The Architecture of Virtual Machines

CARLOS MARTÍN FLORES GONZÁLEZ, Carné: 2015183528

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

A pesar de su increíble complejidad, los sistemas computacionales existen y continúan evolucionando porque están diseñados como jerarquías de interfaces bien definidas que separan niveles de abstracción. Usando interfaces bien definidas se facilita el desarrollo de subsistemas independientes en grupos de software y hardware. Las abstracciones simplificadas esconden detalles de implementación de bajo nivel y, por lo tanto, reduce la complejidad del diseño de procesos. La arquitectura del conjunto de instrucciones de una computadora (*instruction set architecture* - ISA) ejemplifica las ventajas de interfaces bien definidas al permitirle a los desarrolladores interacturas con subsistemas de la computadora no solamente entre organizaciones sino también en diferentes momentos en el tiempo. Por ejemplo, los diseñadores de Intel y AMD desarrollan microprocesadores que implementan el conjunto de instrucciones de Intel IA-32(x86), mientras que los desarrolladores de Microsoft escriben software que se compila en el mismo conjunto de instrucciones. Dado que ambos equipos satisfacen la especificación ISA, se puede esperar que el software ejecute correctamente en cualquier PC construida con un microprocesador IA-32.

Virtual Machines (VMs). El concepto de virtualización puede ser aplicado no solo a subsistemas tales como el sistema de disco sino también a la máquina entera. Para implementar una máquina virtual, los desarrolladores agregan una capa de software a una máquina real para soportar una arquitectura deseada. Al hacer esto, una VM puede eludir las restricciones de compabilitidad con la máquina real y los recursos de hardware.

Architected Interfaces. Debido a que las implementaciones de VMs dependen de interfaces de arquitectura, una consideración importante a la hora de construir una VM is la fidelidad con la que se implementan estas interfaces. *Arquitectura:* se refiere a la especificación formal de una interface en un sistema, incluyendo el comportamiento lógico de recursos gestionados a través de esa inteface. *Implementación:* describe la realización de una arquitectura. Los niveles de abstracción corresponden a capas de la implementación, ya sea en hardware o software, cada una asociada con su propia interface o arquitectura. *Instruction set architecture:* La ISA marca la división entre el hardware y el software, y consiste de: (1) *User ISA:* incluye aquellos aspectos visibles a un programa de aplicación.(2) *System ISA:* un superconjunto de el *user ISA* e incluye aquellos aspectos visibles solo al software del sistema operativo(SO) responsable for gestionar los recursos de hardware. *Application binary interface (ABI):* le provee a un programa el acceso a los recursos de hardware y a los servicios disponibles a través del *user ISA* y la interface de llamados al sistema. La ABI no incluye instrucciones del sistema, en su lugar todos los programas de aplicación interactúan con los recursos de hardware indirectamente al invocar los servicios del SO a través de la interfaz de llamados al sistema. *Applicacion programming interface (API):* le provee a un programa accesso a los recursos de hardware y a los servicios disponibles a través del sistema a través del *user ISA* complementado con llamadas de lenguaje de alto nivel (HLL - *high-level language library calls*). Cualquier llamada al sistema de

realiza usualmente a través de librerías. El usar un API permite a un software de aplicación ser portado fácilmente, a través de recompilación, a otros sistemas que soportan la misma API.

Process and system VMs. Concepto de máquina: (1) Desde la perspectiva de un proceso ejecutando un programa de usuario: la máquina consiste de un espacio de direcciones de memoria lógica asignado al proceso junto con instrucciones de nivel de usuario y registros que permiten la ejecución del código que le pertenece a ese proceso. La entrada/salida(I/O) de la máquina es visible solamente a través del SO, y la única forma en la que el proceso puede interactuar con el I/O del sistema es por medio de llamados al sistema del SO. De esta forma la ABI define una máquina vista como un proceso. De forma similar, el API específica las características de la máquina a través de una aplicación de programa HLL. (2) Desde la perspectiva del SO y las aplicaciones que soporta: la máquina entera corre en la máquina subyacente. Un sistema es un ambiente completo de ejecución que puede soportar numerosos procesos simultaneamente. Esos procesos comparten un sistema de archivos y otros recursos de I/O. El sistema asigna memoria real y recursos de I/O a los procesos y permite a los procesos interactura entre sí. Desde el punto de vista del sistema, el hardware subyacente define la máquina. Es el ISA el que provee la interface entre el sistema y la máquina. *Process VM:* una plataforma virtual que se ejecuta como un proceso individual. Este tipo de VM existe solamente para soportar el proceso. Se crea cuando el proceso se crea y termina cuando el mismo termina. *System VM:* provee un ambiente completo y persistente qeu soporta un SO junto con sus procesos de usuario. Como SO invitado(*guest*) provee acceso a recursos de hardware virtual, incluyendo red, I/O y quizás interfaz gráfica junto con procesador y memoria. El procesos o sistema que corre en una VM es el *guest*, mientras que la plataforma subyacente que soporta la VM es el anfitrión(*host*). El software de virtualización que implementa un *process VM* se le llama "*runtime software*". El software de virtualización en un *system VM* se le llama típicamente *virtual machine monitor* (VMM) En un *process VM* el software de virtualización está a nivel del ABI o API, encima de SO y el hardware. Se emulan instrucciones de nivel de usuario y llamadas al SO. En un *system VM* el software de virtualización está entre el hardware de la máquina del *host* y el software del *guest*. La VMM emula el ISA del hardware de esta forma el software del *guest* puede potencialmente ejecutar un ISA diferente al de la implementado por el *host*.

Process Virtual Machines. *Multiprogrammed systems:* la mayoría de SO puede soportar varios procesos de usuario simultaneamente a través de multiprogramación lo que le da a cada proceso la ilusión de tener una máquina completa para él. *Emulators and dynamic binary translators:* dar soporte a programas compilados en un conjunto de instrucciones diferente al que ejecuta el *host*. (1) Emulación por interpretación: un programa interpretador que recupera, decodifica y emula la ejecución de instrucciones individuales del *guest*. Puede llegar a ser lento. (2) Emulación por traducción dinámica binaria: convierte instrucciones del *guest* a instrucciones para el *host* en bloques en lugar de una por una y las guarda para reutilizarlas en en caché. *Same-ISA binary optimizers:* para prevenir pérdida de rendimiento, los traductores dinámicos binarios a veces realizan optimizaciones durante la traducción. Esta característica tiene mayor aprovechamiento en VMs en las cuales el conjunto de intrucciones del *guest* es el mismo que el del *host*. *High-level-language VMs:* (1) *Conventional platform specific compilation environment:* un *compiler front end* genera primero código intermedio que es similar al código de la máquina pero más abstracto. Luego, un generador de código usa el código intermedio para generar un binario que contiene código máquina para la ISA y el SO especificos. Este archivo binario es distribuido y ejecutado en plataformas que soportan una combinación dada de ISA/SO. *HLