HARDWARE DE CÓMPUTO
Grid de Almacenamiento	ALMACENAMIENTO FÍSICO

Figura 2.2 - Virtualización en la pila arquitectónica computacional

El modelo de *Software as a Service* (Saas) hace disponible el software de la infraestructura, a través de interfaces basadas en servicios, posiblemente a múltiples organizaciones externas. Ésta es una forma de virtualización horizontal. Los recursos compartidos son las aplicaciones compartidas y los usuarios son los clientes que las utilizan, generalmente pagando por ello. La capa de virtualización se implementa con arquitecturas *multi-tenant* o multi-propietario en las cuales todos los clientes y sus usuarios consumen el servicio desde la misma plataforma tecnológica. M*ulti-tenant* se refiere a un principio en la arquitectura del software donde una única instancia del software se ejecuta en un servidor y da servicio a múltiples clientes. Esta idea contrasta con una arquitectura multi-instancia

Las Grid de datos permiten la compartición transparente de servidores de datos entre múltiples aplicaciones a lo largo de una organización o varias organizaciones. El recurso compartido está representado por las fuentes de datos distribuidas, los usuarios están representados por los grupos que acceden a estos datos y la capa de virtualización se implementa a través de un *broker* o intermediario de fuentes de datos.

Como estudiaremos más adelante, una máquina virtual implementada sobre un VMM de tipo 2 se ejecuta sobre un sistema operativo host. Suele usarse para particionar recursos o para ejecutar otro sistema operativo guest. En este caso los usuarios son los sistemas operativos guest, los recursos compartidos son aquellos recursos de hardware ofrecidos por la plataforma y la capa de virtualización se implementa como una aplicación que se ejecuta sobre un sistema host para crear una máquina virtual.

Por otro lado, una máquina virtual implementada sobre un VMM de tipo 1 se ejecutará sobre un hipervisor, sin la necesidad de ser soportado por un sistema operativo host. Los usuarios son sistemas operativos guest que en este caso se ejecutan eficientemente sobre un hipervisor construido directamente sobre el hardware nativo. Nuevamente, los recursos compartidos son aquellos ofrecidos por la plataforma.

Finalmente, se puede mencionar a las Grids de almacenamiento, cuya función es permitir el acceso transparente a los dispositivos de almacenamiento físico distribuidos, con la clara reducción de costos de infraestructura. Los recursos compartidos son los dispositivos de almacenamiento como discos o cintas y los usuarios son las aplicaciones o capas del

middleware o sistema operativo que accede a estos dispositivos. La capa de virtualización es un algún *broker* de almacenamiento.

Hasta ahora, ha habido una necesidad de estándares adicionales tanto en virtualización vertical como virtualización horizontal. Una cuestión clave es la integración de capacidades de virtualización horizontal y vertical como, por ejemplo, máquinas virtuales y Grid Computing. Esta combinación, a veces llamada virtualización distribuida, permite la implantación rápida y segura de aplicaciones en entornos heterogéneos distribuidos para operaciones basadas en red. Como se explicó anteriormente, durante los últimos años los expertos vaticinaban un gran desarrollo de infraestructuras basadas en virtualización distribuida. Parece ser que la reciente explosión de Cloud Computing no es más que la convergencia entre ambos tipos de virtualización.

Sin embarbo, esta tesis se centra principalmente en las formas de virtualización vertical. Es por esto que a continuación se realiza un recorrido por las distintas maneras de aproximarse a una de estas formas de virtualización, haciendo hincapié en las alternativas que se consideran más relevantes en la actualidad para la virtualización de infraestructuras.

El concepto de VMM o Hipervisor

Un Monitor de Máquinas Virtuales (VMM, Virtual Machine Monitor) o Hipervisor es una capa de software que permite que múltiples sistemas operativos se ejecuten de manera simultánea en un mismo equipo. El término VMM apareció en 1972 y se refería a la interfaz a través de la cual un sistema operativo guest podía acceder a servicios virtualizados por el programa de control (el **CP** de los sistemas operativos CP/CMS de IBM, explicado más adelante), de manera análoga a una llamada a sistema por parte de una aplicación corriendo dentro de un sistema operativo no virtualizado.

Dependiendo de su implementación, principalmente de cómo interactúan con la capa inmediatamente inferior y con la capa superior, los VMMs han sido clasificados en la historia reciente en dos tipos principales. Los VMM tipo 1 y los VMM tipo 2, como aclara Tanenbaum en la última versión de su libro sobre sistemas operativos [TAN08]. La principal diferencia entre el tipo 1 y el tipo 2 radica en si el VMM es ejecutado directamente sobre el hardware nativo, de la misma forma que lo haría un kernel monolítico tradicional, o si es implementado como una aplicación que se ejecuta sobre un sistema operativo *host*.

VMM de tipo 1 o hipervisor

También denominado como *unhosted.* En este caso, el VMM corre directamente sobre el hardware nativo, teniendo que implementar sus propios drivers, módulo de manejo de memoria y planificación de la CPU. Es decir, se lo puede considerar como una aproximación de microkernel especializado para la ejecución de sistemas operativos guests. El VMM puede ofrecer la totalidad o sólo algunas de las funcionalidades del hardware nativo a las máquinas virtuales que se ejecuten sobre el mismo. El siguiente diagrama muestra al VMM como la primera capa de software ejecutándose sobre el hardware:

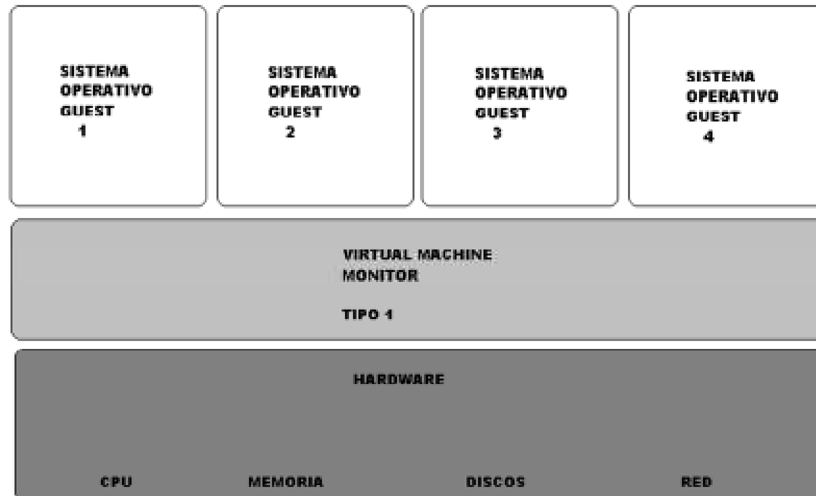


Figura 2.3 – VMM de tipo 1

Al tener acceso directo a los recursos de hardware, en lugar de tener que atravesar las capas de un sistema operativo, un hipervisor es más eficiente que su alternativa, un VMM tipo 2, y ofrece mayor escalabilidad, fiabilidad y mejor performance al reducir latencias e incrementar el rendimiento.

VMM de tipo 2 o hosteado

En este caso el VMM se ejecuta sobre un sistema operativo que es el administrador real del hardware físico. En general el VMM de tipo 2 se implementa como un proceso tradicional del sistema operativo host. Por lo tanto, debe capturar las invocaciones a funcionalidades del hardware que realicen las máquinas virtuales y traducirlas a invocaciones reales al sistema operativo host, el cuál a su vez debe transformar en llamadas a funciones del hardware nativo.

Esta clara separación entre máquina virtual y hardware subyacente tiene la ventaja de facilitar la implementación y permitir un amplio abanico de posibilidades a la hora de ofrecer instancias virtuales de recursos o dispositivos de hardware virtualizados.

Sin embargo, los altos costos de performance de estas soluciones, comparadas con la alternativa del VMM tipo 1, hacen que en los entornos de producción no se utilice esta forma de implementación. Debe tenerse en cuenta que cada intento de acceso al hardware subyacente por parte de una máquina virtual deberá atravesar más dominios de protección y generará más cambios de contexto que en el caso anterior.

En comparación con los VMM de tipo 1, los de tipo 2 se consideran menos confiables, menos escalables, mucho menos eficientes y con latencias considerables. La gran ventaja radica en que pueden ser ejecutados como un simple programa, razón por la cuál un usuario final suele preferir instalar esta variante en su estación de trabajo cuando sus objetivos no incluyen requerimientos de escalabilidad o alta eficiencia. La siguiente figura muestra un VMM de tipo 2:

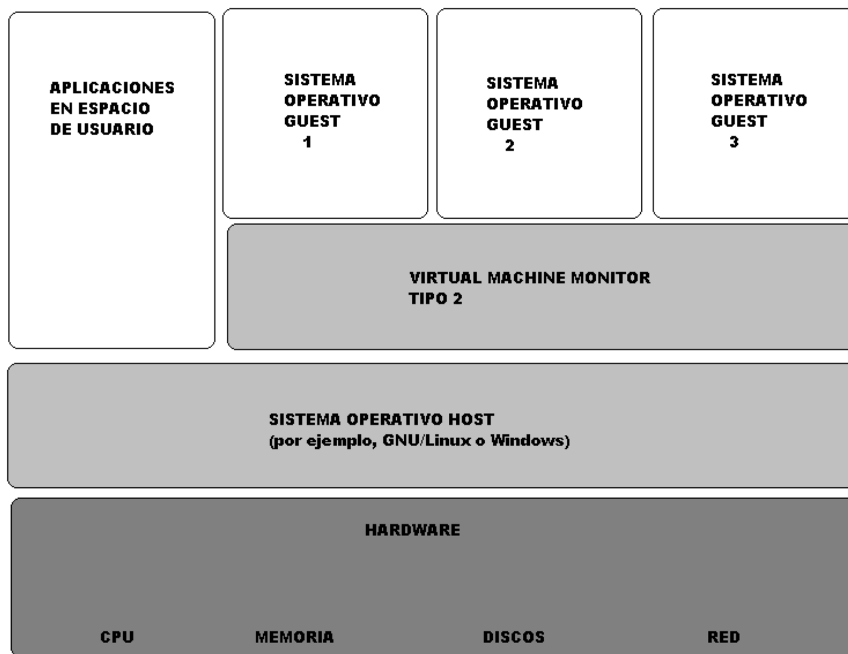


Figura 2.3 – VMM de tipo 2

Formas de virtualización vertical

Teniendo en cuenta la primera clasificación realizada en la sección anterior, a continuación se exploran las formas alternativas y complementarias de aplicar las ideas de virtualización a distintos escenarios. Aunque no todas las aproximaciones mencionadas aquí son relevantes para este trabajo, parece importante listarlas por razones de completitud. Además compararlas entre ellas puede ayudar al lector a aproximarse sin confusiones a los temas principales de la tesis e ir dejando a un lado aquellos aspectos de la virtualización que no serán tratados en la misma.

- **Emulación de un sistema no nativo:** en este tipo de implementación se emula cierto hardware distinto al nativo para permitir que aplicaciones y/o los sistemas operativos concebidos para otras arquitecturas de procesador se puedan ejecutar sobre el mismo. No es necesario que el hardware emulado exista físicamente: en algunos casos bastará con que exista su especificación. Así mismo, esta forma de virtualización permite la portabilidad de aplicaciones ya que sólo el emulador debe ser portado de una arquitectura a otra. *Ejemplos:* JavaVM[JAVA], CLI de .Net[CLI].

- **Emulación de un entorno de ejecución:** en este caso se emula *únicamente* el entorno de ejecución de una o más aplicaciones. Usado sobre todo para hacer frente a la obsolescencia de software desarrollado para plataformas o sistemas operativos antiguos o bien para lograr mayor portabilidad entre distintos sistemas operativos. Ejemplos: DOSBox[DBOX], Wine[WINE], Cygwin[CYG].

- **Emulación del hardware subyacente (ejecución nativa):** En este caso, varias máquinas virtuales independientes corren sobre un VMM de tipo 1 o 2. Cada una de ellas corre su propio sistema operativo, el cual debe ser capaz de correr sobre el hardware físico subyacente. El VMM particiona el hardware para compartirlo y presentarlo a cada VM. *Ejemplos:* VirtualBox[VBOX], VMware ESX Server[VMSX], VMware Player[VMPL], VMware Server[VMSV].

- **Virtualización a nivel de sistema operativo:** un mismo sistema operativo se divide en varios *entornos virtuales* independientes que comparten un kernel común, o al menos

similar. Esto último la convierte en una técnica de virtualización muy eficiente (no requiere tantos recursos para cambios de contexto ya que las *system calls* no son emuladas) aunque limita la variedad de contextos emulados a los soportados por un kernel común. *Ejemplos:* Solaris containers[SCON], FreeBSD jail[JAIL], entorno chroot[CHRO].

- **Paravirtualización:** se utiliza una interfaz virtual presentada a una o más máquinas virtuales, siendo la misma similar, pero no idéntica al hardware físico subyacente. Esto permite reducir la porción de tiempo de ejecución (en ciclos de CPU) que el host virtualizado consume realizando operaciones privilegiadas, lo que mejora el rendimiento y simplifica el diseño del VMM. Hay que señalar que la paravirtualización requiere que el sistema operativo a virtualizar sea portado a la *para-API* (ver capítulo 4). *Ejemplos:* Xen[XEN], Lguest[LGUE], L4[L4].

Beneficios de la virtualización vertical

Se suelen explotar cuatro capacidades fundamentales para implementar una solución de virtualización:

- **Compartición de recursos:** Un único recurso físico puede ser redefinido como varios recursos virtuales.
- **Agregación de recursos:** Varios recursos físicos pueden ser definidos como un único recurso virtual.
- **Emulación:** Los recursos virtuales pueden tener funciones o características que no están disponibles en los recursos físicos subyacentes.
- **Aislamiento:** La separación de un recurso físico en varios recursos virtuales, mediante la compartición de recursos, permite un aislamiento que no es posible obtener compartiendo un recurso físico de la manera tradicional.

Estos conceptos se han vuelto críticos en los procesos de optimización de infraestructuras. Pueden ser simplemente un medio para reducir y simplificar el centro de datos o puede llegar a ser una herramienta para transformar la visión global del mismo. En la actualidad se suelen mencionar algunos de los siguientes como los principales beneficios de las diversas técnicas de virtualización.

Consolidación

El objetivo de la consolidación es combinar y unificar. En el caso de la virtualización, las cargas de trabajo se combinan en menos servidores físicos siempre que sean capaces de soportar la demanda global de carga de trabajo. Actualmente, en muchos centros de datos sin virtualización, las cargas de trabajo de un servidor están lejos de consumir sus recursos, resultando en un uso ineficiente de infraestructura e inversión. La virtualización permite combinar varios sistemas operativos y sus cargas de trabajo en un único servidor compartido con suficientes recursos.

Con la correcta planificación y conocimiento de las cargas de trabajo, el resultado es un incremento de la eficiencia en la utilización del hardware y una disminución en el consumo eléctrico, los costos de administración y el espacio físico requerido.

Sistemas antiguos

La virtualización permite ejecutar sistemas antiguos sobre un hardware virtualizado común y compatible. Esto permite la migración desde servidores antiguos, no soportados y poco fiables hacia hardware nuevo y soportado, con el mínimo impacto.

Virtualización de los puestos de trabajo

En los últimos años, se ha extendido las ideas de virtualización a los escritorios de trabajo para obtener ciertas ventajas asociadas. Principalmente, la posibilidad de heterogeneidad de sistemas operativos sobre el mismo hardware virtualizado, la reducción en el costo de administración y el uso más eficiente de los recursos del hardware de los puestos de trabajo (generalmente subutilizado).

Entornos de Prueba

Una infraestructura virtual independiente del hardware real permite la configuración sencilla de flexibles entornos de prueba y desarrollo en menos servidores físicos y en un menor tiempo.

Fiabilidad

La fiabilidad se obtiene gracias al aislamiento de los errores software en cada máquina virtual, la posibilidad de reubicar máquinas virtuales en otras máquinas físicas cuando se producen errores hardware y la posibilidad de crear máquinas virtuales redundantes o ante un fallo.

Seguridad

El aislamiento entre máquinas virtuales también proporciona seguridad. Es muy difícil poder minimizar el impacto que tiene el mal funcionamiento de una aplicación sobre el resto de aplicaciones en un entorno no virtualizado. En cambio, una máquina virtual comprometida no altera el funcionamiento de las otras máquinas virtuales ni tampoco penaliza el rendimiento de las mismas.

La consolidación de las cargas de trabajo utilizando virtualización, permite asignar diferentes niveles de seguridad a cada partición en lugar de asignar un único nivel de seguridad a todo el servidor. Por ejemplo, en un servidor no virtualizado, la cuenta de administrador permite acceder a todas las aplicaciones del sistema, en cambio en un entorno virtualizado, cada partición, que contiene en general una aplicación, puede tener un cuenta de administrador diferente.

Bibliografía

[BMAR] Distributed Virtualization forNet-Centric Operations Draft. Bob MarcusApproved for Public Release Distribution Unlimited NCOIC-STKOUTRCH-PLEN-2007-DEC. http://home.comcast.net/~bobmarcus1/

[TAN08] Modern Operating Systems, 3/E, Andrew S. Tanenbaum, ISBN-10: 0136006639. Publisher: Prentice Hall, 2008.

[JAVA] http://www.java.com/es/

[CLI] http://msdn.microsoft.com/en-us/library/xey702bw.aspx

[DBOX] www.dosbox.com/

[WINE] *www.winehq.org/*

[CYG] *www.cygwin.com/*

[VBOX] *www.virtualbox.org*

[VMSX] http://www.vmware.com/products/vsphere/esxi-and-esx/index.html

[VMPL] http://www.vmware.com/products/player/

[VMSV] http://www.vmware.com/products/server/

[SCON] http://en.wikipedia.org/wiki/Solaris_Containers

[JAIL] http://www.freebsd.org/doc/handbook/jails.html

[CHRO] http://es.wikipedia.org/wiki/Chroot

[XEN] http://www.xen.org/

[LGUE] http://lguest.ozlabs.org/

[L4] http://en.wikipedia.org/wiki/L4_microkernel_family

3. Virtualización clásica y la arquitectura x86

El presente capítulo comienza con la interesante historia desarrollada en la década de 1960, cuando las nuevas técnicas de los sistemas operativos estaban en pleno desarrollo. Los protagonistas, principalmente empleados por IBM, fundaron las bases para las tecnologías que se usarían por, al menos, 50 años más. Unos años después, se presentaron los requerimientos de Popek y Goldberg, los cuales, a pesar de haber sido publicado a mediados de 1970, continúan en vigencia, por lo que son explicados a continuación. Además, se discuten en detalle las técnicas para implementar la virtualización clásica, algunas de las cuales aún se utilizan.

La histórica arquitectura x86 no se considera como una arquitectura clásicamente virtualizable y los motivos de esta afirmación se detallan en este capítulo. Sin embargo, la virtualización de esta arquitectura es posible, refinando las técnicas de la virtualización clásica, como se explica sobre el final de este capítulo.

El nacimiento de la virtualización

A mediados de la década de 1960, las máquinas virtuales surgieron como solución a la intensa investigación que realizaban en ese entonces los investigadores de IBM y científicos de la Universidad de Cambridge para resolver los problemas que comenzaban a ver la luz en relación al concepto de tiempo compartido (*time sharing)*. Uno de los diseños de *mainframe* del momento fue el 370/67, con un hipervisor llamado CP/67 que luego se convertiría en VM/370 y evolucionaría en la familia de VMMs para mainframes de IBM [VM]. En esta sección estudiamos esta evolución con más detalle.

Podemos interpretar la idea de tiempo compartido como el primer intento de virtualización. Hacia 1960, Christopher Strachey, el primer profesor de informática de la Universidad de Oxford y líder del Grupo de Investigación en Programación, utilizó el término en su artículo denominado "*Time sharing in large, fast computers"* [STRACH]. Gracias a este método el profesor Strachey implementó la técnica de multiprogramación, que permite la ejecución de procesos concurrentes en un mismo procesador.

Poco tiempo después, en 1961, el Centro de Computación del MIT desarrolló uno de los primeros sistemas operativos de tiempo compartido, llamado Sistema Compatible de Tiempo Compartido (CTSS, Compatible Time-Sharing System). Aunque el CTSS no fue un sistema operativo exitoso, tuvo una gran influencia al mostrar que el tiempo compartido era una técnica viable. El CTSS es considerado el primero de los sistemas operativos de tiempo compartido modernos ya que influye en el desarrollo, entre otros, de:

- IBM M44/44X.
- CP-40/CMS, que deriva en z/VM.
- TSS/360
- MULTICS, que influye fuertemente en la familia UNIX, incluyendo a GNU/Linux
- CP/M, que influye fuertemente en 86-DOS, el cual deriva en Microsoft Windows.

En 1964 el Centro Científico de Cambridge de IBM, liderado por Robert Creasy, comenzó el desarrollo del programa de control CP-40 (Control Program 40) y del monitor CMS (Cambridge Monitor System). El CP-40 fue el primer sistema operativo que implementaba virtualización completa, y permitía emular simultáneamente hasta 14 *pseudo-máquinas* (múltiples instancias del CMS), más tarde llamadas máquinas virtuales, ejecutándose en un estado particular llamado *problem state*.

Cuando una máquina virtual ejecutaba una instrucción privilegiada (por ejemplo, una operación de E/S) o utilizaba una dirección de memoria inválida, se producía una excepción que era capturada por el programa de control, y que se ejecutaba en otro estado llamado *supervisor state*, para simular el comportamiento adecuado.

Un año después, en 1965, el Centro de Investigación Thomas J. Watson de IBM implementó una computadora experimental, el IBM M44/44X, basada en el IBM 7044 (M44), con varias máquinas 7044 virtuales (44Xs) simuladas, usando paginación, memoria virtual y multiprogramación.

El M44/44X no implementaba una simulación completa del hardware subyacente, sino que aplicaba la idea de virtualización parcial y de esta forma se demostraba que el concepto de virtualización no generaba necesariamente una gran sobrecarga de recursos (overhead).

IBM anuncia el System/360 modelo 67 (S/360-67) y el sistema operativo de tiempo compartido TSS/360 en sus famosas cartas azules (mecanismo de IBM para anunciar nuevos productos). El TSS/360 implementa memoria virtual y máquinas virtuales, pero es cancelado en 1971 por sus problemas de rendimiento, fiabilidad e incompatibilidad con el sistema operativo de proceso por lotes OS/360.

En 1966, paralelamente a TSS/360, el Centro Científico de Cambridge de IBM comenzó con la conversión del CP-40 y el CMS para ejecutarlos sobre el S/360-67. El CP-67 es una significativa re-implementación del CP-40 y es la primera implementación libremente disponible de la Arquitectura de Máquina virtual conocida como VM.

Corre 1968 e IBM publica en su IBM Type-III Library (colección de código fuente no soportada por IBM, contribuida por clientes y personal de IBM) la primera versión de CP/CMS. National CSS (NCSS), una compañía que exploraba la idea de ofrecer servicios de tiempo compartido, aprovecha la disponibilidad de CP/CMS para iniciar la implementación de VP/CSS (un derivado de CP/CMS) ya que el rendimiento de CP/CMS no era rentable para sus planes de comerciales.

Durante el año 1970 IBM empieza a desarrollar CP-370/CMS, una completa re-implementación del CP-67/CMS para su nueva serie System/370 (S/370). Y en 1972, IBM anuncia el primer sistema operativo de máquina virtual de la familia VM (VM/CMS), el VM/370 (basado en CP-370/CMS) y destinado al System/370 con hardware de memoria virtual. El VM/370 se basa en dos componentes: CP y CMS (ahora llamado Conversational Monitor System). La función más importante del nuevo CP es la habilidad de ejecutar una máquina virtual dentro de otra máquina virtual.

NCSS porta VP/CSS a la serie System/370. VP/CSS mejora el rendimiento del CSS utilizando técnicas de paravirtualización, a través de llamadas directas al hipervisor, en lugar de simular las operaciones de bajo de nivel de los comandos de E/S.

Desde 1976 y hasta 1987, la revolución de las computadoras personales (Apple II, Atari 400/800, Commodore VIC-20, IBM PC, ZX Spectrum, Commodore 64, Apple Macintosh, Atari ST, Commodore Amiga) provocó que la industria pierda interés en los sistemas operativos optimizados para mainframes. No obstante, IBM continuó el desarrollo de su familia VM hasta el día de hoy.

Requerimientos de virtualización de Popek y Goldberg

Los Requerimientos de virtualización de Popek y Goldberg son un conjunto de condiciones suficientes para que una arquitectura de computadoras soporte eficientemente la virtualización. Fueron elaborados por Gerald J. Popek y Robert P. Goldberg en su artículo de 1974 *“Formal Requirements for Virtualizable Third Generation Architectures”* [POPEK].

Aunque los requisitos se derivan de suposiciones simplificadas, todavía constituyen una manera eficaz de determinar si una arquitectura soporta eficientemente la virtualización, y

proporcionan líneas maestras para el diseño de arquitecturas virtualizables. El artículo presenta tres propiedades de interés cuando se analiza el entorno creado por un VMM:

- **Equivalencia / Fidelidad:** un programa ejecutándose sobre un VMM debería que tener un comportamiento idéntico al que tendría ejecutándose directamente sobre el hardware subyacente.

- **Control de recursos / Seguridad:** El VMM tiene que controlar completamente y en todo momento el conjunto de recursos virtualizados que proporciona a cada guest.

- **Eficiencia / Performance:** Una fracción estadísticamente dominante de instrucciones tienen que ser ejecutadas sin la intervención del VMM, o en otras palabras, directamente por el hardware.

Según la terminología de Popek y Gordberg, un VMM debe presentar cada una de las tres propiedades. Según literatura alternativa [SMITH], se asume típicamente que los VMM satisfacen la fidelidad y seguridad, y aquellos VMMs que adicionalmente satisfacen la propiedad de performance son llamados VMMs eficientes.

Popek y Goldberg describen las características que el Conjunto de Instrucciones de la Arquitectura (ISA, por las siglas de Instruction Set Architecture) de la máquina física debe poseer para poder ejecutar VMMs que tengan las propiedades listadas arriba. Su análisis deriva tales características usando un modelo de "arquitectura de tercera generación" (por ejemplo, IBM 360, Honeywell 6000, DEC PDP-10) que, de cualquier forma, es lo suficientemente general como para ser extendido a las máquinas modernas. Este modelo incluye un procesador que opera tanto en modo usuario como en modo privilegiado y tiene acceso a una memoria lineal uniformemente direccionable. Se asume que un subconjunto del ISA está disponible sólo cuando el procesador se encuentra en modo privilegiado y que la memoria es direccionada en términos relativos a un registro de relocación. Las interrupciones y la Entrada/Salida no están modeladas.

Teoremas de Virtualización

Para deducir sus teoremas de virtualización, que dan condiciones suficientes (pero no necesarias) para la virtualización, Popek y Goldberg introducen una clasificación de las instrucciones de un ISA en tres grupos diferentes:

- **Instrucciones privilegiadas:** instrucciones que provocan una excepción al ser ejecutadas en modo usuario y no la provocan al ser ejecutadas desde modo supervisor.

- **Instrucciones sensibles de control:** instrucciones que permiten cambiar la configuración actual de los recursos hardware.

- **Instrucciones sensibles de comportamiento:** instrucciones cuyo comportamiento o resultado dependen de la configuración del sistema. Por esto mismo, permiten conocer el estado de la configuración actual de los recursos hardware.

El resultado principal de Popek y Goldberg puede ser expresado de la siguiente forma:

Teorema 1: Para cualquier computadora convencional de la tercera generación, puede construirse un VMM si el conjunto de instrucciones sensibles (tanto de control como de comportamiento) para esa computadora es un subconjunto del de las instrucciones privilegiadas.

Intuitivamente, el teorema establece que para construir un VMM es suficiente con que todas las instrucciones que podrían afectar al correcto funcionamiento del VMM (instrucciones sensibles) siempre generen una excepción y pasen el control al VMM. Eso garantiza la propiedad de control de los recursos / seguridad. En cambio, las instrucciones no privilegiadas deben ejecutarse nativamente (es decir, eficientemente).

Este teorema también provee una técnica simple para implementar un VMM, llamada virtualización por *'trap and emulate' (excepción y emulación),* más recientemente llamada virtualización clásica: ya que todas las instrucciones sensibles se comportan correctamente, todo lo que un VMM tiene que hacer es 'atrapar' y emular cada una de ellas. Para una explicación detallada de los mecanismos de interrupción y excepción, y de las diferencias entre términos similares, como trap[1] y excepción, consultar el Apéndice A de este trabajo.

Un problema relacionado es el de derivar condiciones suficientes para la virtualización recursiva, es decir, las condiciones bajo las cuales puede construirse un VMM capaz de ejecutar una copia de si mismo. Entonces se postula el segundo teorema:

Teorema 2. Una computadora convencional de la tercera generación es recursivamente virtualizable si

1. es virtualizable y
2. puede construirse un VMM sin dependencias de tiempo para ella

Técnicas de la virtualización clásica

Una aclaración importante es que en este trabajo, al igual que en muchos otros, se usa el término 'clásicamente virtualizable' para describir a una arquitectura que puede ser virtualizada puramente con la técnica '*trap and emulate*'.

En este sentido, x86 no es clásicamente virtualizable, pero es virtualizable, por el criterio de Popek y Goldberg, usando las técnicas descriptas más adelante. En esta sección revisamos las ideas más importantes usadas en las implementaciones de los VMM clásicos: *deprivileging*, estructuras *shadow* y *trazas*. Como se anticipó, se preservan algunos términos en inglés por considerarse universales en la mayor parte de los textos consultados, sin importar su idioma de origen.

Deprivileging

En una arquitectura clásicamente virtualizable, todas las instrucciones que leen o escriben estado privilegiado pueden generar un trap cuando se ejecutan en un contexto no privilegiado. A veces, los traps ocurren por la clase de la instrucción (por ejemplo, una instrucción OUT), y otras veces los traps suceden como resultado de la protección por parte del VMM de ciertas estructuras en memoria que la instrucción intenta acceder (por ejemplo, el rango de direcciones de un dispositivo de E/S mapeado en memoria).

Un VMM clásico ejecuta directamente sobre el hardware a los sistemas operativos guest, pero en un nivel de privilegios reducido. El VMM intercepta los traps del guest con menos privilegios, y emula, sobre el estado de la máquina virtual, la instrucción generadora del trap. Como se observa en la figura 3.1, el VMM en un entorno virtualizado reemplaza a un sistema operativo

[1] Se ha decidido no reemplazar el término anglosajón 'trap' por su usual traducción 'trampa'. Sin embargo, tampoco se reemplaza el término 'trap' por el de de excepción ya que en teoría de sistemas operativos sus significados son diferentes. Por lo tanto, a continuación se aceptará el uso de trap como una subclase específica de excepción (o interrupción por software). Más detalles pueden encontrarse en el Apéndice A.

nativo y de esta forma logra la capacidad de ejecutar instancias no privilegiadas de los guests protegiéndose de ellos y protegiéndolos entre ellos a través del mecanismo de excepciones.

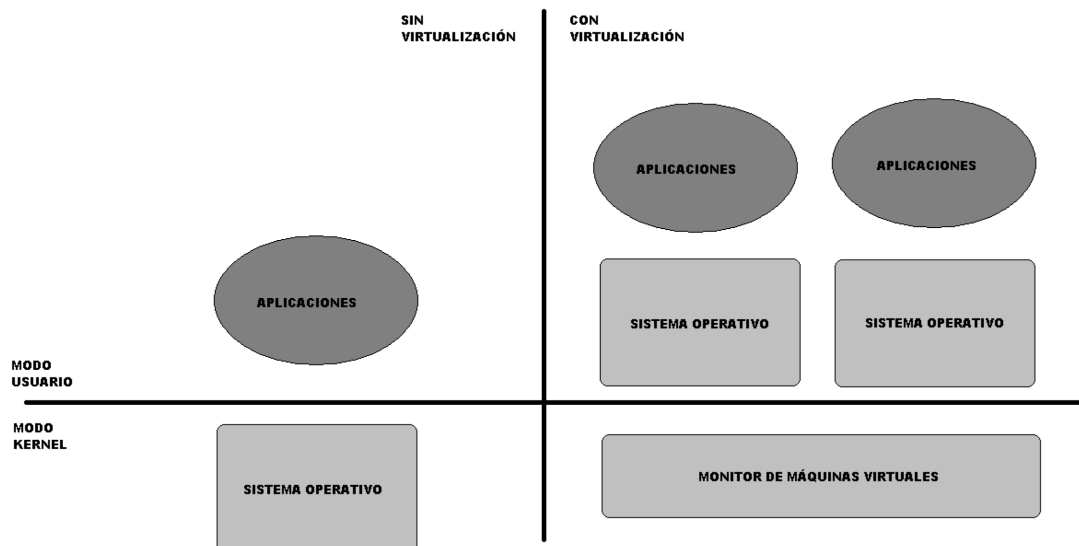


Figura 3.1 - Virtualización

Estructuras primarias y estructuras *shadow*

Por definición, el estado privilegiado de un sistema virtual difiere de aquel del hardware subyacente. La función básica del VMM es proveer un entorno de ejecución que cumpla con las expectativas del guest a pesar de aquella diferencia. Para lograr esto, el VMM deriva estructuras secundarias (más conocidas como estructuras shadow) de las estructuras primarias del guest.

El estado privilegiado contenido en registros de la CPU (on-CPU), como el registro puntero de las tablas de páginas o el registro de estado del procesador, se puede manejar trivialmente: el VMM mantiene una imagen de los registros del guest y actúa sobre esa imagen al emular una instrucción que generó un trap en el guest.

Sin embargo, los datos privilegiados no contenidos en registros de la CPU (off-CPU), como las tablas de páginas, pueden residir en memoria. En este caso, los accesos del guest al estado privilegiado pueden no coincidir naturalmente con instrucciones que generen traps. Por ejemplo, las entradas de tablas de páginas (PTE, por las siglas de Page Table Entry) del guest se consideran estado privilegiado debido a que codifican mapeos a memoria física y permisos sobre páginas físicas de esa memoria. Los accesos del guest a este estado no es acompañado por traps automáticas: cada referencia del guest a su memoria virtual depende de los permisos y mapeos codificados en el PTE correspondiente.

Tal estado privilegiado en memoria puede ser modificado por cualquier instrucción STORE en el flujo de instrucciones del guest, o incluso modificado de manera implícita como un efecto secundario de una operación de E/S DMA. Los dispositivos de E/S mapeados en memoria presentan un problema similar: lecturas y escrituras a esta información privilegiada puede originarse desde cualquier operación a memoria en el flujo de instrucciones del guest.

Por ejemplo, en un sistema nativo, el sistema operativo mantiene un mapeo de los Números de Páginas Lógicas (LPN, por las siglas de Logical Page Number) a los números de páginas físicas (PPN, por las siglas de Physical Page Number) en estructuras de tablas de páginas.

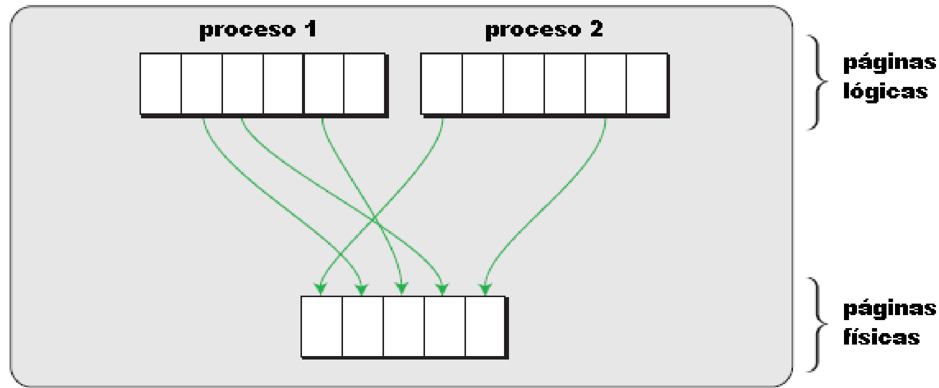


Figura 3.2 - Paginación

Cuando se accede a una dirección lógica, el hardware recorre estas tablas de páginas para determinar la dirección física correspondiente. Para acelerar los accesos a memoria, el hardware copia temporalmente los mapeos más recientemente usados en una memoria caché llamada Translation Lookaside Buffer (TLB).

En un sistema virtualizado, el sistema operativo guest mantiene tablas de páginas, igual que un sistema operativo nativo, pero adicionalmente el VMM mantiene un mapeo de PPNs a números de páginas de máquina (MPNs). En el caso de utilizar estructuras shadow (en este caso también conocido como sistema de MMU por software) el VMM mantiene los mapeos de PPN a MPN en sus estructuras de datos internas y almacena los mapeos LPN a MPN en tablas de páginas shadow que son expuestas al hardware nativo para ser usadas con el guest.

La figura 3.3 muestra una primera asociación con líneas verdes entre páginas lógicas del guest a páginas físicas del guest, las cuales en realidad son páginas virtuales del host. Las líneas rojas mapean las páginas físicas del guest (virtuales del host) a páginas de máquina (del host). Las líneas naranjas son las asociaciones representadas por las estructuras shadow expuestas al hardware y guardadas en la TLB luego de un acceso.

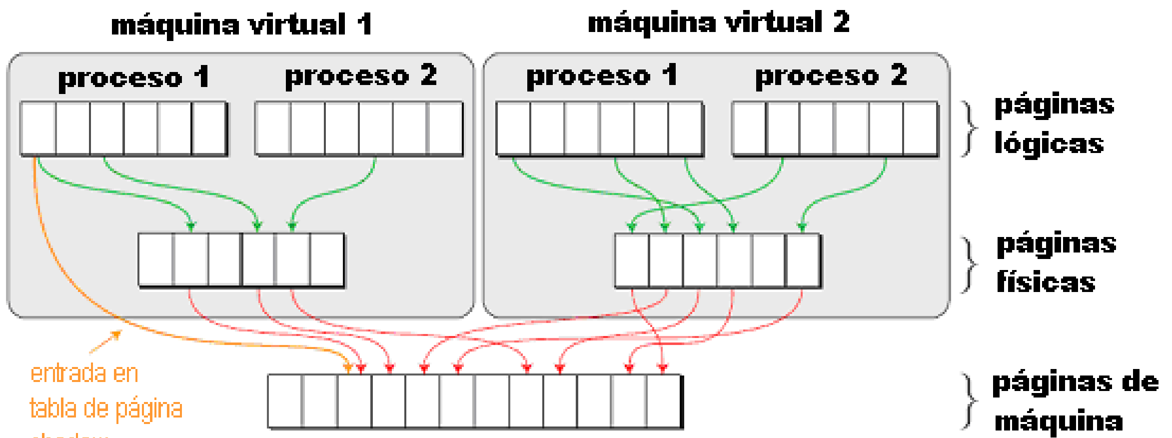


Figura 3.3 – Paginación Shadow

Trazas de memoria

Para mantener la coherencia de las estructuras shadow, los VMMs típicamente usan mecanismos por hardware de protección de páginas para interceptar accesos a las estructuras primarias en memoria. Por ejemplo, los PTEs guest para los cuales se han construidos PTEs shadows pueden ser protegidos contra escritura. Los dispositivos mapeados en memoria generalmente deben ser protegidos tanto contra lectura como contra escritura. Esta técnica basada en protección de página es conocida como *traceado (tracing)*. Los VMMs clásicos manejan un fallo de traza de manera similar a un fallo por instrucción privilegiada: decodifican la instrucción del guest que generó el trap, emulan su efecto sobre la estructura primaria y propagan el cambio a la estructura shadow.

Ejemplo de trazado: tablas de páginas de x86

Para proteger el host de los accesos a memoria del guest, los VMMs típicamente construyen tablas de páginas shadow con las cuales ejecutan el guest. La arquitectura x86 especifica tablas de páginas jerárquicas recorridas por el hardware, que pueden tener 2, 3 o 4 niveles.

El VMM administra sus tablas de páginas shadow como una caché de las tablas de páginas del guest. Cuando el guest accede a regiones de su espacio virtual de direcciones no tocadas previamente, los fallos de página del hardware transfieren el control al VMM. El VMM distingue los fallos de páginas reales de aquellos causados por violaciones de la política de protección codificada en el PTE guest, y de los fallos de páginas *escondidos* causados por fallos en las tablas de páginas shadow. Los fallos reales son reenviados al guest; los fallos escondidos hacen que el VMM construya un PTE shadow apropiado y que continúe con la ejecución del guest. El fallo se denomina escondido porque no tiene efectos visibles por el guest.

El VMM usa trazas para prevenir que sus PTEs shadow se vuelvan incoherentes con respecto a los PTEs del guest. Los fallos de traza resultantes son una fuente importante de overhead.

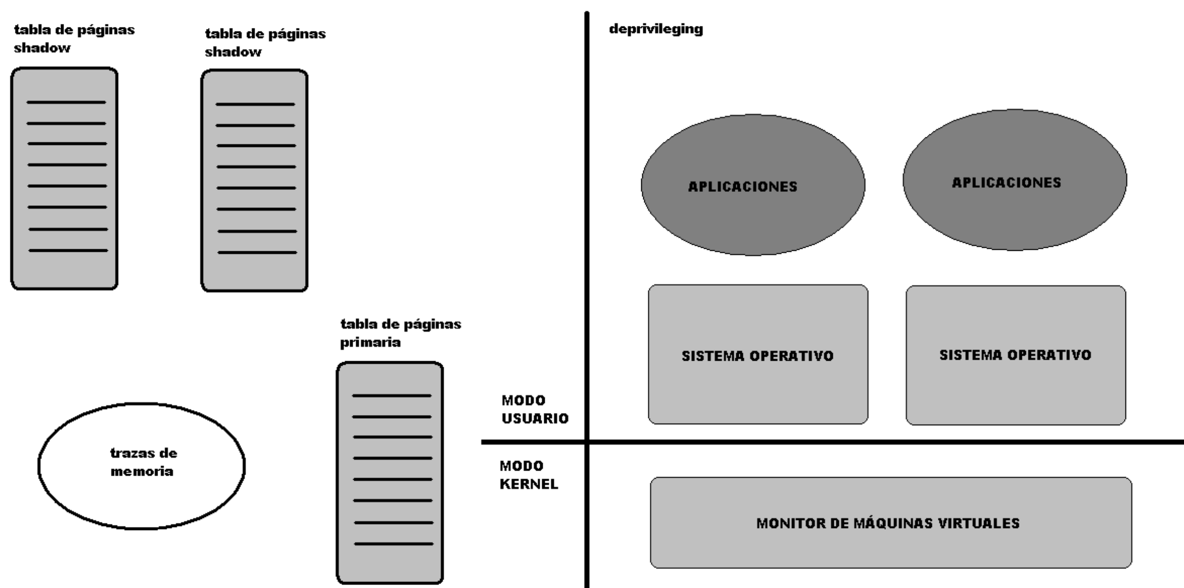


Figura 3.4 - Técnicas utilizadas en la virtualización clásica

VMMs para x86, los obstáculos de una arquitectura que no es clásicamente virtualizable

La arquitectura x86 proporciona un ISA no virtualizable ya que contiene instrucciones de control que, ejecutadas en modo usuario, no provocan una interrupción o excepción y, por lo tanto, el VMM no tiene oportunidad de emular su comportamiento.

Para cumplir con la primera condición de Popek y Goldberg se suele penalizar la tercera condición, es decir, se sacrifica la eficiencia para obtener equivalencia. Existen dos aproximaciones por software para conseguir equivalencia en x86: traducción binaria dinámica (re-compilación del código binario para reemplazar las instrucciones conflictivas) y paravirtualización (modificación del código fuente de los sistemas operativos guest).

Afortunadamente, a partir del año 2006 los principales desarrolladores de procesadores x86 incluyeron el soporte necesario en sus modelos para permitir virtualización clásica respetando los requerimientos de Popek y Goldberg. Estas extensiones a la arquitectura x86 se estudian en el capítulo 7.

En esta sección, se investigan los obstáculos que tuvieron que resolver los diseñadores de VMMs para x86 durante las últimas dos décadas para ofrecer soluciones de virtualización en la arquitectura de cómputo más popular. El ISA de IA-32 contiene 17 instrucciones sensibles no privilegiadas [PENT]. Pueden ser clasificadas en dos grupos:

- Instrucciones sensibles de registros: leen o cambian registros sensibles y/o locaciones de memoria tales como un registro de reloj o registros de interrupciones:
  - SGDT, SIDT, SLDT
  - SMSW
  - PUSHF, POPF

- Instrucciones de protección del sistema: referencian al sistema de protección de memoria o al sistema de relocación de direcciones:
  - LAR, LSL, VERR, VERW
  - POP
  - PUSH
  - CALL, JMP, INT n, RET
  - STR
  - MOV

Desafíos de la virtualización de la arquitectura x86

Ignorando los antiguos modos *real* y *virtual 8086* heredados de la arquitectura x86, los modos protegidos de 32 y 64 bits no fueron clásicamente virtualizables hasta la aparición de las extensiones por hardware para la virtualización.

Los procesadores x86 proveen protección basada en el concepto de niveles o anillos (rings) de privilegios identificados con 2 bits, usando el nivel 0 para el software más privilegiado y el nivel 3 para el menos privilegiado. El nivel de privilegios determina si las instrucciones privilegiadas, que controlan la funcionalidad básica de la CPU, pueden ejecutarse sin generar una excepción; también controlan el acceso al espacio de direcciones basándose en la configuración de las tablas de páginas y los registros de segmento.

La mayor parte del software x86 utiliza sólo los niveles de privilegios 0 y 3. Para que un sistema operativo pueda controlar la CPU, algunos de sus componentes deben ejecutarse con nivel de privilegios 0. Como un VMM no puede permitir que un sistema operativo guest tenga tal control, un sistema operativo guest no puede ejecutarse en el nivel 0, Por lo tanto, los VMMs basados

en x86 deben usar la ya mencionada técnica de *ring deprivileging*, ejecutando el guest en el nivel de privilegios 1 (el modelo 0/1/3) o bien en el nivel de privilegios 3 (el modelo 0/3/3).

Sin embargo, estas modificaciones no suceden sin consecuencias importantes. A continuación se analizan los problemas concretos y/o potenciales problemas que se derivan de aplicar dicha técnica en la arquitectura x86.

Fuga de información de privilegios

Se refiere al problema que surge cuando el software es ejecutado en un nivel de privilegios distinto para el cual fue escrito. Un ejemplo en x86 es la instrucción PUSH (que apila sus operandos en la pila) cuando se ejecuta con el registro CS (parte del cual almacena el nivel de privilegios actual). Un sistema operativo guest podría fácilmente determinar que no está ejecutándose en el nivel de privilegios 0, violando el principio de fidelidad de Popek y Goldberg.

Compresión del espacio de direccionamiento

Los sistemas operativos esperan tener acceso al espacio de direcciones virtuales completo del procesador, conocido como el espacio de direcciones lineales en x86, sin embargo, un VMM debe reservar para si mismo una porción del espacio virtual de direcciones del guest. El VMM podría ejecutarse enteramente dentro del espacio virtual de direcciones del guest, lo cual le permitiría fácil acceso a los datos del guest, aunque las instrucciones y estructuras de datos del VMM podrían ocupar una cantidad sustancial del espacio de direcciones virtuales del guest. Alternativamente, el VMM podría correr en un espacio de direcciones separado, pero incluso en ese caso el VMM debe usar una cantidad de espacio de direcciones virtuales del guest para las estructuras de control que manejan las transiciones entre el software del guest y el VMM (para x86, estas estructuras incluyen el IDT y el GDT, los cuales residen en el espacio de direcciones lineales).

El VMM debe prevenir el acceso del guest a esas porciones del espacio de direcciones virtuales del mismo guest que el VMM está usando. De lo contrario, la integridad del VMM podría ser comprometida si el guest pudiera escribir esas porciones de memoria o el guest podría detectar que está siendo ejecutado en una máquina virtual si pudiera leerlas. Los intentos del guest de acceder a estas porciones del espacio de direcciones deben generar transiciones al VMM, el cual puede emular los accesos o soportarlos de otra forma. El término compresión del espacio de direccionamiento se refiere a los desafíos de proteger estas porciones del espacio de direcciones virtuales y dar soporte a los accesos del guest al mismo.

Acceso al estado privilegiado que no genera excepciones

La protección basada en privilegios previene que el software no privilegiado acceda a ciertos componentes del estado de la CPU. En la mayoría de los casos, los intentos de acceso resultan en excepciones, permitiendo que un VMM emule la instrucción del guest deseada. Sin embargo, la arquitectura x86 incluye instrucciones que acceden a cierto estado privilegiado y no generan fallos cuando se ejecutan con privilegio insuficiente. Por ejemplo, los registros de x86 GDTR, IDTR, LDTR y TR contienen punteros a estructuras de datos que controlan la operación de la CPU. El software puede ejecutar instrucciones que escriban (LOAD) estos registros (LGDT, LIDT, LLDT, y LTR) sólo en el nivel de privilegio 0. Sin embargo, el software puede ejecutar las instrucción que leen (STORE) desde estos registros (SGDT, SIDT, SLDT, y STR) en cualquier nivel de privilegios. Si el VMM mantiene estos registros con valores inesperados, un sistema operativo guest usando estas instrucciones podría determinar que no tiene el control total sobre la CPU.

Interferencia con las facilidades para *system calls* rápidas

La reducción de privilegios del guest puede interferir con la efectividad de las facilidades incluidas en la arquitectura x86 para acelerar el manejo de transiciones hacia el kernel de un sistema operativo. Las instrucciones SYSENTER y SYSEXIT soportan *system calls* de baja

latencia. SYSENTER siempre produce una transición al nivel de privilegios 0, y SYSEXIT, fallará si se ejecuta fuera de este nivel de privilegios. Por lo tanto la técnica de *ring deprivileging* tiene las siguientes implicaciones:

Las ejecuciones de SYSENTER por parte de las aplicaciones del guest causarán una transición al VMM y no al sistema operativo guest, como se supone. Las ejecuciones de SYSEXIT por un kernel guest causarán un fault al VMM. Por lo tanto, el VMM debe emular cada ejecución de SYSENTER y SYSEXIT que se realicen en el guest.

Virtualización de Interrupciones

Proveer soporte para las interrupciones externas, especialmente en lo que se refiere al enmascaramiento de interrupciones, presenta algunos desafíos específicos para el diseño de un VMM. La arquitectura x86 provee mecanismos para enmascarar interrupciones externas, previniendo que sean entregadas cuando el sistema operativo no está listo para ellas. Se usa el flag de interrupciones (IF) en el registro EFLAGS para controlar el enmascaramiento de interrupciones.

Un VMM probablemente administrará las interrupciones externas y denegará al guest la habilidad de controlar el enmascaramiento de interrupciones. Los mecanismos de protección existentes permiten tal negación de control al asegurar que los intentos del guest de controlar el enmascaramiento de interrupciones generarán un fallo. Tal mecanismo basado en excepciones es inapropiado porque los sistemas operativos suelen enmascarar y desenmascarar frecuentemente las interrupciones. Interceptar cada intento de un guest de realizar tales operaciones podría afectar significativamente la performance del sistema.

Incluso si fuera posible prevenir las modificaciones del guest al enmascaramiento de interrupciones sin interceptar cada intento, los desafíos se mantendrían cuando un VMM debe entregar una interrupción virtual a un guest. Una interrupción virtual debería ser entregada sólo cuando el guest ha desenmascarado sus interrupciones. Para entregar las interrupciones virtuales a tiempo, por lo tanto, un VMM debería interceptar algunos, pero no todos, los intentos de un guest de modificar el enmascaramiento de interrupciones. Tal lógica puede complicar el diseño de un VMM.

Compresión de anillos de privilegios

En esta técnica se utilizan dos mecanismos de traducción basados en privilegios para proteger al VMM del software del guest. La arquitectura x86 incluye los dos mecanismos: límites de segmentos y paginación. Como la segmentación no fue implementada en el modo 64-bit de la extensión x86-64, en este modo sólo puede utilizarse la paginación. Y, como la paginación en x86 no distingue entre los niveles de privilegios 0-2 (son todos privilegiados), el sistema operativo guest debería, en principio, ejecutarse en el nivel de privilegios 3. Por lo tanto, el sistema operativo guest correrá en el mismo nivel de privilegios que las aplicaciones del guest y no estará protegido de ellas.

Acceso al estado escondido

Algunos componentes del estado de una CPU x86 no están representados en ningún registro accesible por el software. Los ejemplos incluyen las cachés de descriptores para los registros de segmento. Una lectura a un registro de segmento copia el descriptor referenciado (desde la GDT o la LDT) en su cache, la cual no es modificada si el software luego escribe en las tablas de descriptores. x86 no provee mecanismos para salvar y recuperar estos componentes escondidos del contexto de un guest cuando se intercambia entre contextos de máquinas virtuales o para preservarlos al ejecutar el VMM.

Refinamientos a la virtualización clásica y las nuevas técnicas de virtualización

La performance en la práctica

Los requerimientos de eficiencia en la definición de un VMM de Popek y Goldberg sólo se refieren a la ejecución de instrucciones no privilegiadas que deben ejecutarse nativamente. Esto es lo que distingue a un VMM por un lado de la clase más general de software de emulación de hardware por el otro.

Desafortunadamente, incluso en las arquitecturas que satisfacen los requerimientos de Popek y Goldberg la performance de una máquina virtual puede diferir significativamente de la del hardware real. En el System/370, fue la memoria virtual el principal problema de performance. Cuando el guest era un sistema operativo que implementaba memoria virtual por si mismo, incluso las instrucciones no privilegiadas podían experimentar grandes tiempos de ejecución debido a la necesidad de acceder a las tablas de traducción correspondientes: las tablas de páginas shadow. Los primeros experimentos realizados sobre el System/370 (que satisface los requerimientos formales del Teorema 1 mostraron que la performance de una máquina virtual podía ser tan bajo como sólo el 21% de la máquina nativa en algunos estudios comparativos.

Yendo más allá de Popek y Goldberg

El tipo de carga de trabajo impacta significativamente en la performance de la aproximación de la virtualización clásica, y es por esto que las condiciones para la virtualización expresadas en el Teorema 1 suelen ser suavizadas. Se han venido construyendo VMMs para ISAs no virtualizables (en el sentido de Popek y Goldberg) desde hace tiempo, ignorando las propiedades de fidelidad, seguridad o eficiencia.

Durante el primer boom de las máquinas virtuales, era común que tanto el VMM, el hardware y todos los sistemas operativos guest fueran producidos por una única empresa. Esta integración vertical permitió a los investigadores e ingenieros refinar la virtualización clásica usando dos aproximaciones ortogonales.

Una aproximación explotaba la flexibilidad en la interfaz VMM/sistema operativo guest. Los implementadores que tomaron esta aproximación modificaron el sistema operativo guest para proveer información de alto nivel al VMM. Esta aproximación relaja el requerimiento de fidelidad de Popek y Goldberg para proveer ganancias en la performance y, opcionalmente, para proveer funcionalidades más allá de la definición base de virtualización, tal como la comunicación eficiente entre máquinas virtuales.

La otra aproximación para refinar los VMMs clásicos explotó la flexibilidad en la interfaz hardware/VMM. La arquitectura System/370 de IBM introdujo la ejecución interpretativa (*interpretive execution*), un modo de ejecución del hardware para ejecutar sistemas operativos guests. El VMM codifica gran parte del estado privilegiado del guest en un formato definido por el hardware, luego ejecuta la instrucción SIE para comenzar la ejecución interpretativa (*Start Interpretive Execution*). Muchas operaciones del guest que generarían traps en un entorno menos privilegiado accedían directamente a los campos shadow en la ejecución interpretativa. Mientras que un VMM aún debía manejar algunos traps, SIE fue exitosa en reducir la frecuencia de traps en relación a un VMM tipo *trap and emulate* sin asistencia. Las asistencias fueron agregadas en varias etapas, y finalmente más de 100 extensiones asistían al VMM en los últimos modelos de System/370 [VM], prácticamente duplicando la performance de las máquinas virtuales.

Ambas aproximaciones tienen su herencia intelectual en el boom actual de la virtualización. El intento de explotar la flexibilidad en la capa VMM/guest ha sido revivido bajo el nombre de paravirtualización. Mientras que los fabricantes de x86 están introduciendo facilidades de hardware inspirados por la ejecución interpretativa de IBM.

En este trabajo se estudian en profundidad las tres técnicas alternativas principales para manejar las instrucciones sensibles y privilegiadas para virtualizar la CPU de una arquitectura x86:

- traducción binaria
- paravirtualización
- virtualización asistida por hardware

La virtualización de esta arquitectura requiere el manejo correcto de las instrucciones críticas, es decir, las instrucciones sensibles pero no privilegiadas. La aproximación conocida como traducción binaria adopta técnicas comúnmente usadas en re-compilación dinámica: las instrucciones críticas son descubiertas en tiempo de ejecución y reemplazadas con una llamada explícita al VMM. Una aproximación diferente es la de paravirtualización la cual requiere que el sistema operativo guest sea modificado (portado) antes de ejecutarse en el entorno virtual.

Bibliografía

[VM] *VM and the VM Community: Past, Present, and Future*, revised 08/16/97 http://www.princeton.edu/~melinda/25paper.pdf

[STRACH] Christopher Strachey: Time sharing in large, fast computers. IFIP Congress 1959: 336-341.

[SMITH] Virtual Machines. Versatile Platforms for Systems and Processes. By Jim Smith & Ravi Nair . The Morgan Kaufmann Series in Computer Architecture and Design Series. ISBN: 978-1-55860-910-5.

[POPEK] Gerald J. Popek and Robert P. Goldberg (1974). «Formal Requirements for Virtualizable Third Generation Architectures». *Communications of the ACM* **17** (7): pp. 412 – 421. http://doi.acm.org/10.1145/361011.361073.

[PENT] Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. John Scott Robin, Cynthia E. Irvine. 9th USENIX Security Symposium Paper 2000. http://www.usenix.org/events/sec00/full_papers/robin/robin_html/

4. Traducción Binaria

Como se explicó en el capítulo anterior, la virtualización en arquitecturas x86 no es sencilla y requiere de técnicas complejas para ser implementada. Uno de los grupos pioneros en virtualización de x86 se originó en la Universidad de Stanford a fines de la década de 1990. En ese entonces la traducción binaria fue la principal opción adoptada por los implementadores.

El 9 de febrero de 1999 VMware introdujo el primer producto de virtualización x86, llamado *VMware Virtual Plataform*, basado en un trabajo de investigación anterior realizado por sus fundadores en Stanford.

VMware y otros programas de virtualización similares modifican dinámicamente las instrucciones del kernel del sistema operativo que pueden generar conflictos, por lo que pueden ejecutar cualquier sistema operativo virtualizado para x86, a costa de impactar en el rendimiento.

Kevin Lawton, paralelamente, desarrolló el proyecto Bochs, con funcionalidad similar, que comenzó siendo software privativo pero pasó a ser software libre cuando Mandriva compró el proyecto. Bochs se basa en un concepto de emulación.

Por su parte, Microsoft ofrece dos productos de virtualización basados en Windows, Microsoft Virtual PC y Microsoft Virtual Server, basados en tecnología creada en el 2004 por la empresa Connectix. Estos productos toman ideas de la recompilación dinámica.

De todos los productos existentes que implementan traducción binaria, los de VMware son los más ampliamente adoptados. Los ingenieros de esta empresa han participado activamente en los avances logrados en técnicas de virtualización por software, incluyendo las técnicas alternativas como la paravirtualización. Incluso, suelen ser actores importantes en las decisiones tomadas por la comunidad de software libre que implementa el kernel Linux, aunque sus productos posean licencias privativas. En el próximo capítulo se estudia en detalle la participación de VMware en el desarrollo del framework del kernel Linux para la paravirtualización.

En este capítulo se describen los conceptos detrás de la traducción binaria, siguiendo como ejemplo a los productos de VMware. Se explican términos asociados y las ventajas y desventajas de esta alternativa.

Definiciones previas

Los obstáculos semánticos de la virtualización de x86 pueden ser superados si el guest se ejecuta sobre un intérprete en lugar de ejecutarse directamente en una CPU física. El intérprete puede prevenir la fuga de estado privilegiado desde la CPU física hacia el guest, por ejemplo el valor del CPL, y puede implementar correctamente las instrucciones que no generan traps, como *popf*, referenciando el CPL virtual, sin modificar el CPL físico. En esencia, el intérprete separa el estado virtual (la VCPU) del estado físico (la CPU).

Sin embargo, aunque la interpretación asegura la Fidelidad y la Seguridad, dos de los criterios de Popek y Goldberg (ver capítulo anterior), falla al intentar lograr la propiedad de Performance: el ciclo de búsqueda-decodificación-ejecución del intérprete puede consumir cientos de instrucciones de la CPU física por instrucción del guest.

La traducción binaria, por otro lado, puede combinar la precisión semántica de la interpretación con una buena performance, ofreciendo un motor de ejecución que satisface todas las propiedades de Popek y Goldberg.

Un VMM construido sobre un traductor binario apropiado puede virtualizar la arquitectura x86 y es un VMM correcto de acuerdo al criterio de Popek y Goldberg.

El uso de traducción binaria como motor de ejecución de un VMM tiene un paralelismo cercano con otros trabajos en el área de sistemas, como las Máquinas Virtuales de Java (JVM, por Java Virtual Machine) que usan compiladores Just In Time (JIT) o simuladores de conjuntos de instrucciones, como veremos más adelante.

Interpretación y compilación dinámica

Un Simulador de Conjunto de Instrucciones (ISS, por sus siglas en inglés) es un modelo de simulación que imita el comportamiento de un mainframe o microprocesador leyendo instrucciones y manteniendo variables internas que representan el estado de los registros del procesador virtual. Es una metodología que puede usarse con diversas aplicaciones, por ejemplo para virtualizar una CPU o para monitorear la ejecución de instrucciones en código máquina en un entorno de prueba.

Tradicionalmente, los programas de computadora tienen dos modos de operación en tiempo de ejecución: por interpretación o por compilación estática. El código interpretado es traducido de una forma de representación a otra durante cada ejecución. Por otro lado el código compilado estáticamente se traduce a código máquina antes de la ejecución y esta traducción se realiza sólo una vez.

Un intérprete puede tener distintos modos de funcionamiento:

1. ejecutar el código fuente directamente
2. traducir el código fuente a una representación intermedia más eficiente, e inmediatamente ejecutarla
3. ejecutar código precompilado previamente en algún formato independiente de la máquina

En un sistema tipo *bytecode* [BYTE], el código fuente se traduce a una representación intermedia, conocida como bytecode, que no es el código máquina de ninguna arquitectura particular y es portable. Luego el bytecode puede ser interpretado por - o ejecutado en - una máquina virtual. Esta última variedad se conoce como intérprete de bytecode. El código precompilado en este caso es el 'código de máquina' para la máquina virtual que es implementada por el intérprete de bytecode.

La mayor desventaja de los intérpretes es que los programas corren más lentos que aquellos que se compilan. La diferencia puede ser significativa. Esto se debe a que el intérprete debe analizar cada sentencia en el código de entrada cada vez que se ésta ejecuta y luego realizar la acción deseada. Este análisis en tiempo de ejecución se conoce como 'sobrecarga por interpretación' (*interpretation overhead*) [INOV]. Los accesos a las variables también son más lentos en un intérprete porque el mapeo de identificadores a direcciones de memoria debe realizarse repetidamente en tiempo de ejecución, en lugar de ser estático.

Por otro lado, la compilación dinámica es un proceso usado para ganar performance durante la ejecución de un programa. Como el código de máquina emitido por un compilador dinámico se construye y se optimiza en tiempo de ejecución del programa, el uso de compilación dinámica posibilita ciertas optimizaciones que no están disponibles para programas compilados estáticamente, excepto a través de la duplicación de código o la metaprogramación [META].

La recompilación dinámica es una característica de algunos emuladores y máquinas virtuales con la que el sistema puede recompilar partes de un programa durante la ejecución, y es usada generalmente para producir código máquina más eficiente en base al entorno de ejecución.

En otros casos, un sistema puede emplear la recompilación dinámica como parte de una estrategia de optimización adaptativa para ejecutar representaciones intermedias de un

programa. La optimización adaptativa es una técnica complementaria en la que un intérprete perfila el programa en ejecución y traduce dinámicamente a código binario sus partes más frecuentemente ejecutadas.

A un nivel más bajo, la optimización adaptativa puede aprovechar el acceso a los datos del programa en ejecución y al estado de la CPU para optimizar bifurcaciones (saltos) y para usar expansiones *inline* [INLI] para reducir los saltos a funciones o los cambios de contexto.

Tanto la compilación dinámica como la optimización adaptativa fueron investigadas por primera vez en la década de 1980, apareciendo en lenguajes tales como *Smalltalk* [SMAL].

Traducción Binaria

La traducción binaria es la emulación de un conjunto de instrucciones por otro a través de traducción de código. Las secuencias de instrucciones se traducen del conjunto de instrucciones origen al conjunto de instrucciones destino. En algunos casos como en la simulación de un conjunto de instrucciones (ISS), el conjunto de instrucciones destino puede ser el mismo que el conjunto de instrucciones origen, o un subconjunto de él, como veremos más adelante. Los dos tipos principales son traducción binaria estática y traducción binaria dinámica

Traducción binaria estática

Un archivo ejecutable es traducido completamente a un ejecutable de la arquitectura destino. Esto es difícil de lograr correctamente, ya que no todo el código puede ser descubierto por el traductor. Por ejemplo, algunas partes del archivo ejecutable pueden ser alcanzables solamente a través de saltos indirectos (también conocidos como saltos computados), cuya dirección de destino se conoce sólo en tiempo de ejecución.

Traducción binaria dinámica

La traducción dinámica trabaja con secuencias cortas de código, típicamente en el orden de un único bloque básico, traduciendo y almacenando la secuencia resultante en una memoria caché. El código sólo es traducido a medida que es descubierto y las instrucciones de bifurcación son re-escritas para apuntar al código ya traducido y almacenado.

La traducción binaria dinámica difiere de la emulación simple al eliminar el mayor problema de performance que genera el ciclo lectura-decodificación-ejecución, pagando un costo de tiempo durante la traducción. Esta sacrificio de tiempo es amortizado cuando las secuencias de código traducidas se ejecutan múltiples veces.

Traductores dinámicos más avanzados emplean la recompilación dinámica, donde el código traducido es instrumentado para encontrar las porciones de código que son ejecutadas una gran cantidad de veces, para luego poder optimizarlas agresivamente. Esta técnica tiene reminiscencias de un compilador JIT y, de hecho, tales compiladores pueden ser vistos como traductores dinámicos de un conjunto de instrucciones virtual (el bytecode) a uno real. La técnica de compilación JIT es una forma de compilación dinámica o traducción dinámica y suele usarse como método para mejorar la performance en tiempo de ejecución de un programa. En general esta técnica compila la representación intermedia a código de máquina nativo en tiempo de ejecución. Es una aproximación híbrida en la cual la traducción ocurre continuamente, como con los intérpretes, pero se almacena el código traducido para minimizar la degradación de performance. También ofrece las ventajas de la compilación dinámica, como la optimización exacta para la plataforma. Varios entornos de ejecución (*runtimes*) modernos, como el framework de .NET y la mayoría de las implementaciones de Java utilizan la compilación JIT.

Traducción Binaria de VMware para x86

VMware no esperó a que Intel o AMD resolvieran los problemas de la arquitectura x86 para la virtualización y lanzó su propia solución al final del siglo pasado. Para superar los problemas de las instrucciones críticas de x86, VMware usa técnicas de traducción binaria aunque no es necesario traducir de un conjunto de instrucciones a otro, sino que se basa en un traductor de x86 a x86.

VMware traduce *al vuelo* el código binario que el kernel de un sistema operativo guest quiere ejecutar y almacena el código x86 adaptado en una Cache del Traductor (TC, por Translator Cache). Las aplicaciones de usuario no serán modificadas por el Traductor Binario (BT, por Binary Translator) de VMware ya que se asume que el código de usuario es seguro. Es decir, los procesos en modo de usuario se ejecutan directamente sobre la CPU, como si estuvieran corriendo nativamente, como muestra la figura 4.1.

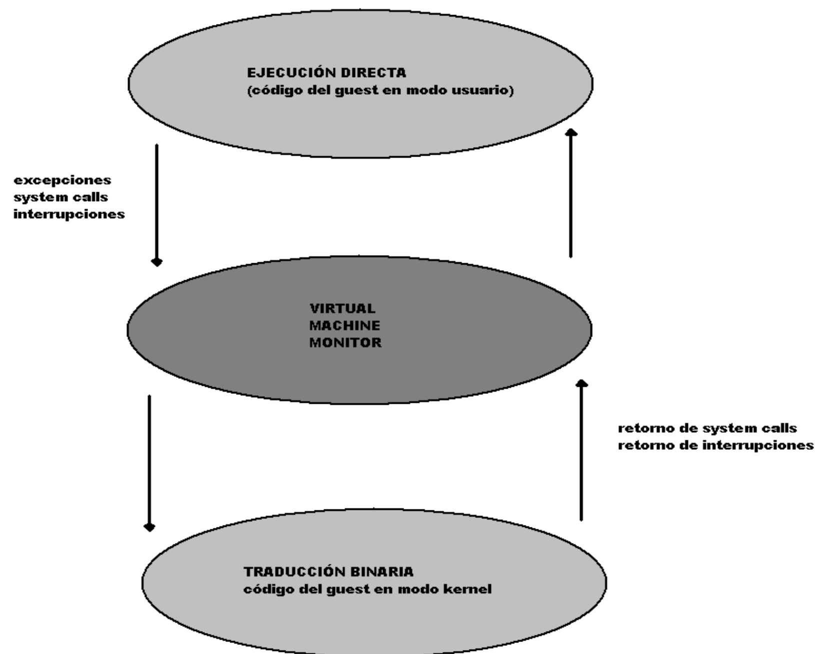


Figura 4.1 - La traducción binaria ocurre cuando se invoca el kernel del sistema operativo guest

La traducción binaria analiza el código que el kernel guest debería ejecutar en un momento dado en el tiempo y lo reemplaza con una versión más segura (virtualizada), bajo demanda. En este contexto, seguridad significa seguridad para los otros guests y para el VMM. Podría decirse que el kernel guest no se está ejecutando, sino que se carga en memoria como una entrada para el proceso de traducción; es entonces el kernel traducido por el traductor binario que se ejecutará en el nivel de privilegios 1.

En resumen, el traductor del VMM de VMware posee las siguientes características:

- **Binario:** La entrada es código binario x86, no código fuente o bytecode.
- **Dinámico:** La traducción sucede en tiempo de ejecución, intercalada con la ejecución del código generado.
- **Bajo demanda:** El código es traducido sólo cuando está a punto de ser ejecutado. Esta aproximación '*lazy*' evita el problema de distinguir código de datos en el binario del kernel.
- **A nivel de sistema:** El traductor no hace ninguna suposición acerca del código guest. Las reglas están definidas por el ISA x86, no por una ABI de más alto nivel.
- **Subconjunto:** La entrada al traductor es el conjunto de instrucciones x86 completo, incluyendo todas las instrucciones privilegiadas; la salida es un subconjunto seguro (formado en su mayoría por instrucciones no privilegiadas).
- **Adaptativo:** El código traducido es ajustado en respuesta a cambios en el comportamiento del guest para mejorar la eficiencia general.

Un ejemplo sencillo

Para ilustrar el proceso de traducción, se recorre a continuación un pequeño ejemplo obtenido de un artículo publicado por VMware [COMP]. Ya que las instrucciones privilegiadas son extrañas incluso en el código del kernel de un sistema operativo, la performance de un sistema de traducción binaria está fuertemente determinada por la traducción de instrucciones regulares, por lo que a continuación se muestra una prueba sencilla de primalidad:

```
int isPrime(int a){
      for(int i = 2; i < a; i++){
           if(a % i == 0)
                return 0;
      }
      return 1;
}
```

Compilando el código C en un binario de 64 bits, se obtiene

```
isPrime: mov %ecx, %edi ; %ecx = %edi ( parámetro formal a)
mov %esi, $2 ; i = 2
cmp %esi, %ecx ; es i >= a?
jge prime ; saltar si lo es
nexti: mov %eax, %ecx ; asigna %eax = a
cdq ; extensión de signo
idiv %esi ; a % i
test %edx, %edx ; el resto es cero?
jz notPrime ; saltar si lo es
inc %esi ; i++
cmp %esi, %ecx ; es i >= a?
jl nexti ; saltar si no lo es
prime: mov %eax, $1 ; valor de retorno en %eax
ret
notPrime: xor %eax, %eax ; %eax = 0
ret
```

Se invoca la función isPrime(49) en una máquina virtual, y se monitorea el código traducido. El código de arriba no es la entrada al traductor; en cambio su representación binaria lo es:

```
89 f9 be 02 00 00 00 39 ce 7d ...
```

El traductor lee la memoria del guest en la dirección indicada por el PC del guest, clasificando los bytes como prefijos, códigos de operación (*opcodes*) y operandos para producir objetos de Representación Intermedia (IR, por sus siglas en inglés). Cada objeto IR representa una instrucción del guest.

El traductor acumula objetos IR en una unidad de traducción (TU, por sus siglas en inglés), hasta que se acumulan doce instrucciones o se encuentra una instrucción de terminación (terminal) (usualmente control de flujo). El límite fijo permite el uso seguro de la pila para alocar todas las estructuras de datos sin riesgo de *overflow*; en la práctica raramente se alcanza el límite ya que el control de flujo tiende a terminar las TUs al cabo de algunas instrucciones. Por lo tanto, el caso común es que una TU sea un bloque básico (BB) [BLOC]. La primer TU en el ejemplo es:

```
isPrime: mov %ecx, %edi
mov %esi, $2
cmp %esi, %ecx
jge prime
```

Al traducir de x86 a un subconjunto de x86, la mayoría del código puede ser traducido a un código idéntico, operación que a será denominada como IDENT. Las primeras tres instrucciones del código de arriba son IDENT.

jge debe ser NO-IDENT porque la traducción no preserva el esquema del código. En cambio, se convierte en dos ramas alternativas de invocación al traductor, una para cada sucesor (bifurcación tomada y no tomada), produciendo esta traducción (los corchetes indican las invocaciones al traductor):

```
isPrime': mov %ecx, %edi ; IDENT
mov %esi, $2
cmp %esi, %ecx
jge [dir-salto-si] ; JCC
jmp [dir-salto-no]
```

Cada invocación al traductor consume una TU y produce un Fragmento de Código Compilado (CCF, por las siglas en inglés). Aunque se muestran los CCFs en forma textual con etiquetas como `isPrime'` para recordarnos que la dirección contiene la traducción de `isPrime`, en realidad el traductor produce código binario directamente y mantiene la correspondencia entrada-salida con una tabla hash. Continuando con el ejemplo, ahora ejecutamos el código traducido

Como estamos calculando `isPrime(49)`, el salto jge no es tomado (%ecx es 49), entonces procedemos con la bifurcación `dir-salto-no` e invocamos al traductor en la dirección del guest nexti. Esta segunda TU termina con jz. Su traducción es similar a la traducción de la TU anterior con todas las instrucciones IDENT excepto el jz final.

Para acelerar las transferencias inter-CCF, el traductor emplea una optimización conocida como *chaining*, permitiendo que un CCF salte directamente a otro CCF sin tener que realizar invocaciones fuera de la caché de traducción (TC). Estos saltos en cadena reemplazan a los saltos de continuación, los cuales por lo tanto sólo se ejecutan una vez. Incluso es frecuentemente posible omitir los saltos en cadena y pasar de un CCF al próximo. Esta intercalación entre traducción y ejecución continúa mientras el guest se ejecute, aunque con una proporción decreciente de traducción a medida que el TC gradualmente capture el conjunto de trabajo del guest. Para *isPrime*, después de iterar lo necesario en el bucle de la sentencia `for` como para detectar que 49 no es un primo, se termina con el siguiente código en la TC:

```
isPrime': *mov %ecx, %edi ; IDENT
mov %esi, $2
cmp %esi, %ecx
jge [dir-salto-si] ; JCC
; si no toma el salto, ejecuta el próximo CCF
nexti': *mov %eax, %ecx ; IDENT
cdq
idiv %esi
test %edx, %edx
jz notPrime' ; JCC
; si no toma el salto, ejecuta el próximo CCF
*inc %esi ; IDENT
cmp %esi, %ecx
jl nexti' ; JCC
jmp [fallthrAddr3]
notPrime': *xor %eax, %eax ; IDENT
pop %r11 ; RET
mov %gs:0xff39eb8(%rip), %rcx ; spill %rcx
movzx %ecx, %r11b
jmp %gs:0xfc7dde0(8*%rcx)
```

Arriba hay cuatro CCFs con la primer instrucción de cada uno marcada con un asterisco. Quedan dos invocaciones al traductor porque nunca fueron ejecutadas mientras que otras dos desaparecieron por completo y una fue reemplazada con un salto en cadena a nexti'.

Como 49 no es un número primo, nunca se traduce el BB que retorna 1 en isPrime. Más generalmente, el traductor captura un trazo de ejecución del código del guest asegurándose que el código TC tenga un buena localidad de i-cache si la primer y subsecuentes ejecuciones sieguen caminos similares en el código del guest.

El manejo de errores y otro código que se ejecuta raramente tiende a ser traducido más tarde que la primera ejecución (si es traducido alguna vez), causando que sea ubicado lejos del camino crítico.

La mayoría de los registros virtuales se asocian a sus contrapartes físicas durante la ejecución del código TC para facilitar la traducción IDENT. Una excepción es el registro de segmento %gs.

isPrime es atípica en el sentido de que no contiene accesos a memoria. Sin embargo, éstos suelen ser comunes, por lo que su traducción debe ejecutarse a una velocidad cercana a la nativa y a la vez prevenir los accesos no intencionados al VMM. Los requerimientos de eficiencia hacen que se prefiera el uso de protección por hardware en lugar de insertar chequeos explícitos de direcciones de memoria.

x86 ofrece dos mecanismos de protección: paginación y segmentación. Para la traducción binaria la segmentación funciona mejor. Se mapea el VMM en la parta alta del espacio de direcciones del guest y se usa la segmentación para segregar la porción del guest (baja) de la porción del host (alta) del espacio de direcciones. Luego se “truncan” los segmentos del guest para que no se superpongan con los del VMM. Si una instrucción traducida intenta acceder al VMM se producirá un fallo. Selectivamente (para comunicación), el traductor inserta un prefijo %gs para ganar acceso al espacio del VMM desde el código guest. Y, por el contrario, para las instrucciones ocasionales del guest que tienen un prefijo %gs, el traductor utiliza una traducción NO-IDENT.

Mientras la mayoría de las instrucciones pueden ser traducidas IDENT, hay varias excepciones notables:

- Direccionamiento relativo al PC: no puede ser traducido IDENT porque la salida del traductor reside en direcciones diferentes a las de la entrada. El traductor inserta código de compensación para asegurarse el correcto direccionamiento. El efecto neto es una pequeña expansión del código y un impacto en la eficiencia.

- Control de flujo directo: Como el esquema del código cambia durante la traducción, el control de flujo debe ser reconectado en el TC. Para llamadas directas, bifurcaciones y saltos, el traductor puede hacer el mapeo desde la dirección del guest a la dirección del TC. Hay un pequeño impacto en el tiempo.

- Control de flujo indirecto (jmp, call, ret): no va a un destino fijo, lo que impide el *binding* en tiempo de traducción. En su lugar el destino traducido debe ser computado dinámicamente, por ejemplo, con una búsqueda en una tabla hash. El overhead resultante puede ser importante.

- Instrucciones privilegiadas: Se usan secuencias internas al TC para operaciones simples. Estas pueden ser más rápidas que las nativas, por ejemplo la instrucción CLI (desactivar interrupciones) puede tardar más ciclos que su traducción (`vcpu.flags.IF:=0`). Las operaciones más complejas, como los cambios de contexto, invocan al VMM, causando un costo notable debido tanto a la invocación como al trabajo de emulación.

Performance de la traducción binaria

Es claro que reemplazar código x86 con un subconjunto más seguro es menos costoso en términos de eficiencia que permitir que las instrucciones privilegiadas ejecutadas por los guests resulten en traps y generen la necesidad manejar posteriormente esos traps en el VMM. Sin embargo existen ciertos puntos en los que la traducción binaria muestra su debilidad, algunos de los cuales fueron brevemente mencionados en el capítulo anterior:

- System Calls
- Virtualización de Entrada/Salida
- Manejo de memoria virtual

System Calls

Una system call es el resultado de una aplicación que requiere un servicio del kernel. La arquitectura x86 provee un mecanismo de muy baja latencia para la implementación de system calls: SYSENTER (o SYSCALL) y SYSEXIT. Con esta técnica, una system call implicará algo de trabajo extra para el VMM. Como se mencionó anteriormente, algunas técnicas de virtualización, como Traducción Binaria, colocan al guest en un anillo menos privilegiado que en una situación nativa (ring 1 en lugar de 0).

El problema surge porque una invocación a SYSENTER lleva automáticamente el control de la CPU a un nivel de privilegios 0. Entonces el VMM debe emular cada system call, traducir el código, y luego entregar el control al código traducido del kernel que correrá en ring 1.

Cuando el código binario traducido del sistema operativo guest haya terminado su tarea, usará la instrucción SYSEXIT para retornar al espacio de usuario. Sin embargo, el guest se estará ejecutando en el nivel de privilegios 1 y por lo tanto no tendrá los privilegios necesarios para ejecutar SYSEXIT, Entonces la CPU generará un fallo que llevará el control nuevamente al ring 0, permitiendo que el VMM emule el retorno. Claramente, las system calls causan un gran costo adicional: una system call en una máquina virtual puede tardar 10 veces más que en una máquina física [TRAD].

Virtualización de Entrada/Salida

La Entrada/Salida es un gran problema para cualquier forma de virtualización. Si un servidor virtualizado requiere más poder de cómputo, podemos simplemente agregar más CPUs o Cores (por ejemplo, podemos reemplazar una CPU dual-core por una quad-core). Sin embargo, el ancho de banda a memoria, el chipset y el HBA (host bus adapter) del almacenamiento son en la mayoría de los casos compartidos por todas las máquinas virtuales y mucho más complicado de actualizar.

Peor aún, a diferencia de lo que sucede con la CPU, el resto del hardware en la mayoría del software de virtualización solía ser emulado, lo que significa que cada acceso al driver de un componente de hardware virtual debía ser traducido al driver real. Esto está comenzando a ser reemplazado por técnicas más eficientes.

Por ejemplo, si inspeccionamos el hardware de una máquina virtual ESX 3.5 de VMware, un producto del tipo hipervisor para el mercado de servidores, veremos que el sistema operativo guest listará los siguientes componentes/dispositivos:

- **BIOS:** Phoenix 6.0
- **Procesador:** Igual al de la plataforma nativa
- **Motherboard:** Intel 440BX
- **Placa de Video:** Adaptador gráfico VMware, VGA estándar con 4 MB de memoria de video
- **CD-ROM:** NEC VMware IDE CDR00
- **Placa de red(NIC):** Dependiendo del sistema operativo guest, encontraremos modelos similares a Pcnet de AMD o e1000 de Intel.
- **Controlador IDE:** Intel 82371 AB/EB PCI Bus Master IDE Controller
- **Controlador SCSI:** Puede ser LSI Logic PCI-X Ultra320 o Buslogic BA80c30 PCI-SCSI MultiMaster.

Es decir, las modernas CPUs de los servidores hosts deberán interactuar con el chipset BX, que aunque es un dispositivo de calidad, ya tiene más de 10 años de existencia, al igual que con el HBA virtual que se implementa con una placa LSI o un antiguo Buslogic.

Manejo de memoria virtual

Como se explicó previamente, es claro que el sistema operativo guest ejecutándose en una máquina virtual no puede tener acceso a las tablas de páginas reales. En su lugar, el sistema operativo guest ve tablas de páginas que se ejecutan en un MMU emulado. Estas tablas dan al guest la ilusión de que puede traducir las direcciones virtuales de sistema operativo guest a las direcciones físicas reales, pero en realidad es el VMM el que maneja esta traducción de manera transparente. Las tablas de páginas reales las administra el VMM y se exponen al MMU físico. Por esto es que se les llama tablas de páginas shadow a las tablas usadas para traducir direcciones virtuales del sistema operativo guest a direcciones físicas del host.

Cada vez que el sistema operativo guest modifica su mapeo, el módulo que implementa el MMU virtual debe capturar (por traps) la modificación y ajustar las tablas de página shadow de manera apropiada. Esto claramente cuesta muchos ciclos de CPU. Dependiendo de las técnicas usadas y la frecuencia de cambios en el mapeo de memoria esta tarea lleva de 3 a 400 veces más ciclos que en la situación nativa [CERN]. El resultado es que en las aplicaciones intensivas en memoria, el manejo de memoria virtual causa la mayor parte del costo que se debe pagar por utilizar virtualización.

Traducción Binaria Adaptativa

Las CPUs modernas tienen mecanismos de traps relativamente costosos, por lo que un VMM basado en traducción binaria puede lograr buenos resultados de performance en comparación con un VMM basado en virtualización clásica, al reemplazar los traps por llamadas explícitas.

Para ilustrar esto, se menciona a continuación una simple prueba publicada en [COMP] en el cual se compara la implementación de una sencilla instrucción privilegiada (RDTSC) en un procesador Pentium 4:

3. trap and emulate: 2030 ciclos
4. llamada explícita y emulación: 1254 ciclos
5. emulación en TC: 216 ciclos

Sin embargo, aunque la traducción binaria elimina los traps de las instrucciones privilegiadas, no se resuelve una fuente de traps incluso más frecuentes: las instrucciones no privilegiadas (como los LOADS y STORES) que acceden a datos sensibles tales como tablas de páginas.

VMware propone el uso de técnicas de traducción binaria adaptativa para eliminar esta última categoría de traps. La idea básica es que una instrucción 'se considera inocente hasta que se demuestre lo contrario'. Las instrucciones del guest comienzan en un estado denominado *inocente*, asegurando el máximo uso de traducciones IDENT. Durante la ejecución del código traducido se detectan las instrucciones que generan traps frecuentemente y se adapta su traducción:

- Se re-traduce las no IDENT para evitar los traps; por ejemplo, las traducciones podrían invocar a un intérprete

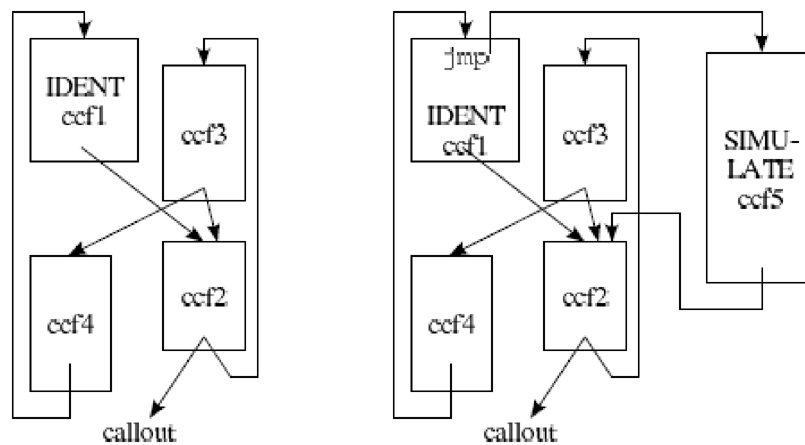
- Se reemplaza la traducción IDENT original con un salto hacia la nueva traducción

La mitad izquierda de la siguiente figura muestra un gráfico de flujo de control con una traducción IDENT en el ccf1, y otro control de flujo arbitrario en la caché del traductor representado por ccf2, ccf3 y ccf4. La mitad derecha muestra el resultado de adaptar desde IDENT en ccf1 a un tipo de traducción que invocará a código de emulación de la instrucción *culpable* en ccf5.

Luego de la adaptación, se evita generar un trap en ccf1 y en su lugar se ejecuta una llamada explícita en ccf5. El código en ccf5 continúa la monitorización del comportamiento de la instrucción conflictiva. Si el comportamiento cambia y la instrucción se vuelve inocente nuevamente, se vuelve a insertar en el flujo la traducción de tipo IDENT, removiendo el salto de ccf1.

El VMM usa la adaptación no sólo de manera bi-modal que distingue entre inocente y culpable, pero además con la habilidad de adaptar una variedad de situaciones, incluyendo accesos a tablas de páginas, accesos a dispositivos particulares y accesos al rango de direcciones de VMM.

Una instrucción guest cuya traducción ha sido adaptada sufre un costo dinámico por el salto necesario para alcanzar la traducción de reemplazo. Sin embargo, el costo estático de la adaptación, por ejemplo, la re-escritura del código y la pérdida consecuente de los contenidos de la i-cache, puede ser controlado con histéresis para asegurar una frecuencia de adaptación baja. Se adaptan agresivamente desde una situación que genera muchos traps a formas que no generen tantos, pero menos agresivamente al acercarse a una traducción más fluida.



Bibliografía

[META] http://www.ibm.com/developerworks/linux/library/l-metaprog1.html?ca=dgr-wikiaMetaprogramming

[BYTE] en.wikipedia.org/wiki/Bytecode

[INOV] A Hardware Approach for Reducing Interpretation Overhead. Wei Chen et. al. 2009 Ninth IEEE International Conference on Computer and Information Technology Xiamen, China. ISBN: 978-0-7695-3836-5.

[INLI] http://en.wikipedia.org/wiki/Inline_function

[SMAL] "Efficient Implementation of the Smalltalk-80 System"
L. Peter Deutsch and Allan M. Schiffman, POPL 1984. Publicado en Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages

[COMP] A comparison of software and hardware techniques for x86 virtualization : Keith Adams VMware Ole Agesen VMware. · Proceeding ASPLOS-XII Proceedings of the 12th international conference on Architectural support for programming languages and operating systems ACM New York, NY, USA ©2006.

[BLOC] en.wikipedia.org/wiki/Basic_block

[CERN] Introduction to virtualisation technology. Predrag Buncic
CERN. CERN School of Computing 2009

[TRAD] http://hi.baidu.com/callesp/blog/item/727bd5fd8d350c57d6887d9a.html

http://www.anandtech.com/show/2480/4

http://it20.info/2007/06/a-brief-architecture-overview-of-vmware-esx-xen-and-ms-viridian/

http://personals.ac.upc.edu/vmoya/translation.html

5. Paravirtualización

La paravirtualización, al igual que las técnicas de virtualización alternativas utilizadas en la actualidad, es una idea que data de los primeros intentos de sistemas operativos de tiempo compartido para mainframes, como se mencionó en el capítulo 2. En particular, IBM la implementó en su sistema operativo conocido como VM.

Más recientemente, Xen, un proyecto de código abierto, volvió a proponer el concepto de paravirtualización como tecnología de virtualización por software alternativa a la traducción binaria de VMware. Xen fue inicialmente un proyecto de investigación de la Universidad de Cambridge (la primer versión del software fue publicada a fines de 2003) liderado por Ian Pratt, quien en enero de 2005 fundó la empresa Xensource Inc, (en la actualidad adquirida por Citrix).

En este capítulo se revisan los conceptos que permiten construir las soluciones basadas en paravirtualización. Además se exploran las facilidades ofrecidas en el kernel Linux para la implementación de los distintos hipervisores y se estudia en profundidad a uno de ellos: Lguest, el proyecto de paravirtualización pura incluido en el kernel Linux.

Primeras Definiciones

La paravirtualización permite que múltiples sistemas operativos se ejecuten en el hardware al mismo tiempo haciendo un uso eficiente de los recursos, como procesadores y memoria, a través de la multiplexación efectiva de esos recursos entre las máquinas virtuales.

A diferencia de la virtualización completa, donde se emula todo el sistema (BIOS, discos, procesadores, NICs, etc), un VMM de paravirtualización opera con guests que han sido modificados para cooperar.

Se explicó anteriormente que el kernel de un sistema operativo monolítico se suele ejecutar en el nivel 0 (más privilegiado) mientras que las aplicaciones corren en el nivel 3 (menos privilegiado). En la paravirtualización también se utiliza la técnica de *ring deprivileging*, por la cual el sistema operativo es modificado para poder ejecutarse en el nivel 1, dejando el nivel 0 para una ejecución segura del hipervisor. Esta técnica fue descripta en detalle en el capítulo 3.

La característica distintiva radica en la nueva forma de comunicación entre el sistema operativo guest y el VMM para mejorar la performance. El código fuente del kernel guest debe modificarse para reemplazar las instrucciones no virtualizables por llamadas explícitas al VMM, denominadas *hypercalls*. Se elimina de esta forma la necesidad de utilizar traps, que suelen ser costosas en términos de performance, o de utilizar traducción binaria, que es compleja de implementar y mantener.

La interfaz de la máquina virtual difiere de la interfaz ofrecida por el hardware subyacente. Es decir las hypercalls son distintas de las instrucciones nativas, motivo por el cuál se requiere la modificación del código fuente del kernel guest. El hipervisor también puede proveer interfaces adicionales de hypercalls para otras operaciones críticas del kernel como el manejo de memoria, el manejo de interrupciones y el mantenimiento del tiempo.

Es por esta comunicación explícita que se suele decir, en sentido figurado, por supuesto, que el sistema operativo guest 'sabe' que está siendo virtualizado y que, por lo tanto, colabora con el hipervisor con mecanismos de comunicación de más alto nivel que en el caso de la traducción binaria. Esta abstracción generalmente implica una mejor performance que en los modelos de virtualización completa, donde cada componente debe ser emulado y la comunicación es de extremado bajo nivel.

Incluso aunque es muy difícil construir las sofisticadas técnicas de traducción binaria adaptativa necesarias para una eficiente virtualización completa, resulta relativamente sencillo, como veremos más adelante, modificar el código fuente del sistema operativo guest para la

implementación de la paravirtualización, reduciendo la complejidad de la capa de virtualización.

Además de la mayor performance alcanzada, la eficiencia de esta técnica puede traducirse en una mayor capacidad para escalar. Asumiendo un costo realista para cada técnica de virtualización, si una solución de virtualización completa requiere un 10% de utilización de la CPU por instancia del guest por procesador para implementar la máquina virtual, y a la vez cada guest necesita utilizar un 10% el tiempo de CPU para cálculos útiles, entonces vemos en el siguiente cuadro que sólo podremos ejecutar cinco instancias del guest, dada la plataforma de hardware.

El costo adicional de la paravirtualización suele estar entre el 0.1% y el 3% [XEN] de tiempo de CPU por guest. Asumimos aquí una media aproximada de 2%. Con lo cual, en el caso anterior, podríamos ejecutar ocho instancias del guest que requirieran un 10% de tiempo de cómputo útil. La tabla sumariza este ejemplo simplificado.

	Instancias de guest	Costo de la virtualización	Procesamiento útil	Total
Virtualización Completa	5	10% (50% total)	10% (50% total)	100%
Paravirtualización	8	2% (16% total)	10% (80% total)	96%

Sin embargo, esta eficiencia tiene un costo asociado en cuanto a flexibilidad y seguridad. La flexibilidad se reduce notablemente porque es necesario modificar un sistema operativo para que pueda ejecutarse como guest. La modificación que deben sufrir los sistemas operativos presenta una limitación ya que sólo puede llevarse a cabo en sistemas operativos de código abierto (o sólo por el fabricante del software, en el caso del software propietario). Por lo tanto, en un comienzo sólo GNU/Linux y algunas variantes de BSD fueron portados para poder ejecutarse en hipervisores.

Más recientemente, sin embargo, esta técnica fue adoptada por productos de virtualización comerciales y sistemas operativos de código cerrado para resolver ciertas cuestiones específicas, como la virtualización de dispositivos. VMware ha usado ciertos aspectos de las técnicas de paravirtualización en algunos de sus productos en los últimos años en forma de herramientas y en drivers de dispositivos virtuales optimizados [VMPAR].

En este universo, la necesidad de estandarizar las comunicaciones entre los distintos actores de la infraestructura se volvió imprescindible algunos años atrás. Los protagonistas de la historia moderna de la virtualización, rápidamente, entablaron una batalla para imponer sus propias interfaces como estándares de facto.

Siendo el kernel Linux el contexto en el cual se encuentran los ingenieros de cada gran empresa involucrada (IBM, Red Hat, VMware, etc), todas las propuestas y prototipos se presentaron como soluciones para este sistema operativo de código libre.

Paravirtualización en el kernel Linux: *paravirt_ops*

Resumiendo lo anterior, la paravirtualización es el acto de ejecutar un sistema operativo guest, bajo el control de un sistema host, donde el guest ha sido portado a una arquitectura virtual que es muy parecida, pero no igual, a la arquitectura del hardware sobre el cual está corriendo realmente. Esta técnica permite implementar virtualización de manera eficiente.

La implementación de código libre más exitosa es Xen, mientras que en el mundo de software propietario VMware ha estado activo por mucho tiempo. Ambos quisieron ver parte de sus soluciones incluidas en el kernel de Linux, ya que se ha predicho que será este sistema el común denominador en infraestructuras virtualizadas [PRED1] [PRED2]. Sin embargo, los desarrolladores del kernel no suelen estar interesados en código intrusivo relacionado con una solución en particular.

En el año 2005, VMware propuso una interfaz de paravirtualización transparente, la Interfaz de Máquina Virtual (VMI por sus siglas en inglés, Virtual Machine Interface), como un mecanismo de comunicación estandarizado entre el sistema operativo guest y el hipervisor.

Como se muestra en la figura 5.1, VMI era una capa entre el hipervisor y el guest paravirtualizado que ofrecía paravirtualización transparente al lograr que el mismo kernel guest pudiera correr tanto en hardware nativo como virtualizado en cualquier hipervisor compatible con VMI.

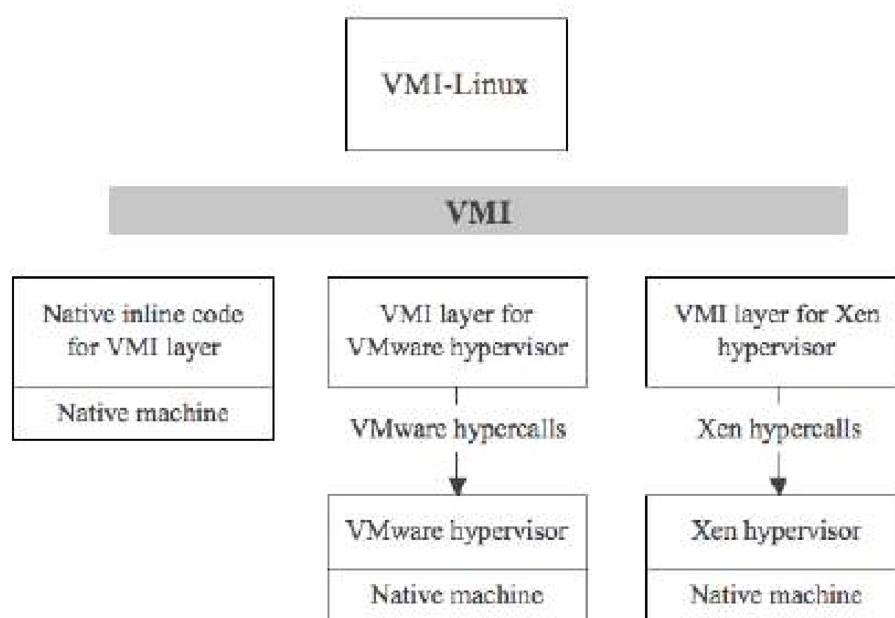


Figura 5.1 – Arquitectura de VMI [VMI06]

Funcionaba gracias a que su diseño incluía dos implementaciones para cada llamada a VMI: instrucciones nativas *inline* para el hardware y llamadas indirectas a una capa de software en máquinas virtuales.

VMI funcionaba aislando cualquier operación que pudiera requerir intervención del hipervisor en un conjunto especial de llamadas a funciones. La implementación de esas funciones no era construida (compilada) con el kernel, en su lugar el kernel, en el momento de inicialización, cargaba un ROM que proveía las funciones necesarias. La interfaz binaria entre el kernel y este segmento link-editable dinámicamente sería fijo, lo que aseguraría la comunicación correcta sin importar las modificaciones internas. Este diseño permite que la misma imagen binaria del kernel se ejecute sobre una variedad de hipervisores compatibles o, con el ROM apropiado, en modo nativo sobre el hardware real. Se lograba mantenibilidad y extensibilidad al permitir el desarrollo independiente de los hipervisores y los sistemas guest.

Sin embargo, la ABI (*Aplication Binary Interface*) fija y la posibilidad de cargar objetos binarios en el kernel no parece funcionar bien con el modelo de los desarrolladores de Linux. Parecía un nuevo intento de filtrar un mecanismo para inyectar código propietario en el kernel, lo cuál es siempre rechazado por la comunidad.

VMware continuó su colaboración con la comunidad Linux para desarrollar una interfaz de paravirtualización que soportara múltiples hipervisores. En el año 2006, VMware publicó la especificación VMI como una especificación abierta y la propuesta en el Ottawa Linux Symposium [VMI06] llevó al desarrollo de la interfaz *paravirt_ops* [PVOPS] por un equipo conjunto que incluye a la comunidad Linux y a IBM, VMware, Red Hat, y XenSource.

Buscando incrementar la compatibilidad de las soluciones de paravirtualización de CPU a través de una abstracción estándar, paravirt_ops incorpora muchos de los conceptos de VMI, incluyendo el soporte de para paravirtualización transparente. Usando esta interfaz, un sistema operativo Linux paravirtualizado es capaz de correr en cualquier hipervisor que lo soporte. La implementación se integró al kernel Linux oficial a partir de la versión 2.6.20.

Para resumir su implementación, parvirt_ops es sólo una estructura con punteros a funciones, como muchas otras estructuras de operaciones usadas en el kernel. En este caso, las operaciones son varias de las funciones específicas de la máquina que tienden a requerir interacción con el hipervisor. Se incluyen cuestiones tales como desactivar las interrupciones, cambiar los registros de control del procesador, cambiar los mapeos de memoria virtual, etc.

Por ejemplo, uno de los miembros de paravirt_ops es:

```
void (fastcall *irq_disable)(void);
```

La implementación de paravirt_ops también incluye la definición de una pequeña función que será usada por el kernel:

```
static inline void raw_local_irq_disable(void)
{
      paravirt_ops.irq_disable();
}
```

Entonces, siempre que el kernel use esta función para desactivar las interrupciones, usará inevitablemente la implementación, cualquiera sea ésta, que haya sido provista por el hipervisor y que se haya conectado con paravirt_ops.

La implementación incluye además un conjunto de operaciones nativas (para sistemas no virtualizados) que causará que el kernel se comporte como siempre lo hizo, cuando se configure para esa opción. Este kernel podría funcionar con cierto costo adicional de tiempo, sin embargo, debido a que muchas de las operaciones que eran realizadas por código assembly inline, ahora en cambio se implementan con una llamada indirecta a función. Para mitigar este problema de performance, la infraestructura de paravirt_ops incluye un mecanismo de reescritura para evitar algunas de las llamadas a funciones, en particular las relacionadas con interrupciones.

Esta interfaz tiene muchas similitudes con VMI: ambas permiten el reemplazo de operaciones importantes de bajo nivel con versiones dependientes del hipervisor. La diferencia es que paravirt_ops es una interfaz inherentemente basada en código fuente, sin garantías con respecto a su interfaz binaria. Se asume que esta interfaz puede cambiar en el tiempo, como la mayoría de las interfaces del kernel.

Lguest

Lguest [LGUEST] fue escrito por Rusty Russel (IBM, Australia) [RUSTY] y fue originalmente pensado como una herramienta para la enseñanza y la investigación y como una prueba de concepto para la implementación y depuración de algunos de los subsistemas incluidos en el framework de virtualización del kernel Linux, principalmente paravirt_ops y virtio [VIRTIO], este último explicado en detalle en el próximo capítulo.

Un módulo del kernel (lg.ko) permite ejecutar otros kernels de Linux de la misma forma en que pueden ejecutarse procesos, aunque sólo se corren guests modificados. Activar la opción de compilación *CONFIG_LGUEST_GUEST* construye el soporte necesario dentro del kernel para que al momento de inicializarse pueda detectar que es un guest y prosiga con la lógica apropiada. Esto tiene la importante implicancia de poder usar el mismo kernel que se ejecuta nativamente como host, también como una instancia con el rol de guest.

En el momento de inicialización, el módulo del kernel guest aloca una porción de memoria y la mapea dentro del espacio de direccionamiento del kernel, en un rango bien alto (por encima de la región conocida como 'región vmalloc' (reservada para operaciones de alocación con la función `vmalloc`) [VMAL], es decir, en el tope de la memoria virtual.

Un pequeño hipervisor es cargado en ese área; básicamente consiste en código assembly que implementa principalmente el cambio de contexto entre el kernel host y el guest virtualizado. Esta operación involucra el cambio de tablas de páginas y el intercambio de los contenidos de los registros de la CPU.

Por otro lado, el *launcher* es el programa en el espacio de usuario del host que prepara, ejecuta y provee servicios al guest. La memoria física del guest es igual a la memoria virtual del launcher más un desplazamiento. Aquí se muestra un diagrama completo del mapa de memoria creado mientras se realizó el análisis del código de Lguest:

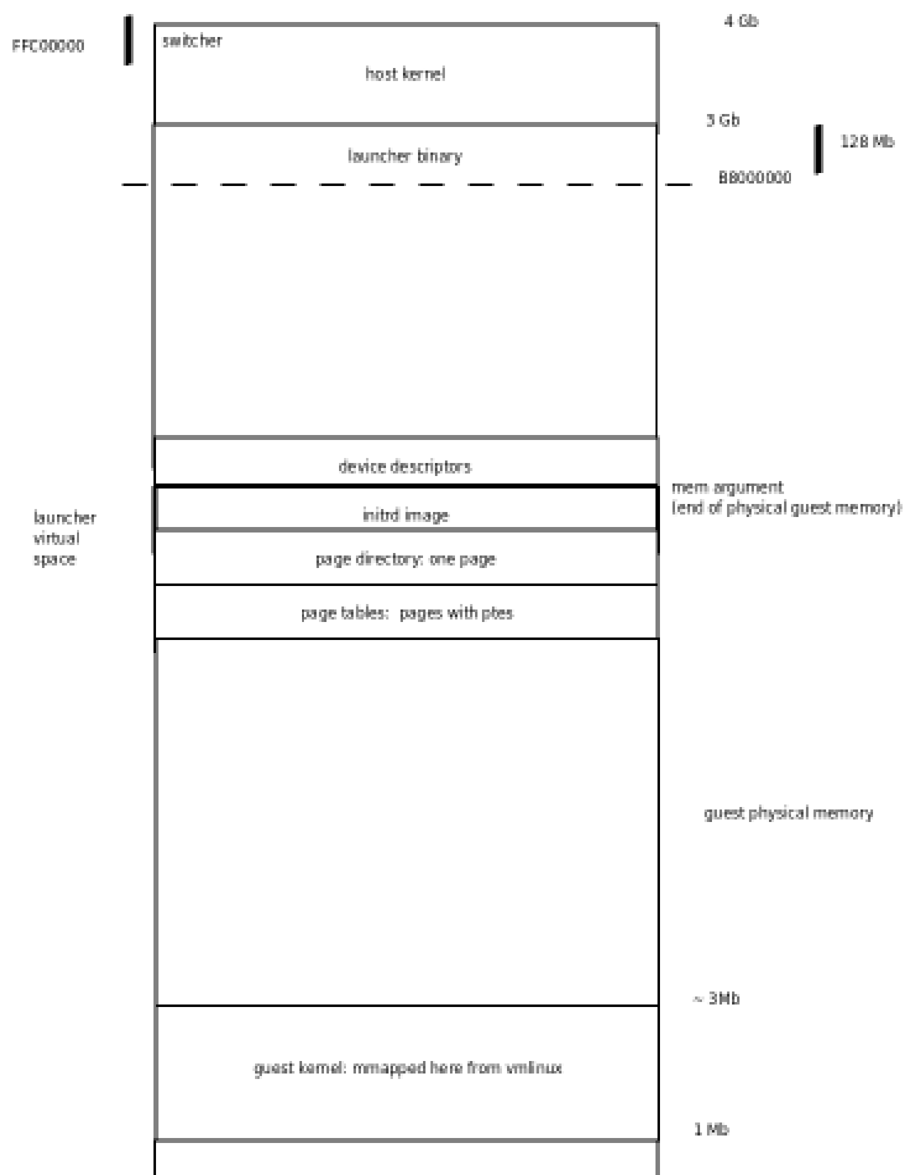


Figura 5.2 – Mapa de Memoria de Lguest

Los guests saben que no pueden realizar operaciones privilegiadas, tales como desactivar las interrupciones, y que deben pedirle explícitamente al host que realice tales tareas por ellos. Algunos de los reemplazos para tales operaciones nativas de hardware de bajo nivel invocan al host a través de las ya mencionadas hypercalls. La otra forma, más eficiente por cierto, es la comunicación a través de una estructura colocada en memoria compartida entre el host y el guest. Sin embargo, no siempre es posible usar este espacio compartido y así evitar la transición al host. Cuando el guest requiere alguna operación que debe ejecutarse en el ring 0, una hypercall invocará a la funcionalidad correspondiente en el host.

El mecanismo de hypercalls de Lguest utiliza el número no reservado más alto del rango de interrupciones de x86 (las interrupciones con número 32 y por encima son usadas por interrupciones de hardware reales, como se explica en el Apéndice A). El mecanismo es muy similar a la implementación tradicional de system calls (previas a las facilidades sysenter/sysexit de los procesadores más modernos): se colocan los argumentos en los registros de la CPU y se genera una interrupción por software.

```
static inline unsigned long hcall(unsigned long call,  unsigned long arg1,
                                        unsigned long arg2, unsigned long arg3)
{
      asm volatile("int $" __stringify(LGUEST_TRAP_ENTRY)
                  /* The call in %eax (aka "a") might be overwritten */
                  : "=a"(call)
                    /* The arguments are in %eax, %edx, %ebx & %ecx */
                  : "a"(call), "d"(arg1), "b"(arg2), "c"(arg3)
                    /* "memory" means this might write somewhere in memory.
                     * This isn't true for all calls, but it's safe to tell
                     * gcc that it might happen so it doesn't get clever. */
                  : "memory");
      return call;
}
```

El kernel guest es modificado, a través de la infraestructura paravirt_ops, para invocar a las nuevas implementaciones que reemplazan las operaciones nativas. Por ejemplo, si un guest quisiera modificar una entrada en una de sus tablas de páginas, el host debería tener la posibilidad de reflejar este cambio en las estructuras shadow. Entonces, el guest es modificado para invocar una hypercall específica para esta tarea. El reemplazo de la función nativa `set_pte_at()` es `lguest_set_pte_at()`, la cual se implementa de la siguiente forma:

```
static void lguest_set_pte_at(struct mm_struct *mm,unsigned long addr,
                              pte_t *ptep, pte_t pteval)
{
      *ptep = pteval;
      lazy_hcall(LHCALL_SET_PTE, __pa(mm->pgd), addr, pteval.pte_low);
}
```

Del lado del host, al detectarse una hypercall se invoca a la rutina `do_hcall()` la cual, como en cualquier función manejadora de system calls, básicamente implementa una gran sentencia *switch* para saltar a la lógica invocada por el guest, según los argumentos enviados.

```
case LHCALL_FLUSH_TLB:
/* FLUSH_TLB comes in two flavors, depending on the argument: */
if (args->arg1)
      guest_pagetable_clear_all(cpu);
else
      guest_pagetable_flush_user(cpu);
break;
```

```
case LHCALL_NEW_PGTABLE:
      guest_new_pagetable(cpu, args->arg1);
break;

case LHCALL_SET_STACK:
      guest_set_stack(cpu, args->arg1, args->arg2, args->arg3);
break;

case LHCALL_SET_PTE:
      guest_set_pte(cpu, args->arg1, args->arg2, __pte(args->arg3));
break;
```

Cuando el kernel guest comienza su ejecución lo hace con el código assembly en el archivo arch/x86/kernel/head_32.S. En general este código sólo prepara el entorno para la posterior ejecución de código C. El kernel en esta instancia espera encontrar un *boot header*, que generalmente es creado por el *bootloader* (usualmente *grub*, en entornos GNU/Linux). En nuestro caso, es el proceso launcher el que simula ser un bootloader y coloca el boot header en el lugar de la memoria donde el kernel guest lo buscará. Cuando el guest encuentra el header, verifica uno los datos incluidos en él, el campo *'hardware_subarch'*. Éste fue introducido en la versión 2.6.24 del kernel Linux, principalmente para las soluciones de paravirtualización Xen y Lguest: si está asignado con el número 1 (el identificador de Lguest), entonces el guest invoca a la función `lguest_entry`.

Se preparan las tablas de páginas para crear un mapeo identidad (*identity mapping)* y un mapeo lineal (*linear mapping)*. El mapeo identidad asocia direcciones virtuales a las mismas direcciones físicas. Por ejemplo, siendo x una dirección virtual y P() la función que devuelve la dirección física para una dirección virtual de entrada,

$P(x) = x$

El mapeo lineal es análogo al mapeo identidad pero desplazado una cierta cantidad de direcciones. Un ejemplo podría ser mapear el 4to gigabyte del espacio de direcciones virtuales en un sistema 32 bits a direcciones físicas desde el primer gigabyte. Por ej:

$P(x) = x - 3Gb$

cuando los sistemas operativos se están inicializando, configurando y activando la paginación, suelen necesitar ambos mapeos en las tablas de páginas. Después se puede descartar el mapeo identidad para recuperar ese espacio de direcciones virtuales.

Finalmente se prepara la pila inicial para poder así ejecutar el primer código C, que en el caso estudiado será la función lguest_init. Una vez que la función lguest_init() está ejecutándose, ese kernel *sabe* que lo que está construyendo es un sistema operativo guest que correrá sobre el hipervisor Lguest. Debe realizar las modificaciones necesarias para comportarse correctamente en un entorno paravirtualizado, sobre-escribiendo las operaciones nativas. El framework paravirt_ops provee los reemplazos principales para casi todas las rutinas nativas que deben ser modificadas por las versiones del guest. Luego de realizar el reemplazo, se continúa la inicialización como en un entorno nativo. El guest puede funcionar normalmente.

A continuación se muestra el código de Lguest que prepara su propia *subclase* de la estructura paravirt_ops, para ser usada por un eventual guest en el momento de inicialización para reemplazar las operaciones nativas:

```
lguest_init(void)
      /* interrupt-related operations */
      pv_irq_ops.init_IRQ = lguest_init_IRQ;
      pv_irq_ops.save_fl = save_fl;
      pv_irq_ops.restore_fl = restore_fl;
```

```
        pv_irq_ops.irq_disable = irq_disable;
        pv_irq_ops.irq_enable = irq_enable;
        pv_irq_ops.safe_halt = lguest_safe_halt;

        /* init-time operations */
        pv_init_ops.memory_setup = lguest_memory_setup;
        pv_init_ops.patch = lguest_patch;

        /* Intercepts of various cpu instructions */
        pv_cpu_ops.load_gdt = lguest_load_gdt;
        pv_cpu_ops.cpuid = lguest_cpuid;
        pv_cpu_ops.load_idt = lguest_load_idt;
        pv_cpu_ops.iret = lguest_iret;
        pv_cpu_ops.load_sp0 = lguest_load_sp0;
        pv_cpu_ops.load_tr_desc = lguest_load_tr_desc;
        pv_cpu_ops.set_ldt = lguest_set_ldt;

        /* pagetable management */
        pv_mmu_ops.write_cr3 = lguest_write_cr3;
        pv_mmu_ops.flush_tlb_user = lguest_flush_tlb_user;
        pv_mmu_ops.flush_tlb_single = lguest_flush_tlb_single;
        pv_mmu_ops.flush_tlb_kernel = lguest_flush_tlb_kernel;
        pv_mmu_ops.set_pte = lguest_set_pte;
```

El segundo método de comunicación con el host es a través de la estructura *lguest_data*. Una vez que el guest se inicializa, le indica al host, mediante una hypercall de inicialización, dónde encontrar esta estructura. Luego, el guest y el host pueden publicar información en la misma sin ningún costo significativo, ya que la estructura se aloca en una porción de memoria compartida entre ambos.

Por ejemplo, dentro de esta estructura existe un campo, denominado *irq_enabled*, que puede ser usado por el guest para actualizar su estado con respecto a las interrupciones con una única y simple instrucción. Entonces, el host sabe que debe verificar el estado de esta variable compartida antes de intentar inyectar una interrupción.

El trabajo realizado con Lguest durante el período de investigación

Gran parte de los mecanismos fundamentales de la paravirtualización están implementados en Lguest: la paginación shadow, el cambio de contexto de guest a host y viceversa, la virtualización de dispositivos, el manejo de tiempo virtual, etc. Es por esto que este proyecto fue elegido para el estudio detallado de una implementación de virtualización real y para la prototipación de conceptos e ideas que fueron desarrollados para esta tesis.

A continuación se explican las modificaciones realizadas al código del Lguest para implementar nuevas funcionalidades, bajo la supervisión técnica de su autor, Rusty Russell. El código correspondiente puede ser examinado accediendo a los repositorios oficiales del kernel Linux, y se presenta en forma de parches enviados y aceptados por la comunidad. Otra parte del trabajo de los últimos años se encuentra en un estado prototípico, generalmente funcional aunque aún requiere ser mejorado para cumplir con las exigencias de eficiencia o de formato del código para poder ser presentado a la comunidad.

Para acceder a los parches en el repositorio del kernel Linux, sólo es necesario hacer una búsqueda por el apellido del autor de esta tesis, aunque se indica aquí como referencia rápida [PATCH].

Los parches en GIT se identifican con un *hash* único, por lo que es posible descargar el árbol del código fuente del kernel Linux y revisar la historia de cada cambio realizado. Además, los parches se asocian a un título representativo de la funcionalidad implementada. A continuación se listan los cambios introducidos, mostrando el encabezado de cada uno de los parches y una

breve explicación. La discusión completa puede seguirse en la lista de correos de Lguest: https://ozlabs.org/mailman/listinfo/lguest.

a) lguest: Pagetables to use normal kernel types
This is my first step in the migration of page_tables.c to the kernel types and functions/macros (2.6.23-rc3). Seems to be working OK.

Signed-off-by: Matias Zabaljauregui
Signed-off-by: Rusty Russell

El primer paso para lograr portabilidad en el manejo de memoria, reducir el tamaño del código e incrementar la legibilidad del mismo, fue la de modificar Lguest para que utilizara los mismos tipos de datos que utiliza el código del kernel para representar entradas en las tablas de páginas. Esto facilita el trabajo posterior de implementar nuevas funcionalidades y permite usar las funciones y macros ya implementadas.

b) lguest: move the initial guest page table creation code to the host
This patch moves the initial guest page table creation code to the host, so the launcher keeps working with PAE enabled configs.

Signed-off-by: Matias Zabaljauregui
Signed-off-by: Rusty Russell

Cuando un guest se inicializa, requiere una tabla de páginas temporal para acceder a la memoria virtual hasta que el sistema operativo guest construye sus tablas propias. Esta tarea se realizaba originalmente desde el espacio de usuario, en el proceso llamado *Launcher.* Cuando comenzamos a implementar la posibilidad de ejecutar Lguest en procesadores con distintas capacidades, migramos esta funcionalidad al kernel *Host* de manera que el proceso Launcher no tuviera que ser modificado para cada variante de CPU en la que Lguest fuera a ejecutarse. De esta forma, por ejemplo, si se compila el kernel *Host* con soporte para *Physical Address Extension (PAE)*, las tablas de página iniciales serán creadas teniendo en cuenta este formato.

c) lguest: Physical Address Extension support
This patch adds Physical Address Extension support to lguest.
Signed-off-by: Matias Zabaljauregui

Physical Address Extension (PAE) es una extensión introducida en la arquitectura x86 que permite a los sistemas de 32 bits acceder hasta 64 gigabytes de memoria física, suponiendo que el sistema operativo lo soporte, superando de esta forma la conocida barrera de los 4 gigabytes que permiten los 32 bits de espacio de direccionamiento de memoria. PAE está disponible en todos los procesadores desde el Intel Pentium Pro y compatibles como AMD. Lguest no soportaba PAE y ésta era una de sus limitaciones más importantes para poder ser usado en **todas** las distribuciones GNU/Linux, ya que normalmente los kernels empaquetados en las distribuciones son compilados con soporte para PAE.

d) lguest: Page Size Extension support
This patch adds Page Size Extension support to lguest.
Signed-off-by: Matias Zabaljauregui

Page Size Extension (PSE) es una característica de los procesadores x86, desde el procesador Pentium, que permite la utilización de páginas más grandes que las tradicionales de 4 kilobytes. Utilizado por defecto en todos los kernels Linux y en los otros sistemas operativos, esta extensión ayuda a disminuir el uso de memoria principal al reducir la cantidad de estructuras intermedias y acelera el acceso al quitar un nivel de indirección en la traducción de direcciones virtuales a direcciones físicas.

e) lguest: Change over to using KVM hypercalls mechanism
This patch allows us to use KVM hypercalls.
Signed-off-by: Matias Zabaljauregui

Otro objetivo estratégico importante que se planteó a partir del surgimiento de KVM como la solución de virtualización por defecto elegida por las distribuciones Linux y empresas más importantes es el de mantener a Lguest tan compatible como sea posible con KVM.

Ambos proyectos tienen metas completamente distintas. Mientras KVM es un proyecto de enormes dimensiones, que está siendo portado a muchas arquitecturas y en cuyo entorno no se suele prestar lugar para la experimentación (por ser un proyecto con orientación a infraestructuras en producción), Lguest implementa un conjunto mínimo de funcionalidades, ofrece muy buena documentación e invita a la propuesta de ideas nuevas.

Sin embargo, pueden verse como proyectos complementarios en el sentido de que KVM requiere el uso de CPUs de nueva generación que implementen las extensiones de virtualización asistida por hardware, mientras que Lguest sigue el modelo de paravirtualización pura por software.

Además, generalmente los desarrolladores de KVM y Lguest intercambian ideas constantemente, con lo cual parece natural poder intentar compartir porciones de código específico o capas de abstracción completas (virtio o paravirt_ops son ejemplos concreto de esto). La mejor documentación (en inglés) de este parche se encuentra *inline,* por lo que se cita a continuación una parte del código:

*"... After Anthony's "Refactor hypercall infrastructure" kvm patch, we decided to change over to using kvm hypercalls. KVM_HYPERCALL is actually a "vmcall" instruction, which generates an invalid opcode fault (fault 6) on non-VT cpus, so the easiest solution seemed to be an *emulation approach*: if the fault was really produced by an hypercall (is_hypercall() does exactly this check), we can just call the corresponding hypercall host implementation function.*

*But these invalid opcode faults are notably slower than software interrupts. So we implemented the *patching (or rewriting) approach*: every time we hit the KVM_HYPERCALL opcode in Guest code, we patch it to the old "int 0x1f" opcode, so next time the Guest calls this hypercall it will use the faster trap mechanism."*

También se realizaron **pruebas de performance** que surgieron de las discusiones con respecto al mecanismo más eficiente para la implementación de las hypercalls, con una muy buena aceptación de los resultados por parte de la comunidad. A continuación se cita parte del texto

"We made a performance test comparing direct software interrupt vs invalid opcode fault vs patching. Following Rusty's and Anthony's suggestions, the benchmark is based on code you can find in kvm-userspace/user/test/x86/, and basically it does a lot of hypercalls in a loop (I used LCALL_FLUSH_ASYNC, wich does nothing), measuring the total count of cycles with rdtsc and then prints the hypercall average cycle scost. I made 5 runs for every scenario and these are the results:

** 1) direct software interrupt: 2915, 2789, 2764, 2721, 2898*
** 2) emulation technique: 3410, 3681, 3466, 3392, 3780*
** 3) patching (rewrite) technique: 2977, 2975, 2891, 2637, 2884*
One two-line function is worth a 20% hypercall speed boost! "

Bibliografía

[XEN] http://staging.xen.org/about/paravirtualization.html

[VMPAR] "Performance of VMware VMI" (PDF). VMware, Inc.. 2008-02-13. http://www.vmware.com/pdf/VMware_VMI_performance.pdf. Retrieved 2009-01-22. http://www.vmware.com/pdf/VMware_VMI_performance.pdf

[PRED1] http://vmblog.com/archive/2010/12/17/gluster-prediction-2011-the-future-of-virtualization.aspx

[PRED2] http://www.thewoodlandswebdesign.com/development/news/latest/vmware_linux_is_ideal_os_for_virtualization.html

[VMI06] VMI: An Interface for Paravirtualization, OLS2006 Proceedings. Zach Amsden, Daniel Arai, Daniel Hecht, Anne Holler, Pratap Subrahmanyam. VMware, Inc.

[PVOPS] Connecting Linux to hypervisors. http://lwn.net/Articles/194543/

[LGUEST] Lguest: A simple virtualization platform for Linux. http://www.linux.com/archive/feature/126293

[VIRTIO] virtio: towards a de-facto standard for virtual I/O devices. Rusty Russell IBM OzLabs, Canberra, Australia. Newsletter ACM SIGOPS Operating Systems Review - Research and developments in the Linux kernel archive Volume 42 Issue 5, July 2008

[RUSTY] http://en.wikipedia.org/wiki/Rusty_Russell

[VMAL] http://www.makelinux.net/ldd3/chp-8-sect-4.shtml

[PATCH]http://git.kernel.org/?p=linux%2Fkernel%2Fgit%2Ftorvalds%2Flinux-2.6.git&a=search&h=HEAD&st=commit&s=zabaljauregui

6. Virtualización de Dispositivos

La performance de cualquier solución de virtualización se ve íntimamente ligada a las decisiones relacionadas con la forma de implementar la virtualización de la Entrada/Salida. Aunque suele tenerse muy en cuenta la aproximación utilizada para modelar la CPU, es común que no se considere con tanta relevancia la manera en que las máquinas virtuales interactúan con el entorno. Y es justamente aquí donde pueden generarse los principales focos que atentan contra la eficiencia de la implementación de virtualización.

En los últimos años ha habido importantes avances con respecto a la virtualización de dispositivos. Desde la alternativa inicial de emulación total, que ofrece gran flexibilidad a costa de eficiencia, hasta la asignación directa de dispositivos a máquinas virtuales, lo que implica velocidad cercana a la nativa, se abordan en este capítulo las técnicas más importantes.

Son de particular interés para esta tesis los esfuerzos recientes de la comunidad del kernel Linux por estandarizar los mecanismos de transporte y notificación entre hipervisores y dispositivos virtuales, cuando se utilizan técnicas de paravirtualización de dispositivos. Es por ésto que la segunda mitad del capítulo se dedica al análisis de virtio, un componente esencial en el framework de virtualización del kernel Linux y una de las abstracciones sobre la cuál se construyó el prototipo presentado en esta tesis.

Virtualización de dispositivos

Cuando se virtualiza un dispositivo de E/S, es necesario que la capa de virtualización responda a diversos tipos de operaciones sobre ese dispositivo virtual. Las interacciones entre el software y el dispositivo físico incluyen:

4. **Descubrimiento del dispositivo:** un mecanismo para que el software descubra, consulte y configure dispositivos en la plataforma
5. **Control del dispositivo:** un mecanismo para que el software se comunique con el dispositivo e inicie operaciones de E/S
6. **Transferencia de datos:** un mecanismo para que el dispositivo transfiera datos hacia y desde la memoria del sistema. La mayoría de los dispositivos soportan DMA para transferir datos.
7. **Interrupciones de E/S:** un mecanismo para que el hardware sea capaz de notificar al software sobre eventos y cambios de estado.

En esta sección se discute cada una de estas interacciones, cubriendo implementaciones, desafíos, ventajas y desventajas de cada técnica de virtualización. El VMM podría ser una única pila de software monolítica o podría ser una combinación de hipervisor y guests especializados (aproximación de Xen).

Emulación

Los mecanismos de E/S en plataformas nativas (no virtualizadas) normalmente se realizan sobre algún tipo de dispositivo físico de hardware. La pila de software, comúnmente un driver del kernel de un sistema operativo, interactuará con el hardware a través de algún tipo de mecanismo mapeado en memoria (MMIO, Memory Mapped I/O), en el cual el procesador emite instrucciones para leer y escribir rangos específicos de direcciones de memoria, o puertos. Los valores leídos y escritos se corresponden con funciones del hardware.

La emulación se refiere a la implementación de una copia del hardware real, pero completamente en software. Su mayor ventaja es que no requiere ningún cambio en el software guest. El software se ejecuta de la misma forma en que lo haría en el caso nativo, interactuando con el emulador del VMM igual que como lo haría con el hardware real. El

software no sabe que está dialogando, en realidad, con un dispositivo virtualizado. Para que la emulación funcione se requieren varios mecanismos complementarios.

El VMM debe exponer un dispositivo de manera tal que pueda ser descubierto por el software del guest. Un ejemplo sería presentar un dispositivo en un espacio de configuración PCI para que el guest pueda ver el dispositivo y descubrir las direcciones de memoria que puede usar para interactuar con el dispositivo.

El VMM también debe tener algún método para capturar las lecturas y escrituras emitidas a las direcciones dentro del rango de direcciones del dispositivo, así como capturar los accesos al espacio de descubrimiento del dispositivo. Ésto permite que el VMM emule el hardware real con el que el guest cree que está dialogando.

El dispositivo virtualizado, usualmente llamado modelo de dispositivo, se implementa completamente por software en el VMM, como se muestra en la figura 6.1. El VMM podría acceder a un dispositivo de hardware real en la plataforma para responder a las peticiones de E/S, pero ese componente de hardware real es totalmente independiente del modelo de dispositivo en lo que respecta al guest. Por ejemplo, un guest podría ver un modelo de disco rígido tipo IDE expuesto por el VMM, mientras que la plataforma real contiene un disco Serial ATA (SATA).

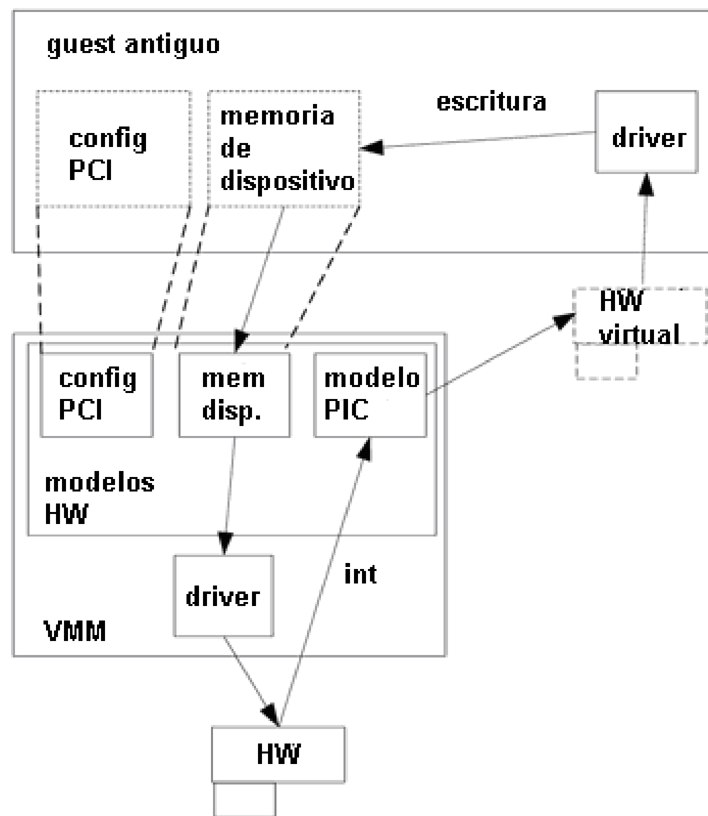


Figura 6.1 - Modelo de emulación de un dispositivo

El VMM también debe tener un mecanismo para inyectar interrupciones en el guest, en los momentos apropiados y en nombre del dispositivo emulado. Ésto usualmente se logra emulando un Controlador Programable de Interrupciones (PIC). Una vez más, cuando el guest intenta acceder al PIC, estos accesos deben ser interceptados y el VMM debe responder apropiadamente en función del modelo de PIC emulado. Aunque el PIC puede interpretarse simplemente como otro dispositivo de E/S, debe ser visible para que otros dispositivos basados en interrupciones puedan ser emulados correctamente.

La emulación facilita la migración de máquinas virtuales de una plataforma a otra. Como los dispositivos están emulados completamente y no tienen relación con los dispositivos físicos en la plataforma, es fácil mover una MV a otra plataforma donde el VMM puede soportar exactamente los mismos dispositivos emulados. Si el guest tuviera acceso a algún dispositivo físico de la plataforma, esos mismos dispositivos físicos deberían estar presentes en la plataforma de destino, complicando la migración.

La emulación también facilita la multiplexación de los dispositivos físicos de la plataforma, al exponer instancias de un modelo emulado a potencialmente un gran número de guests. El VMM puede usar algún mecanismo de planificación para permitir que los modelos emulados de cada guest puedan acceder ordenadamente al dispositivo físico. Por ejemplo, el tráfico desde muchos guests con interfaces de red emuladas podría combinarse y enviarse por una única placa de red física.

Ya que la emulación presenta una interfaz exacta de un dispositivo físico de hardware al software de un guest, permite soportar distintos guests de manera independiente con respecto al tipo de sistema operativo. Por ejemplo, si un dispositivo de almacenamiento particular se emula completamente, entonces podrá funcionar con cualquier implementación de software escrita para ese dispositivo, independientemente del sistema operativo guest (por ejemplo, Windows, Linux u otro). Ya que la mayoría de los sistemas operativos modernos incluyen drivers para muchos dispositivos conocidos, es posible seleccionar uno de estos dispositivos para ser emulados y aquellos sistemas operativos lo soportarán.

Sin embargo, mientras la gran ventaja de la emulación es que no se requieren modificaciones a los drivers de dispositivos del guest, su peor desventaja es la baja performance que ofrecen. Cada interacción del driver del sistema operativo guest con el dispositivo emulado requiere una transición al VMM, donde el modelo de dispositivo realiza la emulación necesaria, y luego una transición para volver al guest con los resultados apropiados. Dependiendo del tipo de dispositivo de E/S que se está emulando, puede ser necesario un gran número de estas transacciones para obtener realmente los datos completos desde el dispositivo. Estas actividades agregan un costo considerable comparado con las interacciones normales software/hardware en un sistema no virtualizado. La mayor parte de este costo se traduce en ciclos de CPU. También se torna importante el problema de latencia generado por los tiempos involucrados en cada interacción guest-VMM.

Otra desventaja de la emulación es que el modelo de dispositivo necesita emular el hardware con una gran precisión, y debe cubrir cualquier caso posible, incluso llegando a la necesidad de emular los defectos del hardware y cada una de las nuevas revisiones de los fabricantes.

Paravirtualización de Entrada/Salida

Otra técnica para virtualizar la E/S consiste en modificar el software en el guest, una aproximación que, como ya hemos visto, se denomina paravirtualización. La ventaja de la paravirtualización de E/S es una gran mejora en performance. La desventaja, como se mencionó previamente, es que requiere la modificación de los drivers de dispositivos del guest, lo cual limita su aplicabilidad en los casos de sistemas operativos antiguos y en los drivers de dispositivos que sólo se presentan en formato binario.

Con la paravirtualización el guest modificado interactúa directamente con el VMM, usualmente a un nivel de abstracción mayor que con la interfaz hardware/software normal. El VMM expone una API específica para E/S, por ejemplo, para enviar y recibir paquetes de red, en el caso de un adaptador de red. El software modificado en el guest entonces usa esta API del VMM en lugar de interactuar directamente con la interfaz virtual de un dispositivo de hardware.

La paravirtualización reduce el número de interacciones entre el guest y el VMM, resultando en una mejor performance (mayor rendimiento, menor latencia, menor uso de CPU), comparado con la emulación de dispositivos.

En lugar de usar un mecanismo de interrupciones emulado, la paravirtualización usa un mecanismo basado en eventos o *callback [CBAC]*. Nuevamente, ésto tiene el potencial de ofrecer mayor performance, porque se eliminan las interacciones con la interfaz de un PIC emulado, y porque la mayoría de los sistemas operativos manejan las interrupciones del hardware en etapas, agregando costo y latencia. Primero, las interrupciones son atendidas por una pequeña rutina de servicio de interrupciones (ISR). El ISR usualmente confirma la interrupción y planifica una tarea *worker* correspondiente. Esta tarea luego es ejecutada en un contexto diferente para manejar el grueso del trabajo asociado con la interrupción. Esta delegación de trabajo genera un costo adicional importante como se estudió en [ZABAL]. En cambio, con un evento o un *callback* iniciado directamente en el guest, el trabajo puede ser manejado en el mismo contexto.

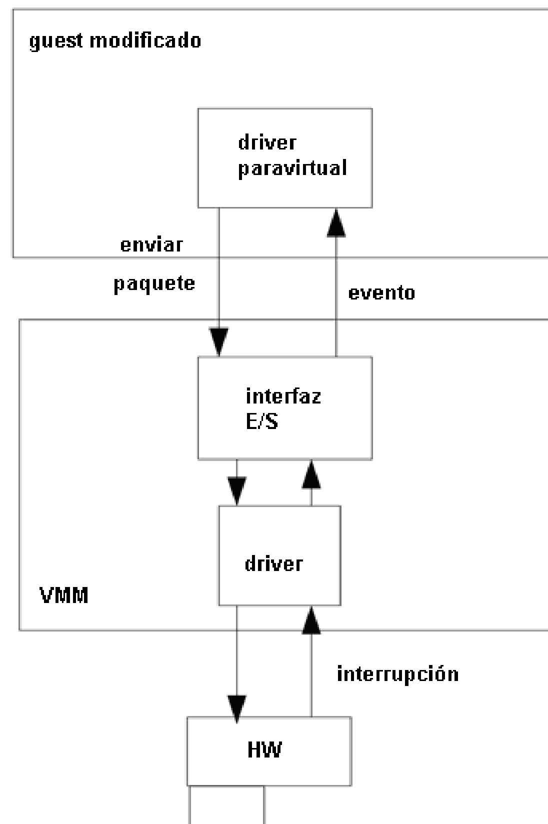


Figura 6.2 - Paravirtualización de dispositivos

Debido a que la paravirtualización involucra modificaciones al guest, usualmente los componentes modificados son específicos del entorno del guest. Por ejemplo, un driver de almacenamiento paravirtualizado para windows xp no trabajará, naturalmente, en un entorno Linux. Por lo tanto, debe desarrollarse un componente paravirtualizado separado y soportado para cada entorno de guest.

Al igual que con la emulación, la paravirtualización soporta migración de VM, suponiendo que la MV es migrada a una plataforma que soporta la misma API de VMM requerida por el guest.

La multiplexación de dispositivos físicos de la plataforma también es soportada de la misma forma que con la emulación. Por ejemplo, dos guests que usan dispositivos de almacenamiento paravirtualizados para leer y escribir datos podrían, en realidad, estar almacenando datos en el mismo dispositivo de almacenamiento físico.

El uso de esta técnica está creciendo rápidamente para satisfacer los requerimientos de performance de las aplicaciones intensivas en E/S. La paravirtualización de los tipos de dispositivos de E/S que son sensibles a la performance, como el hardware de red, de almacenamiento y de gráficos avanzados, parece ser el método elegido en las arquitecturas de VMM modernos.

Además, la paravirtualización ofrece un mayor nivel de abstracción para las interfaces de E/S dentro de los guests. La alocación de buffers de E/S y las políticas de administración, que son "*conscientes*" de que están siendo virtualizadas, pueden lograr mayor eficiencia en el uso de las facilidades de protección y traducción de VT-d (explicado en el próximo capítulo) de lo que podría ser posible con un driver no modificado que confía en una emulación completa del dispositivo.

Asignación Directa

Hay casos en los que es deseable que un dispositivo de E/S físico de la plataforma sea directamente accedido por un guest particular. Como en la emulación, la asignación directa, permite que el guest usuario del dispositivo pueda interactuar directamente con la interfaz estándar del dispositivo de hardware. Por lo tanto, la asignación directa de dispositivos provee una experiencia nativa para el guest, porque éste puede reusar los drivers existentes para hablar directamente con el hardware.

La asignación directa mejora la performance sobre la emulación porque permite que el driver del guest hable directamente con el dispositivo utilizando los comandos nativos del hardware, eliminando el costo de traducir desde el formato de comandos del dispositivo virtual emulado al formato real. Más importante aún, la asignación directa incrementa la confiabilidad de VMM y disminuye su complejidad ya que los drivers de dispositivos pueden ser removidos del VMM, asumiendo que el guest posee sus propios drivers.

Sin embargo, la asignación directa no es apropiada para todos los usos. Primero, un VMM sólo puede alocar tantos dispositivos como haya físicamente presente en la plataforma. Segundo, la asignación directa complica la migración de la máquina virtual de varias maneras. Para migrar una máquina virtual entre plataformas, un dispositivo de tipo, marca y modelo similares debe estar presente y disponible en cada plataforma. El VMM también debe desarrollar métodos para extraer cualquier estado del dispositivo físico de la plataforma de origen y restablecer ese estado en la plataforma de destino.

Además, si las plataformas físicas no ofrecen el soporte por hardware para asignación directa, ésta puede fallar en alcanzar su máximo potencial para incrementar la performance y mejorar la confiabilidad, por varios motivos.

Primero, las interrupciones de la plataforma todavía deben ser manejadas por el VMM ya que éste administra el resto de la plataforma física. Estas interrupciones deben ser ruteadas al guest apropiado, en este caso, el que posee el control del dispositivo físico. Por lo tanto, todavía hay cierto costo adicional asociado al manejo de interrupciones.

Segundo, las plataformas existentes hasta hace poco tiempo no proveían mecanismos para que un dispositivo realice transferencias de datos directamente desde y hacia la memoria del sistema, reservada para la máquina virtual, de manera eficiente y segura. Un guest, típicamente, opera en un subconjunto del espacio de direcciones físico. Lo que el guest cree que es su memoria física en realidad es un subconjunto de la memoria del sistema, virtualizada por el VMM para el guest. Esta diferencia de direccionamiento causa un problema para dispositivos con capacidades de DMA. Tales dispositivos ubican los datos directamente en la memoria del sistema sin involucrar a la CPU. Cuando el driver de dispositivo del guest instruye al dispositivo para realizar la transferencia, está usando las direcciones físicas del guest, mientras que el hardware accede a la memoria del sistema usando direcciones físicas del host.

Para resolver el problema del direccionamiento, los VMMs que soportan asignación directa deben emplear un driver conocido como *pass-through* [PASS] que intercepta todas las comunicaciones entre el driver de dispositivo del guest y el dispositivo de hardware. El driver pass-through realiza la traducción entre las direcciones físicas del guest y las direcciones físicas del host.

Los drivers pass-through son específicos de cada dispositivo ya que deben decodificar el formato de comando de un dispositivo en particular para realizar la traducción necesaria. Tales drivers realizan tareas más sencillas que los drivers de dispositivos tradicionales, por lo tanto, la performance mejora con respecto a la emulación. Sin embargo, la complejidad del VMM sigue siendo alta, impactando, de esta forma, en la confiabilidad. Aun así, los beneficios de performance han probado ser suficientes como para emplear este método en VMMs orientados a servidores, donde es aceptable soportar asignación directa para sólo un número relativamente pequeño de dispositivos comunes.

Implicaciones de la arquitectura del VMM

Los diferentes métodos para virtualización de E/S no son igualmente aplicables a todas las opciones de arquitectura de VMM.

La emulación es el método de virtualización de E/S más general, capaz de exponer dispositivos de E/S estándar a sistemas operativos guest no modificados. Por lo tanto es ampliamente utilizado en opciones de VMM tipo 1, tipo 2 e implementaciones híbridas (ver capítulo 2).

Como ya se mencionó, la paravirtualización está siendo aceptada rápidamente para mejorar la performance de los guests. Se puede aplicar fácilmente a un VMM tipo 1 y también puede ser usada en la interacción entre el sistema operativo guest y el monitor en espacio de usuario en un VMM tipo 2.

La asignación directa se usa en los casos donde el sistema operativo guest no puede ser modificado, ya sea porque es complicado hacerlo o porque los drivers de dispositivos paravirtualizados no son apropiados para una aplicación específica. Sin embargo, es difícil introducir asignación directa en un VMM tipo 2 ya que en general, tales VMMs no tienen acceso a los dispositivos reales de la plataforma y no incluyen drivers de dispositivos para el hardware.

Por otro lado, la asignación directa naturalmente reduce la complejidad en VMMs tipo 1 y aproximaciones híbridas. Esta reducción de complejidad no es posible con la emulación o la paravirtualización.

Es posible que un VMM emplee varias técnicas diferentes para la virtualización E/S de manera concurrente. Por ejemplo, en el contexto de un VMM híbrido, la asignación directa podría ser usada para asignar un dispositivo físico de la plataforma a un guest particular, cuya responsabilidad sea la de compartir ese dispositivo con varios guests. Dependiendo de las necesidades y requerimientos del guest, podría ofrecer tanto modelos emulados como soluciones paravirtualizadas para los diferentes guests. Una configuración común es proveer soluciones paravirtualizadas para los entornos de guest más comunes mientras una solución emulada se ofrece a los entornos antiguos.

Virtio: framework de virtualización de Entrada/Salida para Linux

El kernel Linux soporta una amplia gama de soluciones de virtualización que continúa creciendo a medida que la virtualización avanza y se descubren nuevos esquemas. Pero con todas las formas de virtualización que se ejecutan sobre Linux, rápidamente surgió la necesidad de estandarización de los mecanismos de comunicación para la virtualización de E/S. La respuesta fue la creación de una nueva capa de software llamada *virtio*.

Linux es el campo de juego para las nuevas ideas de virtualización y se proyecta como la solución de base para todas las implantaciones en producción en el futuro cercano, ofreciendo

la mayor variedad en soluciones de hipervisor con diferentes atributos y funcionalidades [PRED1], [PRED2]. Ya vimos en este capítulo que la virtualización de dispositivos es un tema central que determina la performance y complejidad general de toda la implementación. A su vez, cada técnica puede ser ideal para algunos escenarios y no aplicable en otros. Esta diversidad podría rápidamente crecer hasta hacerse inmanejable para los responsables del código del kernel Linux. Afortunadamente, virtio proporciona un front-end común que estandariza la interfaz y aumenta las posibilidades de reutilización del código de virtualización de dispositivos en distintas plataformas.

Virtio proporciona una capa de abstracción para hipervisores y un conjunto común de drivers de virtualización de E/S. Es decir, virtio es un nivel de abstracción sobre dispositivos en un hipervisor paravirtualizado. Fue desarrollado por Rusty Russell e implementa una idea compartida por otras soluciones alternativas. Xen proporciona drivers de dispositivos paravirtualizados y VMware aporta las denominadas *Guest Tools (herramientas de huéspedes).*

Resumiendo lo visto en la sección anterior, en el esquema de virtualización total el hipervisor debe emular el hardware del dispositivo, es decir, realizar la emulación en el nivel más bajo de la conversación. La abstracción en esta emulación es limpia, pero también es de alta complejidad y es la más ineficiente de todas. En el esquema de paravirtualización, el guest y el hipervisor pueden cooperar para lograr una emulación eficiente. La desventaja del enfoque de paravirtualización consiste en que el sistema operativo sabe que está siendo virtualizado y requiere modificaciones para funcionar.

El hardware se mantiene en constante cambio en relación con la virtualización. Los nuevos procesadores incorporan instrucciones avanzadas para mejorar la eficiencia de las transiciones entre los hipervisores y sistemas operativos guests. Además, el hardware continúa cambiando para facilitar la virtualización de E/S, como se explica en el próximo capítulo. Sin embargo, en entornos tradicionales de virtualización total, el hipervisor debe capturar las solicitudes y luego emular los comportamientos del hardware real como muestra el sector izquierdo de la figura 3.

El sector derecho de la figura muestra un caso de paravirtualización. En éste, el sistema operativo guest sabe que se está ejecutando en un hipervisor e incluye drivers que actúan como front-end. El hipervisor implementa los drivers back-end para cada emulación de dispositivo en particular. *Virtio* participa en estos drivers de front-end y back-end proporcionando una interfaz estandarizada que reduce la complejidad de desarrollo de dispositivos virtualizados a partir de la técnica de paravirtualización.

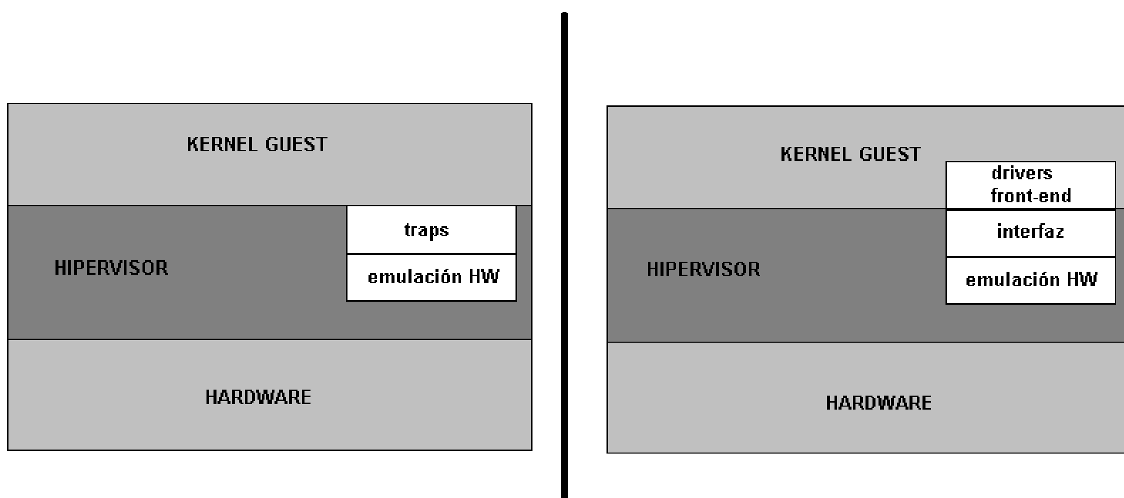


Figura 6.3 - Emulación de dispositivos en entornos de virtualización total y paravirtualización

Abstracción para guest de Linux

Virtio es una abstracción para un conjunto de dispositivos virtuales comunes en un hipervisor de paravirtualización. Este diseño permite al hipervisor exportar un conjunto común de dispositivos y ponerlos a disposición a través de una API conocida. La próxima figura muestra por qué esto resulta importante. Con los hipervisores paravirtualizados, los guests implementan un conjunto común de interfaces y la emulación de dispositivos particulares se realiza detrás de un conjunto de drivers de back-end. No es necesario que los drivers de back-end sean comunes, siempre y cuando estos implementen los comportamientos requeridos por el front-end.

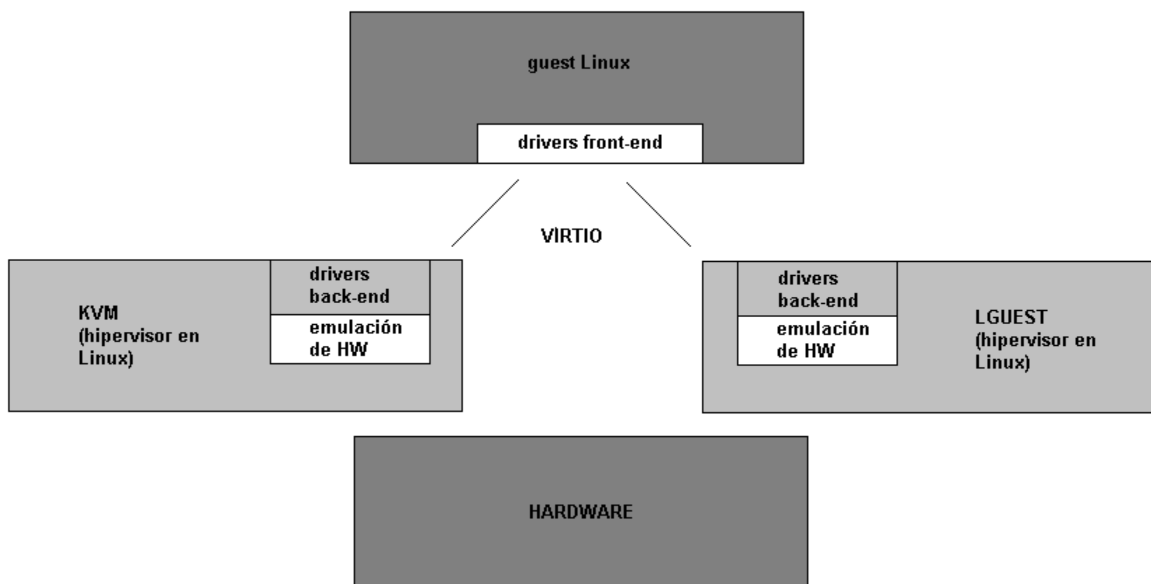


Figura 6.4 - Abstracciones de drivers con virtio

En el caso de KVM, la emulación de dispositivos ocurre en el espacio de usuario usando QEMU [QEMU], a fin de que los drivers de back-end se comuniquen dentro del espacio de memoria del hipervisor para facilitar la E/S a través. QEMU es un emulador que, además de brindar una plataforma de virtualización de sistemas operativos, proporciona emulación para sistemas completos (controladores de PCI, discos, redes, hardware de video, controladores USB y otros elementos de hardware).

La API de virtio usa una simple abstracción de buffer para encapsular las necesidades de comandos y datos del guest. A continuación se exploran los componentes internos de la API de virtio.

Arquitectura de virtio

Además de los drivers de front-end (implementados en el sistema operativo guest) y los drivers de back-end (implementados en el hipervisor), *virtio* define dos capas para soportar la comunicación del guest al hipervisor.

En el nivel superior (denominado *virtio*) encontramos la interfaz de cola virtual que comunica conceptualmente los drivers de front-end a los drivers de back-end. Los drivers pueden usar colas o no, dependiendo de las necesidades de cada uno. Por ejemplo, el driver de red virtio usa dos colas virtuales (una de recepción y otra de transmisión), mientras que el driver de bloques virtio usa una sola. Las colas, por el hecho de ser virtuales, se implementan en realidad como anillos que efectúan transversalmente la transición del guest al hipervisor. Esto podría implementarse de varias maneras, siempre que el guest y el hipervisor compartan la forma de implementación.

En la próxima figura se listan los cinco drivers front-end para dispositivos de bloques (como discos, por ejemplo), dispositivos de red, emulación PCI, un driver *balloon* [BALL] (para la gestión dinámica del uso de memoria de guest), y un driver de consola. A cada driver de front-end le corresponde un driver de back-end del hipervisor.

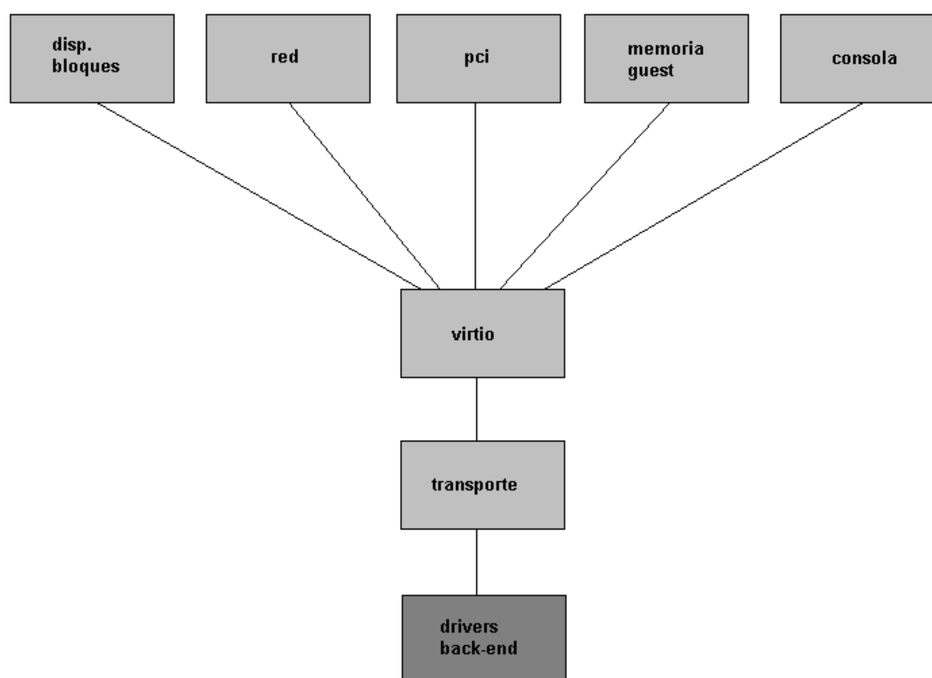


Figura 6.5 - Arquitectura de virtio

Bibliografía

[ZABAL] Performance y Escalabilidad del Kernel Linux aplicado a Redes de Alta Velocidad. Matías Zabaljáuregui. Trabajo final para obtener el grado de Licenciado en Informática de la Facultad de Informática, Universidad Nacional de La Plata, Argentina. Director: Lic. Javier Díaz. Co-director: Ing. Luis Marrone. La Plata, Abril de 2007.

[CBAC] http://es.wikipedia.org/wiki/Callback_(informática)

[PRED1] http://vmblog.com/archive/2010/12/17/gluster-prediction-2011-the-future-of-virtualization.aspx

[PRED2]
http://www.thewoodlandswebdesign.com/development/news/latest/vmware_linux_is_ideal_os_for_virtualization.html

[QEMU] http://www.qemu.org/

[BALL] Memory Resource Management in VMware ESX Server Carl A. Waldspurger. VMware, Inc.. Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI), Boston, MA, December 2002.

[VIRTIO] virtio: towards a de-facto standard for virtual I/O devices. Rusty Russell IBM OzLabs, Canberra, Australia. Newsletter ACM SIGOPS Operating Systems Review - Research and developments in the Linux kernel archive Volume 42 Issue 5, July 2008

[PASS] Linux virtualization and PCI passthrough. *Device emulation and hardware I/O virtualizatio.* M. Tim Jones, Independent author. Date: 13 Oct 2009. http://www.ibm.com/developerworks/linux/library/l-pci-passthrough/index.html

7. Virtualización asistida por hardware

En este capítulo se discuten los recientes cambios arquitecturales que permiten la virtualización clásica de la arquitectura x86. La discusión se aplica tanto a la solución de AMD como a la de Intel; la similitud entre estas dos arquitecturas es evidente. Aunque, en el caso de explicar funcionalidades específicas, se elegirá la implementación de Intel, por disponer el autor del hardware para la experimentación.

En los últimos años los fabricantes de hardware se han adaptado rápidamente a las nuevas ideas relacionadas con la virtualización y han desarrollado nuevas características para simplificar las técnicas del software de virtualización.

En el año 2005 y 2006, Intel y AMD (trabajando independientemente) crearon nuevas extensiones para los procesadores de la arquitectura x86. La primera generación de procesadores con soporte para virtualización resolvía el problema de las instrucciones privilegiadas pero no ofrecía soporte para la virtualización de la MMU. Como resultado, la performance no era mejor que la de las técnicas de traducción binaria o paravirtualización, aunque simplificaba la implementación permitiendo las técnicas clásicas de *trap and emulate*.

AMD desarrolló su primer generación de extensiones de virtualización bajo el nombre código *Pacifica*, e inicialmente lo publicó como *AMD Secure Virtual Machine (SVM)*, pero luego fue comercializado bajo la marca *AMD Virtualization,* abreviado *AMD-V.* En mayo de 2006, AMD lanza al mercado el *Athlon 64* (*Orleans*), el *Athlon 64 X2* (*Windsor*) y el *Athlon 64 FX* (*Windsor*) como los primeros procesadores en soportar esta tecnología.

Las CPUs *Opteron* de AMD, comenzando con la familia de la línea *Barcelona*, y los CPUs *Phenom II*, soportan una segunda generación de virtualización por hardware llamada *Rapid Virtualization Indexing* (RVI) (conocida como *Nested Page Tables* durante su desarrollo), adoptada luego por Intel como *Extended Page Tables* (EPT).

Por otro lado, *VT-x*, antes conocido con su nombre código *Vanderpool*, representa la tecnología de virtualización de x86 de Intel. La inclusión de las EPT, una tecnología para virtualización de tablas de página, sucedió en la micro-arquitectura *Nehalem*. El primer procesador con esta arquitectura fue el *Core i7*, lanzado en noviembre de 2008.

Primera Generación

Ambos fabricantes resuelven el problema de las instrucciones privilegiadas con un nuevo modo de ejecución de la CPU que permite que el VMM se ejecute en un nivel más privilegiado que el *ring 0* del guest. Las instrucciones críticas y privilegiadas se configuran para, automáticamente, producir una transición al hipervisor, liberándose de la necesidad tanto de traducción binaria como de paravirtualización.

El hardware exporta un número de primitivas nuevas para soportar VMMs clásicos para la arquitectura x86. Una estructura de datos en memoria, a la cual se suele denominar genéricamente como Bloque de Control de la Máquina Virtual (VMCB, por sus siglas en inglés) combina el estado de control con un subconjunto del estado de la CPU virtual del guest.

Un nuevo modo de ejecución menos privilegiado, el modo guest, soporta la ejecución directa en el hardware del código del guest, incluyendo a su código privilegiado. Y se denomina modo host al modo de funcionamiento de la CPU diseñado para ejecutar el VMM.

Una nueva instrucción, VMRUN, transfiere el control del modo host al modo guest y el hardware carga en los registros de la CPU el estado del guest desde el VMCB, continuando a partir de allí su ejecución en modo guest. A esta operación se la denomina genéricamente *VM Entry* (la documentación de ambos fabricantes puede no coincidir en algunos términos).

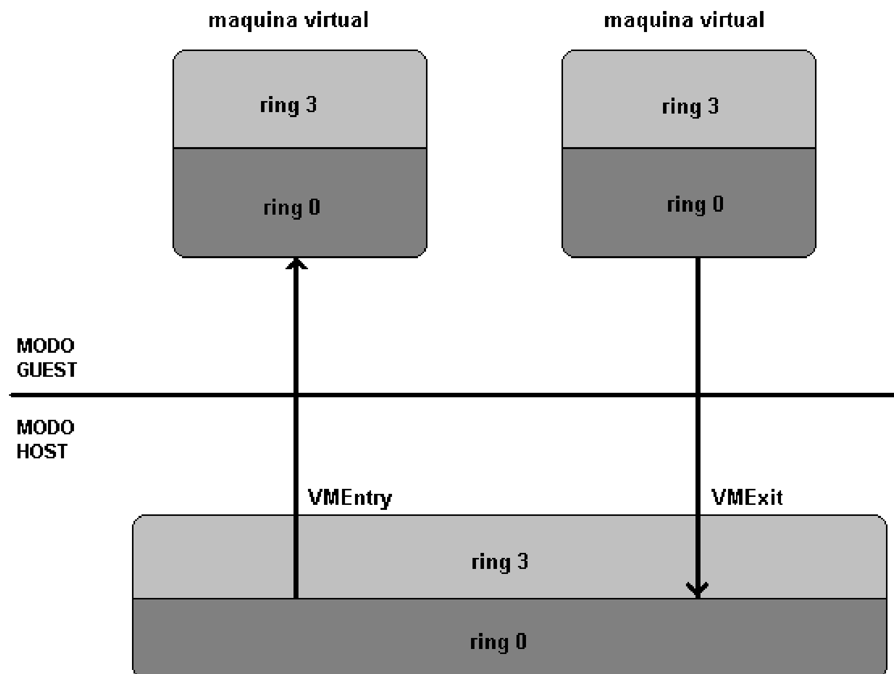


Figura 7.1 – Modo Guest y Modo Host

La ejecución del guest procede hasta que alguna condición, expresada por los bits de control que el VMM configuró en el VMCB, es alcanzada. En este punto, el hardware realiza una operación inversa al Entry, denominada *VM Exit*, en la cual guarda el estado del guest en el VMCB, carga el estado previo del VMM en los registros y retoma la ejecución en modo host, para ejecutar el VMM.

Los campos de diagnóstico en el VMCB asisten al VMM para manejar el exit; por ejemplo, en los exits producidos por los intentos del guest de realizar operaciones de E/S, los campos de diagnóstico proveen el puerto, el ancho (en bytes) y la dirección de la operación de E/S. Después de emular sobre el VMCB el efecto de la operación que produjo el Exit, el VMM retoma la ejecución del guest.

VMCB

Los bits de control del VMCB proveen cierta flexibilidad en el nivel de confianza depositado en el guest. Por ejemplo, como se estudió en el capítulo anterior, un VMM comportándose como un hipervisor para un sistema operativo de propósito general podría permitir que ese SO controle dispositivos físicos de la plataforma, que maneje las interrupciones o que construya sus propias tablas de página. Sin embargo, cuando se desea aplicar la asistencia del hardware para implementar virtualización completa, las libertades del guest se ven notablemente limitadas: el VMM programa el VMCB para escapar en fallos de página del guest, en limpiezas de TLB (TLB flush) y en cambios de espacio de direccionamientos (escrituras del CR3) para mantener las tablas de páginas shadow; se programa también para escapar ante instrucciones de E/S para ejecutar los modelos emulados de dispositivos para el guest; y también debería programarse para escapar en accesos a estructuras de datos privilegiadas como tablas de páginas y dispositivos mapeados en memoria.

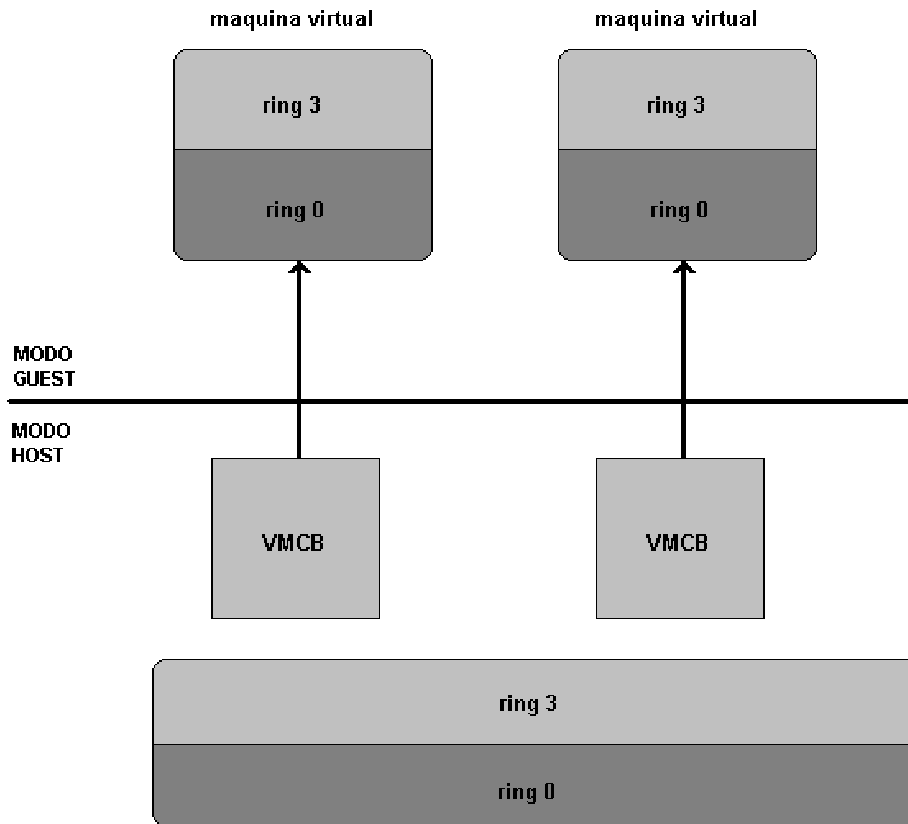


Figura 7.2 – Virtual Machine Control Block o VMCB

El área del guest del VMCB contiene elementos del estado de la CPU virtual asociada con ese VMCB. Por ejemplo:

- Los registros de segmento, para mapear desde direcciones lógicas a direcciones lineales.
- El registro CR3, que apunta a la jerarquía de tablas de páginas para mapear desde las direcciones lineales a las direcciones físicas.
- El *Interrupt Descriptor Table Register* (IDTR), para la entrega de eventos y otros campos que no se corresponden con ningún registro accesible por el software.

Por otro lado, el VMCB contiene un número de campos que controlan la operación en modo guest, al especificar las instrucciones y eventos que causan VM Exits:

a) Controles que soportan la virtualización de las interrupciones:

- Habilitación de interrupciones externas: si está activado, todas las interrupciones externas causan VM Exits. El guest no es capaz de enmascarar estas interrupciones.
- Ventana de interrupciones de escape: si está activado, ocurrirá un VM Exit cuando el guest está preparado para recibir interrupciones
- Uso de un *Task Priority Register* (TPR) shadow: si está activado, los accesos a la TPR de la APIC a través del registro de control CR8 se manejan de manera especial: las ejecuciones de MOV CR8 acceden a un TPR shadow referenciado por un puntero en el VMCB. El VMCB también incluye un umbral para el TPR; ocurre un VM Exit después de cualquier instrucción que reduzca el TPR shadow por debajo del umbral de TPR (Intel llama Flex Priority a esta capacidad).
- Virtualización del CR0 y el CR4.

b) Mapa de bits de excepciones: 32 entradas para las excepciones de IA-32. Para especificar qué excepciones deberían causar VM exits y cuáles no.

c) Mapa de bits para E/S: una entrada para cada puerto en el espacio de E/S de 16 bits. Una E/S causa una VM exit si intenta acceder a un puerto cuya entrada está activada en el mapa de bits de E/S.

d) Mapa de bits MSR: dos entradas (lectura y escritura) para cada Registro Específico del Modelo (MSR por sus siglas en inglés, Model-Specific Register) usado. La ejecución de RDMSR (o WRMSR) causa un VM exit si intenta leer (o escribir) un MSR cuyo bit de lectura (o escritura está activado en el mapa de bits de MSR.

Ejemplo de operación: creación de un proceso

Consideremos un sistema operativo tipo UNIX ejecutándose en modo guest y un proceso en espacio de usuario que invoca a la system call fork(). La llamada al sistema cambia el CPL desde 3 a 0, la transición sucede sin la intervención del VMM.

En la implementación del fork, el guest utiliza la aproximación, llamada *copy on write,* que consiste en proteger contra escritura los espacios de direcciones del padre y el hijo. El VMM ya ha creado las tablas de página shadow para el espacio de direccionamiento del proceso padre, usando trazos para mantener la coherencia (capítulo 3). Por lo tanto cada escritura del guest en sus tablas de páginas causará un exit. El VMM decodifica la instrucción que genera el exit para emular sus efectos en las páginas del guest y refleja este efecto en las tablas de páginas shadow. Al actualizar la tabla de página shadow, el VMM protege contra escritura el espacio de direccionamiento del proceso padre.

El planificador del sistema operativo guest descubre que el proceso hijo se encuentra en estado ejecutable y cambia de contexto. Carga el registro puntero al mapa de memoria a la tabla de páginas del hijo, causando un exit. El VMM construye una nueva tabla de páginas shadow y apunta el registro de tabla de páginas del VMCB a ésta.

Mientras el proceso hijo se ejecuta, accede a porciones de su espacio de direccionamiento que no están mapeadas aún en sus tablas de páginas shadow. Esto causa exits por fallos de páginas escondidos. El VMM intercepta los fallos de página, actualiza su tabla de páginas shadow y retoma la ejecución del guest.

Mientras ambos procesos, padre e hijo, se ejecutan, escriben en nuevos lugares de memoria causando nuevamente fallos de páginas. Estos fallos son reales (no escondidos) y reflejan las políticas de protección impuestas por el guest. El VMM debe interceptarlas antes de poder inyectarlas al guest, para asegurarse de que no son artefactos del algoritmo de shadowing.

Performance

Las extensiones VT-x y AMD-V hacen que sea posible la virtualización clásica sobre arquitecturas x86. La performance resultante depende principalmente de la frecuencia de exits. Un guest que nunca genera exits se ejecuta a velocidad nativa, incurriendo en un costo adicional prácticamente nulo. Sin embargo, este guest no sería demasiado útil ya que no puede realizar operaciones de E/S. Si por otro lado, cada instrucción del guest dispara un exit, el tiempo de ejecución estará dominado por las transiciones del hardware entre el modo guest y modo host. La reducción de la frecuencia de transiciones de exit es la optimización más importante para VMMs clásicos.

Para ayudar a evitar los exits más frecuentes, la asistencia del hardware incluye ideas similares a la funcionalidad de ejecución interpretativa del s/370 (ver capítulo 3). Cuando es posible, las instrucciones privilegiadas afectan al estado dentro de la CPU virtual tal como se representa en el VMCB, en lugar de generar traps de manera incondicional.

Consideremos el caso de POPF. Una implementación ingenua de x86 para soportar virtualización clásica habría disparado un exit en todas las ejecuciones de POPF en modo guest para permitir que el VMM actualice la instancia virtual del bit de interrupciones. Sin embargo, los guests suelen ejecutar POPF muy frecuentemente, llevando a una cantidad de exits inaceptable. En su lugar, el VMCB incluye un registro %eflags shadow mantenido por el hardware. Cuando la CPU se encuentra en modo guest, las instrucciones que operan sobre %eflags, lo hacen sobre la versión shadow, liberándonos de la necesidad de exits.

Resumiendo, la tasa de exits es una función del comportamiento del guest, el diseño del hardware y el diseño del VMM: un guest que sólo realiza cómputo no necesita generar exits; el hardware provee mecanismos para controlar algunos tipos de exit; y las decisiones de diseño del VMM, en particular el uso de trazos y técnicas shadow, directamente impactan en la tasa de exits como se mostró en el ejemplo de fork.

La mayoría de las dificultades de performance de la asistencia por hardware se relacionan con la virtualización del MMU y con el manejo de dispositivos en el caso de la emulación. Es por ésto que la segunda generación de tecnologías de asistencia a la virtualización por hardware intentó resolver algunas de estas cuestiones, presentando soluciones a la virtualización de memoria y dispositivos, entre otras funcionalidades adicionales.

Segunda generación de procesadores con extensiones para la virtualización

Tanto Intel como AMD han continuado con la rápida evolución de extensiones del hardware para el soporte de plataformas diseñadas para la virtualización. A partir de la tercer generación de procesadores Opteron, con nombre código Barcelona, AMD ofreció nuevas características de virtualización de MMU, llamadas actualmente *Rapid Virtualization Indexing (RVI)*. Según estudios de los ingenieros de VMware, RVI ofrece hasta un 42% de mejora en la performance [VMRVI], comparado con las implementaciones por software de tablas de páginas shadow. Red Hat por su parte, mostró una mejora del doble de velocidad con sus benchmarks OLTP [RHRVI]. Intel comenzó a incluir algunas de estas características en sus CPUs basadas en el núcleo Nehalem; en particular Core i7, Core i5, Core i3, Pentium G6950 y algunos procesadores Xeon.

A continuación se describen conceptualmente estas nuevas técnicas. En particular, se estudian las implementaciones de Intel, recalcando que AMD suele ofrecer técnicas análogas con nombres distintos. En primer lugar se listan las nuevas mejoras a las extensiones de VT-x. Estas son: CPUID spoofing (también conocida como *Flex Migration*), Extended Page Table (EPT) y Virtual Processor IDs (VPID).

Manipulación de la instrucción CPUID

Esta extensión se ubica entre dos capacidades importantes: por un lado, Intel ha mantenido una histórica tradición de agregar nuevas características a su omnipresente ISA a lo largo de los años, tales como las Extensiones Multimedia [MMX] (MMX, por *MultiMedia eXtension)* y varias generaciones de Extensiones SIMD [SSE] (SSE, por Streaming SIMD Extensions). Al mismo tiempo, los productores de VMMs están en la carrera por lograr exitosamente la migración de una máquina virtual desde una plataforma física a otra, lo cual es extremadamente útil para el manejo de recursos y el balaceo de carga en un centro de cómputos. Esta migración, en el mejor de los casos, debería ser ´en caliente´ o *live*, es decir, sin tener que desactivar la máquina virtual para realizar la migración.

La migración de máquinas virtuales es uno de los temas más interesantes y técnicamente complejos de estos últimos tiempos. Ha sido difícil lograr una implementación exitosa de migración en caliente de máquinas virtuales entre servidores basados en diferentes generaciones de procesadores, cada uno con diferentes conjuntos de instrucciones. Una forma de intentar evitar este problema es particionar el centro de cómputos en 'islas' de servidores

tales que cada una comparte un conjunto de características idéntico y luego limitar la migración de máquina virtual sólo entre servidores dentro de estas islas.

Hay otra aproximación a la solución del problema y se construye sobre la forma en que Intel reporta la disponibilidad de nuevas características del ISA vía una instrucción conocida como CPUID. Esta instrucción le permite al software descubrir el conjunto exacto de nuevas características provistas por cualquier implementación de procesador. Por ejemplo, la instrucción CPUID retorna un conjunto de bits que indican si el procesador ofrece SSE3 o SS4.

Si un VMM tiene una forma de interceptar las ejecuciones de CPUID por parte del guest, entonces podría manipular los bits retornados y engañar al guest para que vea un conjunto de características diferentes. Por ejemplo, podría hacer que todos los sistemas del centro de cómputos sólo soportaran el conjunto de instrucciones SS3 (incluso aunque algunos podrían soportar SS4). Luego, las VMs podrían migrar libremente entre todos los sistemas del centro de cómputos sin generar problemas.

Antes no había una forma sencilla para que el VMM controlara los valores retornados al guest por la instrucción CPUID. La segunda generación de procesadores para virtualización ofrece un conjunto de mecanismos para que la modificación de los bits de CPUID esté bajo el control del VMM, ya sea a través de una máscara de bits que limita el reporte de nuevas características o dándole al VMM una forma de interceptar la instrucción CPUID y emular los valores de retorno alternativos. Claramente, se presenta un compromiso: para permitir la migración libre de VMs entre plataformas físicas diferentes, debe elegirse un subconjunto de características que sea un denominador común, lo que puede significar resignar el uso de algunas de las nuevas extensiones del ISA.

Extended Page Table (EPT)

Tanto RVI de AMD como ETP de Intel intentan resolver el que demostró ser el principal problema de performance en la primera generación de extensiones de virtualización asistida por hardware: la virtualización del MMU. Estas soluciones mejoran notablemente la performance y reducen la complejidad del VMM al liberarlo de tener que implementar las técnicas de shadowing, usualmente, la parte más compleja del VMM por software.

En ambos esquemas, el VMM mantiene una tabla de páginas anidada utilizada por el hardware para traducir direcciones físicas del guest a direcciones de máquina (o direcciones físicas del host). Este mapeo permite que el hardware dinámicamente maneje las operaciones de MMU del guest, eliminando la intervención del VMM.

El VMM mantiene las asociaciones entre esas direcciones en un nivel adicional de tablas de páginas (anidadas). En este caso, tanto las tablas de páginas del guest como las tablas de páginas anidadas se exponen al hardware a través de registros de la CPU.

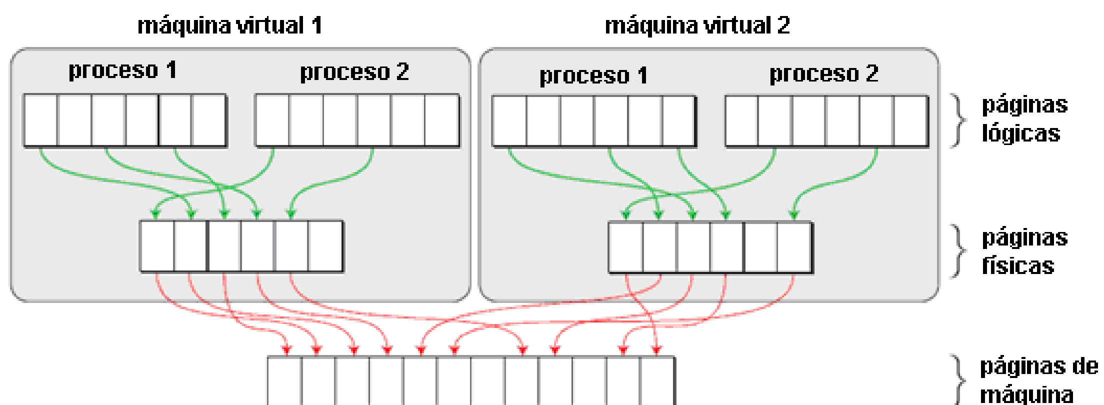
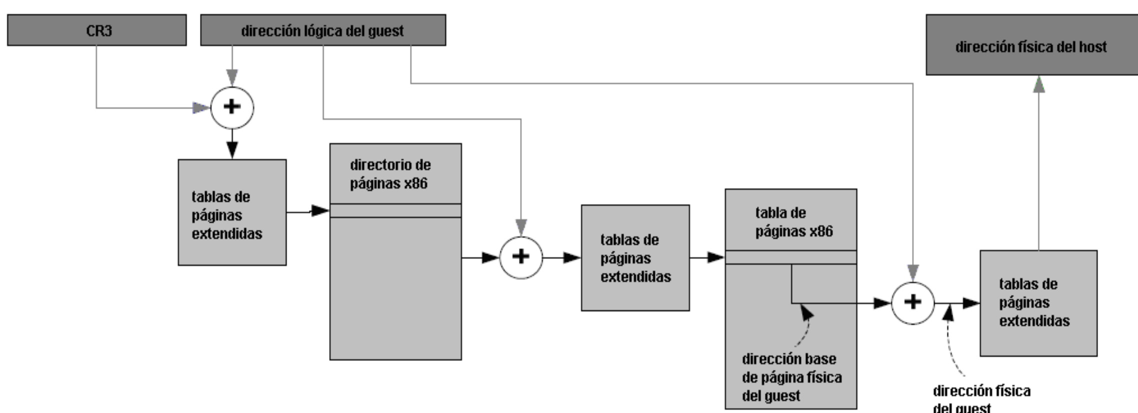


Figura 7.3 – Tablas de páginas en una arquitectura virtualizada

Cuando se accede a una dirección virtual del guest, el hardware recorre las tablas de páginas del guest como en el caso de la ejecución nativa, pero para cada página física del guest accedida durante el recorrido por las tablas del guest, el hardware debe recorre las tablas de páginas anidadas para determinar la dirección de memoria física del host.

Esta doble traducción elimina la necesidad de mantener tablas de páginas shadow y tener que sincronizarlas con las tablas de páginas del guest. Sin embargo, la operación extra también incrementa el costo de un recorrido por una tabla de páginas, impactando de esta manera en la performance de las aplicaciones que estresan la TLB. Este costo puede ser reducido con el uso de páginas grandes, reduciendo el uso de la TLB para aplicaciones con buena localidad espacial.



7.4 – Tablas de Páginas Extendidas

Cuando se ejecuta un guest sobre un VMM asistido por hardware, la TLB contiene entradas que mapean direcciones virtuales del guest a direcciones físicas del host. El proceso de completar la TLB en el caso de un fallo es por lo tanto más complicado que el de un sistema de memoria virtual tradicional.

Virtual Processor IDs (VPID)

Cada entrada en la TLB cachea una traducción de dirección virtual del guest a dirección de máquina (física del host) para una página en memoria, y la traducción es específica de un proceso y máquina virtual dados. Las CPUs más antiguas limpiaban la TLB cuando el procesador cambiaba entre una instancia de guest virtualizado y el VMM, para asegurarse que los procesos sólo accedieran a la memoria que se les permitía acceder.

La idea general de una TLB etiquetada no es nueva y significa identificar, de alguna manera, cada entrada en la TLB. Normalmente, las entradas en una TLB de x86 no están asociadas con ningún espacio de direccionamiento virtual. Es por esto que, cada vez que hay un cambio de espacio de direccionamiento, por ejemplo, un cambio de contexto, la TLB entera debe ser limpiada.

Mantener una etiqueta en software que asocie cada entrada de TLB con un espacio de direccionamiento y comparar por software esta etiqueta durante una búsqueda (o limpieza) en TLB es muy costoso, especialmente porque la TLB de x86 está diseñada para operar con una muy baja latencia y completamente por hardware.

En el año 2008, tanto Intel como AMD introdujeron etiquetas como parte de las entradas de TLB y dedicaron una cantidad de hardware al chequeo de las etiquetas durante las búsquedas. Aunque todavía no son explotadas completamente, se prevé que en el futuro estas etiquetas identificarán el espacio de direccionamiento al cual pertenece cada entrada en la TLB. Por lo

tanto un cambio de contexto no resultará en una limpieza total de la TLB, sino que sólo se reemplazará el identificador de etiqueta del espacio de direccionamiento actual por el del próximo proceso/contexto.

En el caso de la virtualización, la etiqueta es un Virtual Procesor Identifier (VPID) que distingue distintos espacios de direccionamiento. El host se ejecuta con el VPID cero mientras que cada CPU virtual tiene su propio VPID distinto de cero. De esta forma se evita tener que limpiar la TLB completa cada vez que hay una transición entre host y guest. Y la CPU física usa los VPIDs de las entradas de TLB para evitar la compartición involuntaria de TLB (*TLB sharing*).

El VPID indica con cuál máquina virtual está asociada una traducción dada en una entrada de TLB, para que cuando ocurra una VM Exit y una posterior re-entrada, las TLBs no tengan que ser limpiadas completamente por seguridad. En su lugar, si un proceso intenta acceder a una traducción con la que no está asociado, simplemente generará un fallo de TLB. El VPID es útil para mejorar la performance de virtualización al reducir el costo de las transiciones de las máquinas virtuales**;** Intel afirma que la latencia de una transición completa (VM Entry y VM Exit) en Nehalem es un 40% comparada con Meron (Core 2 con tecnología de 65 nm) y aproximadamente un tercio menor que el Penryn (con tecnología de 45nm).

tag virtual	dirección lógica	dirección física
Host	0x1000	0x10001000
Host	0x2000	0x10002000
Host	0x3000	0x10003000
Host	0x4000	0x10004000
Guest	0x1000	0xFFF01000
Guest	0x2000	0xFFF02000
Guest	0x3000	0xFFF03000
Guest	0x4000	0xFFF04000

Figura 7.5 – TLB con etiquetas

Unidad de manejo de memoria de entrada/salida o IOMMU

Una Unidad de Manejo de Memoria de Entrada/Salida (Input/Output Memory Management Unit o IOMMU) es una MMU que conecta un bus capaz de realizar operaciones DMA a la memoria principal. Al igual que un MMU tradicional, que traduce direcciones virtuales visibles por la CPU a direcciones físicas, el IOMMU se ocupa de mapear direcciones virtuales visibles por los dispositivos (también llamadas direcciones de dispositivos o direcciones de E/S, en este contexto) a direcciones físicas, como se muestra en la siguiente figura. Algunas unidades también proveen protección de memoria.

AMD ha publicado una especificación para la tecnología IOMMU en su arquitectura *HyperTransport*. Intel publicó otra especificación de IOMMU con el nombre de *Virtualization Technology for Directed I/O* [INVTD], abreviada VT-d. El PCI-SIG ha realizado cierto trabajo relevante bajo los términos *I/O Virtualization* (IOV) y Servicios de Traducción de Direcciones (ATS, por las siglas de Address Translation Services).

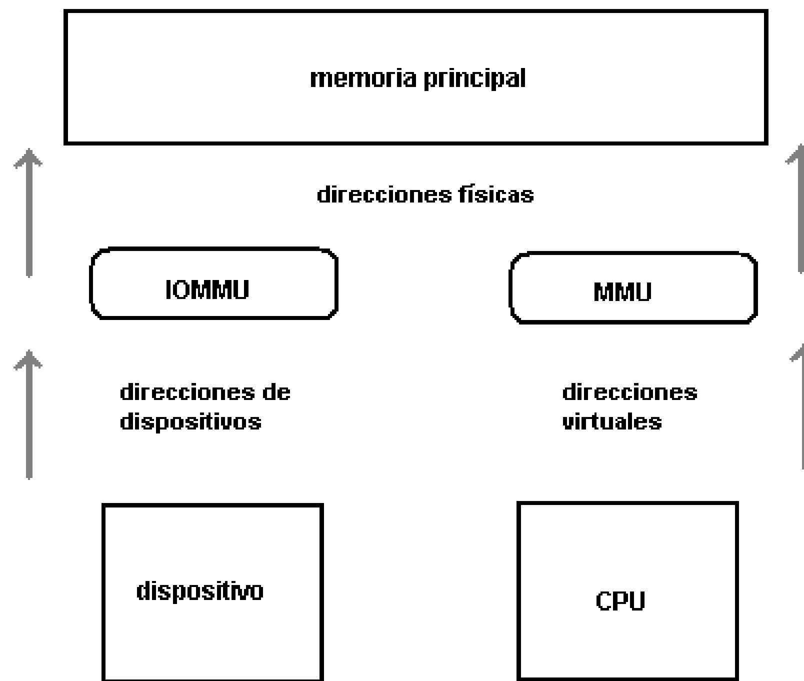


Figura 7.6 – IOMMU vs MMU

Las ventajas de tener un IOMMU, comparado con el direccionamiento físico directo de la memoria, pueden resumirse de esta forma:

8. Pueden alocarse grandes regiones de memoria virtual sin ser contiguas en la memoria física. De esta forma puede evitarse la E/S vectorizada (operaciones *scatter-gather*), que podrían reducir la performance de ciertas operaciones de E/S [ZABAL].
9. Los dispositivos que no soportan direcciones lo suficientemente grandes, pueden acceder igualmente a todo el espacio de direcciones, evitando el overhead por copia de buffers en memoria. Por ejemplo, un dispositivo PCI de 32 bits no puede acceder a la memoria por encima de los 4Gb, obligando a los sistemas operativos a usar los denominados *bounce buffers* (o *double buffers* en sistemas windows).
10. Protección de la memoria de dispositivos en mal funcionamiento o maliciosos.

Las desventajas de tener un IOMMU se relacionan con la performance. Principalmente el costo adicional generado por la traducción y el manejo de las tablas de páginas y el mayor consumo de memoria por las tablas de traducción adicionales. Pueden estudiarse los detalles de los costos de performance en [IOPER].

Con respecto a la virtualización, los dispositivos de mayor performance, como las interfaces de red, usan DMA para el acceso directo a la memoria. Cuando un SO está corriendo dentro de una máquina virtual, no conoce las direcciones físicas de la memoria a la que accede. Esto hace que proveer acceso directo al hardware de la plataforma sea complicado, porque si el guest intentara instruir al hardware para realizar un acceso DMA, corrompería la memoria ya que el hardware no conoce el mapeo entre las direcciones virtuales y reales usado por el SO guest.

La corrupción de memoria suele evitarse gracias a que el hipervisor interviene en la operación de E/S para realizar la traducción; desafortunadamente esto introduce latencia en la operación de E/S y consume ciclos de CPU. Un IOMMU resuelve el problema re-mapeando las direcciones accedidas por el hardware de acuerdo a la misma tabla de traducción (o alguna dirección compatible) usada por la el guest.

La implementación de Intel: VT-d (Intel Virtualization Technology for Directed I/O)

VT-d de Intel es una característica integrada en el chipset y por lo tanto no relacionada con la CPU. Antes que apareciera VT-d y el soporte correspondiente por parte de los hipervisores, una máquina virtual trabajaba generalmente con dispositivos emulados o paravirtualizados, o resolvía la asignación directa a través de mecanismos de software complejos (ver capítulo 6). Cuando VT-d está habilitado, el guest puede elegir usar o bien la aproximación tradicional o, si es necesario, el modelo de acceso *pass-through*. En este último modo, el dispositivo PCI no es alocado por el hipervisor y, por lo tanto, puede ser alocado directamente por una máquina virtual que ahora tiene acceso directo al dispositivo físico, como muestra la figura siguiente.

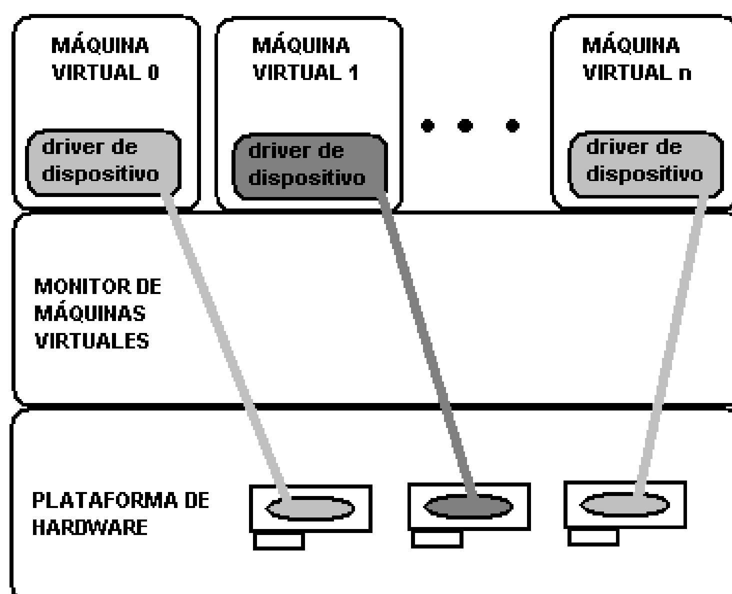


Figura 7.7 – Asignación directa de dispositivos

Re-mapeo DMA

La arquitectura VT-d es una arquitectura IOMMU generalizada que habilita al software para crear múltiples dominios de protección DMA. Un dominio de protección se define abstractamente como un entorno aislado para el cual se aloca un subconjunto de la memoria física del host. Dependiendo del modelo de uso del software, un domino de protección DMA puede representar memoria alocada a una máquina virtual o la memoria DMA alocada por el driver de un SO guest o como parte del mismo VMM.

La arquitectura VT-d permite que el software asigne uno o más dispositivos de E/S a un dominio de protección. La aislación DMA se consigue al restringir el acceso a la memoria física del dominio de protección por parte de los dispositivos no asignados a él, a través de tablas de traducción de direcciones.

Los dispositivos de E/S asignados a un dominio pueden ser provistos con una imagen de la memoria que puede ser diferente de la del host. El hardware VT-d trata a las direcciones especificadas en un requerimiento DMA como Direcciones Virtuales DMA (DVA, DMA Virtual Address)

Dependiendo del modo de uso, una DVA puede ser una dirección física del guest (GPA, Guest Physical Address) al cual se le asignó el dispositivo de E/S, o alguna dirección virtual abstracta (similar a las direcciones lineales de la CPU). VT-d transforma la dirección de un requerimiento

DMA realizada por un dispositivo de E/S a su dirección física del host (HPA, Host Physical Address) correspondiente.

La figura ilustra la traducción de direcciones DMA en un modelo de uso multi-dominio. Los dispositivos de E/S 1 y 2 están asignados a los dominios 1 y 2, respectivamente, cada uno con su propia visión del espacio de direcciones DMA.

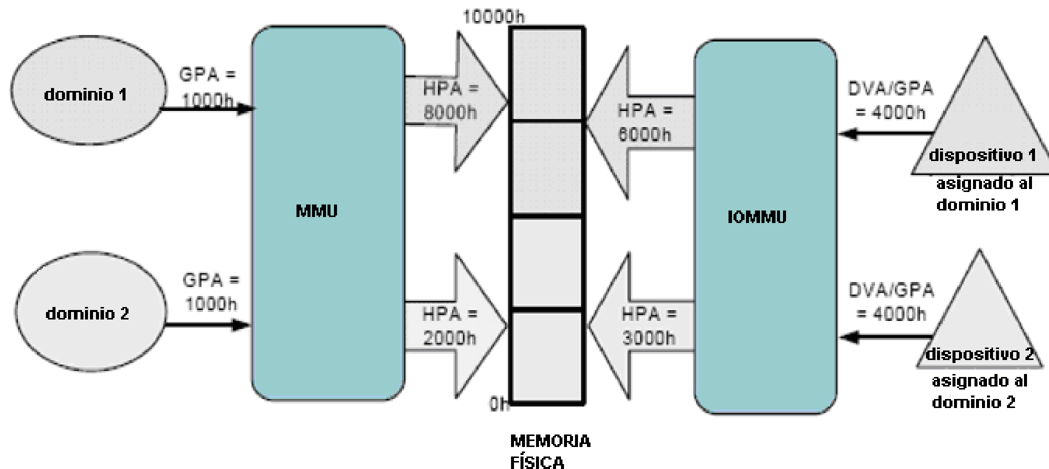


Figura 7.8 – Uso de la memoria en VT-d

Para soportar los múltiples dominios de protección, el hardware de re-mapeo DMA debe identificar el dispositivo que origina el requerimiento. El identificador del dispositivo se compone de su número de Bus/Dispositivo/Función asignado por el software de configuración PCI y que identifica unívocamente la función de hardware que inició el requerimiento, según la especificación PCI.

La estructura para la traducción de direcciones DMA se define como una tabla de páginas multi-nivel, similar a las tablas de páginas de los procesadores IA-32, permitiendo el uso de páginas de 4KB o de granularidad mayor. El hardware implementa la lógica de recorrido de la estructura.

Re-mapeo de las interrupciones

Los requerimientos de interrupciones generados por los dispositivos de E/S deben ser controlados por el VMM. Cuando la interrupción sucede, el VMM debe presentar la interrupción al guest. Esto no era realizado por el hardware y requería intervención del VMM.

La arquitectura de re-mapeo de interrupciones de VT-d resuelve este problema al redefinir el formato del mensaje de interrupción. El nuevo mensaje de interrupción contiene un identificador del dispositivo que requiere la interrupción y un identificador de interrupción, y el hardware de re-mapeo transforma estos mensajes en interrupciones físicas.

La arquitectura soporta el re-mapeo de mensajes de interrupciones desde todas las fuentes incluyendo controladores de interrupciones de E/S (IOAPIC), y todas las variantes de interrupciones MSI y MSI-X, definidas en las especificaciones PCI.

Bibliografía

[VMRVI] Performance Evaluation of AMD RVI
Hardware Assist. VMware ESX 3.5. http://www.vmware.com/pdf/RVI_performance.pdf

[RHRVI] Red Hat Enterprise Linux 5.1 utilizes nested paging on AMD Barcelona Processor to improve performance of virtualized guests. http://magazine.redhat.com/2007/11/20/red-hat-enterprise-linux-51-utilizes-nested-paging-on-amd-barcelona-processor-to-improve-performance-of-virtualized-guests/

[MMX] http://es.wikipedia.org/wiki/MMX

[SSE] http://es.wikipedia.org/wiki/SSE

[ZABAL] Performance y Escalabilidad del Kernel Linux aplicado a Redes de Alta Velocidad. Matías Zabaljáuregui. Trabajo final para obtener el grado de Licenciado en Informática de la Facultad de Informática, Universidad Nacional de La Plata, Argentina. Director: Lic. Javier Díaz. Co-director: Ing. Luis Marrone. La Plata, Abril de 2007.

[IOPER] *"The Price of Safety: Evaluating IOMMU Performance", 2007 Ottawa Linux Symposium, Volume One.*

[INVTD] *"Intel Virtualization Technology for Directed I/O"*, Inte Technology Journal, Volume 10, Issue 3.

8. Un kernel diseñado para la virtualización

Luego de participar por algunos años en el proyecto Lguest, surgió la idea que dio forma al prototipo presentado en esta tesis. Hasta el momento no existía un sistema operativo diseñado exclusivamente para ser ejecutado como un guest de un hipervisor para la arquitectura x86.

Con la experiencia adquirida en el trabajo previo, fue natural elegir a Lguest como el código base para escribir el prototipo para esta propuesta. Fue necesario adaptar los componentes paravirt_ops y virtio a un entorno genérico y construir cierta abstracción sobre ellos para sentar las bases fundamentales de cualquier implementación que pudiera plantearse a continuación.

El prototipo se ofrece como una herramienta útil para la investigación, prototipación y actividades pedagógicas relacionadas con sistemas operativos. Su interfaz queda abierta y con las licencias apropiadas para la colaboración de todo interesado en el tema, y su implementación se encuentra disponible en Internet.

El código completo del trabajo realizado al implementar el diseño propuesto fue liberado bajo una licencia GPL (General Public License). Para su fácil acceso, se utilizó un repositorio público, que suele albergar proyectos de software libre: GITHUB [GITHUB]. Puede accederse al código a través del proyecto denominado OSom, ubicado en [OSOM].

Esta implementación es la que se documenta a continuación, explicando la idea principal, el diseño y algunos detalles de implementación.

Aporte

La propuesta de esta tesis se centra en un diseño sencillo de kernel framework especializado para actuar como un guest del hipervisor Lguest, aprovechando las facilidades que nos ofrecen las dos abstracciones mencionadas anteriormente.

Claramente, este kernel no es capaz de ejecutarse sobre ninguna arquitectura de hardware nativo, ya que se construye sobre las interfaces de alto nivel mencionadas. Y justamente ésta es la gran ventaja que permite el sencillo diseño y la rápida prototipación.

El prototipo funciona como una plataforma ideal para la construcción simplificada de funcionalidades de más alto nivel usualmente implementadas en los sistemas operativos tradicionales tales como el manejo de memoria, los protocolos de red, planificación de la CPU (virtual), entre otras.

También se presenta como una herramienta útil para la investigación y experimentación en temas de sistemas operativos, y para la enseñanza en materias de grado y postgrado de sistemas operativos en general y cursos de virtualización en particular, como se propuso en [TEYET].

Finalmente, este trabajo se encuentra en cierta sintonía con el trabajo previo del autor [ZABAL], con respecto a la idea de más alto nivel relacionada con la necesidad de especializar los sistemas operativos para roles específicos. El framework aquí presentado ofrece un camino que facilita la construcción de kernels específicos, teniendo en cuenta que se pronostica la virtualización de la infraestructura como el estándar del futuro a mediano plazo.

Arquitectura

Como se explicó a lo largo de esta tesis, el kernel Linux ha sido el universo técnico ideal donde las nuevas propuestas de virtualización han sido discutidas, prototipadas y evaluadas por actores tanto de la comunidad libre como de las empresas protagonistas de este movimiento.

En particular, Linux ofrece un sofisticado framework que soporta a los distintos proyectos de paravirtualización, en un intento de estandarización de las interfaces involucradas. *Paravirt_ops* y *virtio* son los componentes centrales de este framework y han sido diseñados y perfeccionados a lo largo de estos últimos años.

Paravirt_ops facilita la conexión de un kernel Linux a cualquier hipervisor que sea compatible con esta tecnología, aislando las operaciones que pueden requerir intervención del VMM. Debe recordarse que algunos de los hipervisores modernos se implementan como componentes del propio kernel Linux, por lo que paravirt_ops suele ser en realidad una forma de conectar un kernel Linux guest con un kernel Linux actuando como host, como muestra la siguiente figura

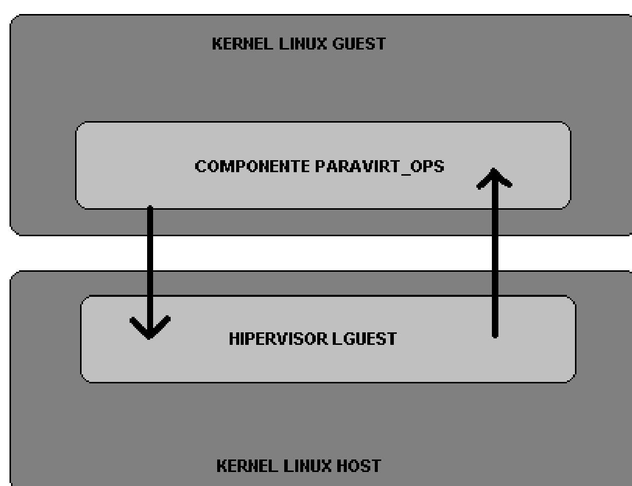


Figura 8.1 - Paravirtualización de la cpu

Paravirt_ops principalmente implementa operaciones que reemplazarán a las llamadas nativas relacionadas con la CPU. Es decir, cuestiones como la configuración de tablas de páginas, el manejo de interrupciones, la configuración de la segmentación, etc. Por este motivo suele decirse que es una solución de virtualización, o, más precisamente, paravirtualización de la CPU.

Por otro lado, virtio implementa cierta funcionalidad común en el transporte necesario entre los drivers front-end y back-end, en un escenario de dispositivos paravirtualizados. De hecho, virtio ofrece los drivers front-end para un kernel Linux guest, liberándonos, de esta forma, de la necesidad de lidiar con la complejidad de la virtualización de dispositivos, como se ve en la figura 8.2.

Es importante destacar que ambos componentes pertenecen al kernel Linux. Es decir, sus diseños fueron implementados como subsistemas conectados a otros subsistemas del mismo kernel. Paravirt_ops se coloca entre las invocaciones a llamadas de bajo nivel y la implementación de estas mismas llamadas para poder, eventualmente, desviar el control a implementaciones alternativas. Por lo tanto, aunque correctamente modularizado, este componente está íntimamente relacionado con el código de más bajo nivel del guest.

Lo mismo sucede con virtio, que del lado del kernel guest implementa drivers de dispositivos, registrándolos en el subsistema de drivers de Entrada/Salida para que el resto del kernel pueda acceder a los servicios de estos dispositivos virtualizados. Es decir, los drivers front-end de virtio están fuertemente atados a la infraestructura de drivers de Linux.

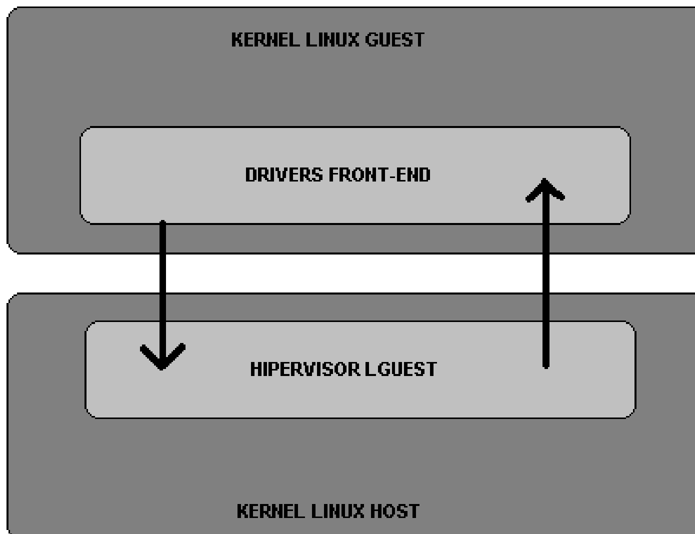
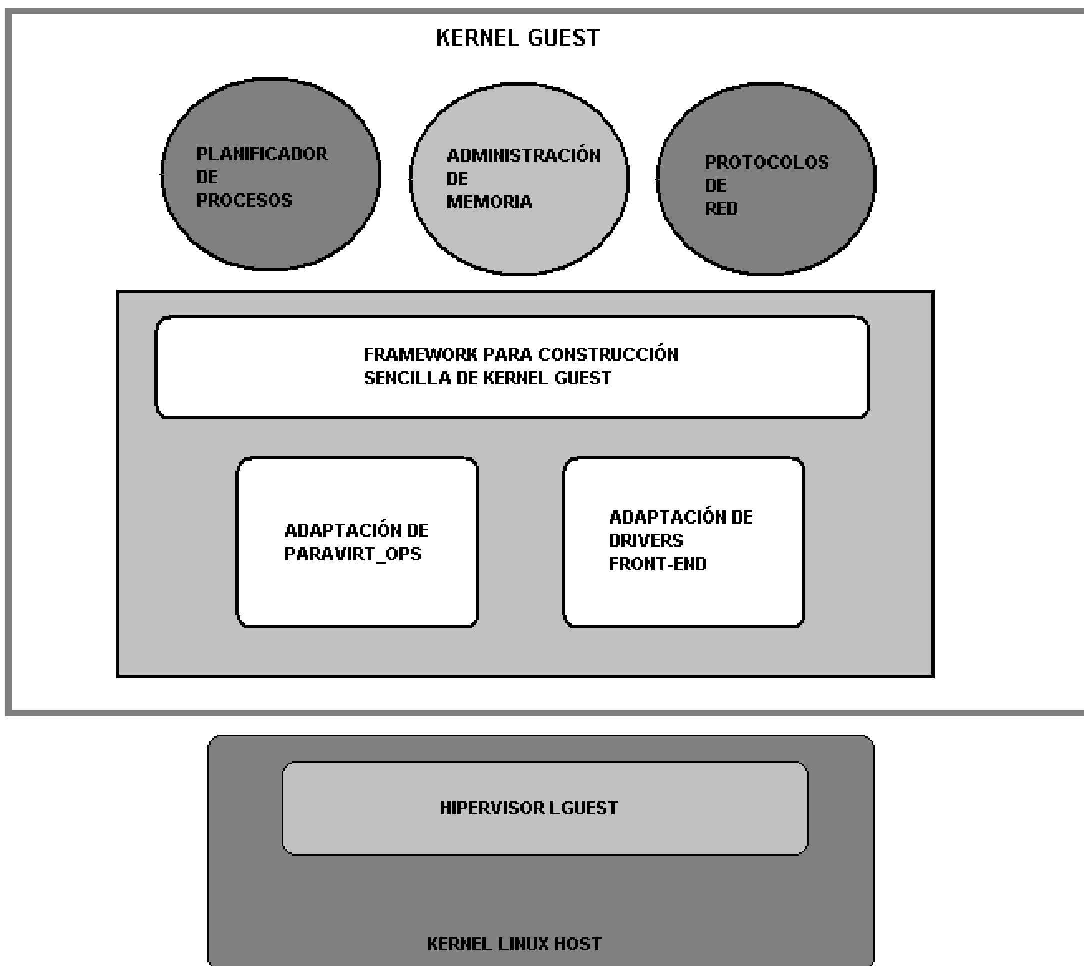


Figura 8.2 – Virtio: paravirtualización de dispositivos

Entonces, aquí se presentó el primer paso en la implementación del framework. Fue necesario adaptar un subconjunto de funcionalidades de paravirt_ops y virtio para poder utilizar sus diseños desde un entorno genérico, externo al kernel Linux. De esta manera se logra invocar a las funcionalidades del hipervisor Lguest, utilizando estas interfaces estándar.

Además, pareció apropiado agregar una abstracción adicional construida sobre las facilidades de virtualización ofrecidas por el subconjunto adaptado de virtio y paravirt_ops. Es decir, mientras estos mecanismos fueron implementados para ser usados sólo por guest con un kernel Linux, la construcción de una nueva capa de software sobre ellos permite construir nuevos guests no basados en Linux pero que accedan a las facilidades de virtio y paravirt_ops. La siguiente imagen puede clarificar esta idea:



Implementación

La implementación final fue concluida en colaboración con Emilio Moretti, alumno avanzado de la Universidad Nacional del Noroeste de la Provincia de Buenos Aires, y se detalla a continuación.

Aunque el pequeño sistema operativo está lejos de considerarse como usable, se ha implementado un intérprete de línea de comando para propósitos de pruebas. Básicamente permite apagar el sistema operativo, verificar las interrupciones y generar fallos de página explícitamente.

Los comandos básicos son los siguientes:

- shutdown: apaga el sistema operativo, terminando la máquina virtual
- ss: imprime el contenido del registro ss en la pantalla
- cs: imprime el registro cs en la pantalla
- irq: configura un timer y le indica a Lguest que genere una interrupción después de un tiempo definido. Es usado para testear los manejadores de interrupciones y el timer de Lguest.
- fallo: este comando genera un fallo de página y detiene al sistema operativo.
- test: imprime un mensaje de prueba en la pantalla.

El intérprete y sus comandos han sido agregados tan sólo con propósitos de prueba, por lo que se consideran temporales y serán removidos en las futuras revisiones del software, ya que no se consideran funcionalidades centrales del proyecto.

El prototipo implementa la capa de más bajo nivel que se puede construir sobre las abstracciones ofrecidas por paravirt_ops y virtio. Pretende ofrecer una plataforma mínima sobre la cual construir ideas de más alto nivel de manera sencilla. Se ejecuta en el ring 1 e interactúa con Lguest para realizar operaciones que requieran un mayor nivel de privilegios. En otras palabras, es un kernel paravirtualizado.

Para comprender el proceso básico de inicio, es útil analizar el archivo init.c. Los primeros pasos se relacionan con el acceso al boot header construido por el launcher de Lguest, en el cual se almacena, entre otros datos, la cantidad de memoria del sistema. Luego es necesario notificar a Lguest sobre la existencia de este nuevo guest y ésto se implementa a través de una hypercall especial que permite el intercambio de información con el VMM. Por ejemplo, se indica la ubicación de la memoria compartida o las tablas de páginas iniciales.

El prototipo es compatible con la especificación *Multiboot*, pero esto no debe hacer pensar que se ofrece soporte para hardware nativo. El código para configurar el GDT, IDT, la paginación y el soporte de consola básico aún no ha sido escrito.

Tablas de Descriptores

La configuración de la Global Descriptor Table (GDT) y de la Interrupt Descriptor Table (IDT) se llevan a cabo de manera distinta a como se hace en un entorno nativo de hardware x86. En un procesador real simplemente se escribirían en memoria y luego se le indicaría al procesador de la existencia de estas tablas a través de simples instrucciones.

En el entorno paravirtualizado que ofrece Lguest esta operación es distinta. Para configurar la IDT y la GDT es necesario iterar sobre cada una de estas tablas y notificar a Lguest sobre cada entrada de la tabla.

descriptor_table.c es el archivo encargado de configurar la GDT y para esto declara cinco entradas que básicamente establecen un segmento nulo, un segmento de código, un segmento de datos, un segmento de código de usuario y un segmento de datos de usuario.

```
static gdt_entry_t gdt_entries[5] =
{GDT_ENTRY_INIT(0x0,0x0,0x0),GDT_ENTRY_INIT(0xC0BA,0x0,0xFFFFFFFF),GDT
_ENTRY_INIT(0xC0B2,0x0,0xFFFFFFFF),GDT_ENTRY_INIT(0xC0FA,0x0,0xFFFFFFF
F),GDT_ENTRY_INIT(0xC0F2,0x0,0xFFFFFFFF)};
```

Es importante notar que el valor de los flags 0xC0BA configurarán el DPL (Descriptor Privilege Level) en 1 porque los guests se ejecutan en el nivel de privilegios 1. Luego 0xC0FA establece el DPL a 3, por que ése es el nivel de privilegios del usuario. Luego será necesario notificar al guest acerca de la nueva GDT, pero en lugar de invocar la función nativa *lgdt,* tenemos que usar una hypercall por cada entrada de la GDT:

```
for (i = 0; i < (unsigned int) (desc->size+1)/8; i++)

      hcall(LHCALL_LOAD_GDT_ENTRY, i, gdt[i].a, gdt[i].b, 0);
```

La IDT consiste en un arreglo de estructuras las cuales apuntan a las funciones que actúan como manejadores de interrupciones. El archivo isr.c realiza exactamente los mismos pasos requeridos para cargar la GDT: itera sobre una IDT declarada y notifica a Lguest sobre cada entrada.

Manejadores de interrupciones

La implementación básica propuesta es flexible y fácil de comprender. Consiste en los siguientes componentes:

- Una función que actúa de manejador global que es invocado por todos los manejadores de interrupciones definidos en la IDT: **isr_handler(register_t regs)**
- Un arreglo interno que almacena los punteros a funciones y los números de interrupciones
- Una función que asocia una función recibida como parámetro con una interrupción: **register_interrupt_handler(u8 n, isr_t handler)**

Por lo tanto resulta trivial explicar su funcionamiento:

1. Se invoca a la función **interrupt_handler**(u8 n, isr_t handler) para asociar una función a un número de interrupción. Básicamente se realiza la siguiente asignación: interrupt_handlers[n] = handler
2. Cuando el kernel recibe una interrupción, siempre llamará a **isr_handler(register_t regs)** a menos que la interrupción sea mayor o igual a 32 y menor o igual a 47, en cuyo caso se invocará a irq_handler, explicado más adelante.
3. **isr_handler(register_t regs)** buscará en el arreglo interrupt_handlers para verificar que existe una función manejadora definida para ese número de interrupción.
4. Si encuentra 0x0, no hace nada. Si es distinto de 0x0, entonces existe un manejador válido y se invoca.

Éste es el manejo básico de interrupciones. Para crear una función manejadora de interrupciones, simplemente debe registrarse y el kernel hará el resto del trabajo.

IRQs

La manera en la que manejamos las IRQs se simplifica enormemente por el propio framework de interrupciones del kernel y básicamente es el mismo algoritmo recién explicado. En Lguest no es necesario re-mapear las IRQs porque no colisionan con las excepciones del procesador. Por lo tanto, comparado con el algoritmo general de interrupciones, hay un único cambio: en lugar de usar isr_handler, las IRQs llaman a **irq_handler**(registers_t regs).

En el archivo isr_asm.S se crean los manejadores de interrupciones reales, los cuales adaptan cada interrupción a una forma más general, para que puedan ser administrados con la misma función, sin importar si retornan un código de error o no.

Por otro lado, el código que registra esta función reside en init_idt() (isr.c). Básicamente registra cada función manejadora con su interrupción correspondiente para que sea posible aplicar el algoritmo simplificado.

Timer Programable o PIT (Programmable Interval Timer)

No hay tal cosa como un contador programable periódico en este contexto. Lguest ofrece un PIT de un único evento, lo que significa que sólo interrumpe una vez y luego deja de medir el tiempo. El código actual sólo ofrece una prueba de concepto, que puede ser invocada desde la línea de comandos. Si se tipea el comando irq, se invocará al siguiente código:

```
/* registra un manejador para la interrupción del PIT */

register_interrupt_handler(32,test_irq0);



/* habilita la interrupción */

enable_lguest_irq(0);



/* define el intervalo de tiempo que debe pasar antes de que nos
despierte en el futuro */

unsigned long delta =100UL;

hcall(LHCALL_SET_CLOCKEVENT, delta, 0, 0, 0);
```

El código es sencillo. Con estos mecanismos será trivial implementar un PIT periódico.

Paginación

Éste es otro de los puntos que difieren del hardware real. En Lguest no debemos modificar cr0 porque la paginación ya se encuentra activada cuando nuestro guest comienza a ejecutarse. Además, los guest no tienen forma de acceder al cr0 real.

Aunque no exista una opción para escribir el cr0, el guest tiene acceso al cr2 y cr3, lo que significa que es posible obtener la dirección que generó el fallo para implementar el algoritmo de paginación apropiado. Los registros shadow cr2 y cr3 se almacenan en la estructura lguest_data en memoria compartida, por lo que leer de ellos simplemente significa acceder a los miembros de esa estructura:

```
unsigned long lguest_read_cr2(void)

{

      return lguest_data.cr2;

}



unsigned long lguest_read_cr3(void)

{

      return lguest_data.pgdir;

}
```

Escribir en el registro shadow cr3 implica una hypercall:

```
void lguest_set_pgd(unsigned long *pgdp, unsigned long pgdval)

{

      *pgdp = pgdval;

      hcall(LHCALL_SET_PGD, __pa(pgdp) & PAGE_MASK,(__pa(pgdp) &
(PAGE_SIZE - 1)) / sizeof(unsigned long),0,0);

      lguest_flush_tlb_kernel();

}
```

El resto del código es relativamente sencillo de comprender y se recomienda su lectura para una estudio detallado de la implementación de nuestro diseño.

Bibliografía

[TEYET] ***Desarrollo de un framework para la enseñanza de los Sistemas Operativos usando Lguest.*** Lic Javier Díaz, Mg. Lía Molinari, Lic. Matías Zabaljauregui
LINTI, Laboratorio de Investigación en Nuevas Tecnologías Informáticas
Universidad Nacional de La Plata. La Plata, Argentina. Aceptado para exposición en el **V Congreso de Tecnología en Educación y Educación en Tecnología** que se realizará en la Ciudad de El Calafate, provincia de Santa Cruz, los días 6 y 7 de Mayo de 2010.

[ZABAL] Performance y Escalabilidad del Kernel Linux aplicado a Redes de Alta Velocidad. Matías Zabaljáuregui. Trabajo final para obtener el grado de
Licenciado en Informática de la Facultad de Informática,
Universidad Nacional de La Plata, Argentina. Director: Lic. Javier Díaz. Co-director: Ing. Luis Marrone. La Plata, Abril de 2007.

[GITHUB] https://github.com/

[OSOM] https://github.com/dc740/OSom

9. Conclusiones y trabajo a futuro

En este capítulo se presentan las conclusiones finales del trabajo realizado en el período de investigación que culminó con la realización de esta tesis. Principalmente la verificación de que es posible la implementación de un kernel construido como un módulo guest de una capa de paravirtualización. Además se mencionan las posibles líneas futuras de trabajo que quedan abiertas para que otros investigadores puedan continuar con el desarrollo de las ideas expuestas.

Como se mencionó previamente, el potencial de la nube como transformador de la tecnología está acelerando la adopción de estas técnicas. Se calcula que para fines del 2012, el 76% de las compañías seguirá una estrategia de nube privada.

La marcada tendencia hacia la adopción de tecnologías de virtualización que se planteaba como una novedad hace algunos años se ha vuelto una realidad de la infraestructura computacional demostrando sus ventajas en flexibilidad, costos, consumo de energía y seguridad.

Hemos visto una notable evolución en las arquitecturas de hardware ofrecidas por los principales desarrolladores donde el soporte para la virtualización de la CPU, de la memoria y de los dispositivos de Entrada/Salida ha sido uno de los cambios más importantes en los últimos años de la historia reciente del hardware.

Los sistemas operativos, a su vez, han adaptado antiguas técnicas de virtualización y emulación a los nuevos contextos y se han realizado considerables esfuerzos de estandarización, principalmente en las implementaciones basadas en el kernel Linux.

En esta línea, este trabajo ofreció una propuesta de diseño para un kernel que se construye desde sus cimientos sobre las facilidades que ofrecen las técnicas de virtualización. El autor considera que es necesario un replanteo profundo del diseño de los sistemas operativos para finalmente incorporar completamente a la virtualización en la pila de la infraestructura de cómputo.

El prototipo presentado en el capítulo 8 demuestra que la propuesta es viable y que muchos de las dificultades usuales a la hora de implementar software de bajo nivel son atenuadas si contamos con una delgada capa de paravirtualización, al estilo microkernel, que ofrezca una interfaz eficiente y de alto nivel a los sistemas operativos guests, en lugar de intentar emular con precisión el hardware subyacente.

También se ha demostrado una vez más la gran potencia del modelo de desarrollo basado en software libre gracias al cual tanto los ingenieros de las grandes empresas como los investigadores de la academia pueden proponer, compartir y comparar las nuevas ideas en un entorno de desarrollo real como el del kernel Linux. A su vez, este modelo facilitó la implementación del prototipo presentado en esta tesis, ya que permitió la colaboración a distancia de un alumno de grado avanzado y diversos comentarios de colaboradores externos.

Las posibilidades que se abren desde aquí para futuros trabajos de investigación, desarrollo y enseñanza de los sistemas operativos se enumeran a continuación como una muestra de posibles trabajos a futuro.

En primer lugar, el desarrollo completo de un sistema operativo completo construido sobre nuestro prototipo sería una buena demostración de las aplicaciones de este trabajo. Teniendo en cuenta que no serán necesarios esfuerzos de implementación de las capas de más bajo nivel, los desarrolladores pueden poner su atención en mejores algoritmos y estructuras de datos para los módulos de alto nivel, como el planificador de CPU, el administrador de la memoria o los protocolos de red.

Al facilitarse la implementación de sistemas operativos, podemos plantear diferentes soluciones para diferentes escenarios. La idea de sistemas operativos especializados para diversas tareas

ha sido mencionada por el autor en trabajos anteriores (linux.linti.unlp.edu.ar). Ésta es una forma práctica de llevar a cabo la idea.

Además, la posibilidad de diseñar e implementar los algoritmos de un sistema operativo en un entorno virtualizado se presenta como una herramienta ideal para la enseñanza y la investigación en el tema. Hemos notado que los alumnos se ven más motivados cuando trabajamos con implementaciones reales al explicar los conceptos de la materia.

Finalmente, el tema de estudio de las técnicas de virtualización puede ser abordado para su aprendizaje, prototipación y medición a través de nuestro prototipo, ya que el código fuente y los comentarios del mismo se encuentran disponibles para todo el interesado en el tema. De esta forma, nuestro trabajo se presenta como un punto de partida para explorar nuevas ideas para la virtualización de recursos.

Apéndice A
Interrupciones y Excepciones en x86

Una interrupción es usualmente definida como un evento que altera la secuencia de instrucciones ejecutada por un procesador. Tales eventos se corresponden con señales eléctricas generadas por circuitos tanto dentro como fuera del chip de la CPU.

Las interrupciones se dividen frecuentemente en interrupciones síncronas e interrupciones asíncronas.

Las interrupciones síncronas son producidas por la unidad de control de la CPU mientras ejecuta instrucciones y son llamadas síncronas porque la unidad de control las genera sólo después de terminar la ejecución de una instrucción.

Las interrupciones asíncronas son generadas por otros dispositivos de hardware en momentos arbitrarios con respecto a las señales de reloj de la CPU.

Los manuales de los microprocesadores de Intel designan a las interrupciones síncronas y asíncronas como excepciones e interrupciones, respectivamente. En este trabajo se adopta esta clasificación, aunque se acepta el término genérico "señal de interrupción" para designar ambos tipos en general.

Las interrupciones son generadas por los timers y dispositivos de entrada/salida; por ejemplo, la llegada de un paquete de red a la interfaz de red de la plataforma generará una interrupción.

Las excepciones, por otro lado, son causadas por errores de programación o condiciones anómalas que deben ser manejadas por el kernel. En el primer caso, el kernel maneja la excepción enviando al proceso actual una de las señales especificadas en los sistemas operativos UNIX. En el segundo caso, el kernel realiza todos los pasos necesarios para recuperarse de la condición anómala, tal como un fallo de página o un requerimiento, vía una instrucción en lenguaje assembly como INT o SYSENTER, de un servicio del kernel.

A continuación se clasifican las interrupciones clásicas, siguiendo la terminología utilizada en los manuales Intel:

Interrupciones

- Interrupciones enmascarables: Todos los requerimientos de interrupción (IRQs, por Interrupt Request) invocados por dispositivos de Entrada/Salida generan interrupciones enmascarables. Una interrupción de este tipo puede estar en dos estados: enmascarada o no enmascarada. Una interrupción enmascarada es ignorada por la unidad de control por el tiempo que permanezca en ese estado.

- Interrupción no enmascarables: Sólo unos pocos eventos críticos (como fallos del hardware) pueden generar interrupciones no enmascarables. Éstas siempre serán recibidas y atendidas por la CPU.

Excepciones

Podemos clasificar las excepciones en dos conjuntos: aquellas detectadas por el procesador y las que son programadas.

Las excepciones detectadas por el procesador son generadas cuando la CPU detecta una condición anómala mientras ejecuta una instrucción. Éstas se clasifican a su vez en tres

grupos, dependiendo del valor del registro EIP que es guardado en la pila en modo kernel cuando la unidad de control de la CPU eleva la excepción.

- Fallos: Pueden generalmente ser corregidos; una vez corregidos, se le permite al programa seguir con su ejecución sin perder su continuidad. El valor guardado del registro EIP es la dirección de la instrucción que causó el fallo, y por lo tanto esa instrucción puede ser retomada cuando el manejador de excepciones termina.

- Traps: Se reportan inmediatamente al terminar la ejecución de una instrucción que genera la excepción; cuando el kernel retorna el control al programa, se le permite seguir con su ejecución sin perder continuidad. El valor guardado del registro EIP es la dirección de la instrucción que debería ser ejecutada inmediatamente después de la instrucción que causó esta excepción. Sólo se dispara un trap cuando no hay necesidad de re-ejecutar la instrucción que lo generó. El uso principal de los traps se relaciona con propósitos de depuramiento. El rol de la señal de interrupción en este caso es el de notificar al depurador que una instrucción específica ha sido ejecutada (por ejemplo, un breakpoint ha sido alcanzado en un programa). Una vez que el usuario ha examinado los datos provistos por el entorno, puede retomar la ejecución del programa, comenzando con la próxima instrucción.

- Abortos: Suceden ante la ocurrencia de un error serio; la unidad de control podría ser incapaz de almacenar el valor del registro EIP con la localización precisa de la instrucción que causó la excepción. Los abortos se usan para reportar errores severos, tales como fallos de hardware o valores inválidos o inconsistentes en las tablas del sistema. La señal enviada por la unidad de control es una señal de emergencia usada para ceder el control al manejador de excepción de aborto correspondiente. Este manejador no tiene más opciones que forzar al proceso afectado a terminar.

Las excepciones programadas ocurren bajo requerimiento del programador. Son disparadas por las instrucciones INT o INT3; las instrucciones INTO (verifica si hubo overflow) y BOUND (verifica límites de una dirección) también elevan una excepción programada cuando la condición que están verificando no es verdadera. Las excepciones programadas son manejadas por la unidad de control como si fueran traps; usualmente son llamadas *interrupciones por software.* Tales excepciones tienen dos usos comunes: la implementación tradicional de system calls y la notificación a un depurador sobre un evento específico.

Cada interrupción o excepción se identifica por un número que va desde el 0 al 255; Intel llama vector a este número sin signo de 8 bits. El vector de las interrupciones no enmascarables y excepciones está especificado en los manuales, mientras que aquellos de las interrupciones enmascarables pueden ser alterados programando el controlador de interrupciones.

Los procesadores x86 generan aproximadamente 20 excepciones diferentes (el número exacto depende del modelo del procesador). Para algunas excepciones la unidad de control de la CPU también genera un *código de error de hardware* y lo coloca en la pila en modo kernel antes de que comience el manejador de excepción.

La siguiente lista muestra el vector, el nombre el tipo y una breve descripción de las excepciones encontradas en los procesadores x86.

0. “División por cero” (fallo): se dispara cuando un programa realiza una división entera por cero.
1. “Debug” (trap o fallo): Sucede cuando el flag TF del registro EFLAGS está activado (muy útil para implementar ejecución paso a paso).
2. “Sin usar”: Reservada para las interrupciones no enmascarables.
3. “Breakpoint” (trap): Causada por una instrucción INT3 (breakpoint), usualmente insertada por un depurador.
4. “Overflow” (trap): Una instrucción INTO (verificar por overflow) ha sido ejecutada mientras el flag OF del registro EFLAGS está activado.

5. “Verificación de límites” (fallo): Una instrucción BOUND (verifica el límite de una dirección) es ejecutado con un operando fuera de los límites de direcciones válidos.
6. “Código de Operación Inválido” (fallo): La unidad de ejecución de la CPU ha detectado un *opcode* inválido.
7. “Dispositivo no disponible” (fallo): Un instrucción de ESCAPE, MMX o SSE/SSE2 ha sido ejecutada con el flag TS del registro CR0 activado.
8. “Doble fallo” (aborto): Normalmente, cuando la CPU detecta una excepción mientras intenta invocar a un manejador por una excepción previa, el procesador no puede manejarlas en serie, por lo que debe generar esta excepción.
9. “Segmento del coprocesador” (aborto): Problemas con el coprocesador matemático externo. Se aplica sólo a los viejos microprocesadores 80386.
10. “TSS inválido” (fallo): El CPU ha intentado un cambio de contexto a un proceso que tiene un *Task State Segment* inválido.
11. “Segmento no presente” (fallo): Se hizo una referencia a un segmento que no está presente en memoria.
12. “Fallo de segmento de pila” (fallo): La instrucción intentó exceder el límite del segmento de pila, o el segmento identificado por el registro SS no está presente en memoria.
13. “Protección general” (fallo): Una de las reglas de protección del modo protegido de x86 ha sido violada.
14. “Fallo de página” (fallo): La página accedida no está presente en memoria, la entrada correspondiente en la tabla de páginas es nula, o ha ocurrido una violación del mecanismo de protección de paginación.
15. Reservado por Intel.
16. “Error de punto flotante” (fallo)
17. La unidad de punto flotante integrada en el chip de la CPU ha señalizado una condición de error, tal como un desbordamiento numérico o una división por cero.
18. “Verificación de máquina” (aborto): Un mecanismo de verificación de máquina ha detectado un error de CPU o del bus.
19. “Excepción de punto flotante SIMD” (fallo): La unidad SSE o SSE2 integrada en el chip de la CPU ha señalizado una condición de error en una operación de punto flotante.

Los valores desde el 20 al 31 están reservados por Intel para desarrollos futuros. Y aquellos valores desde el 32 al 255 están disponibles para las interrupciones enmascarables.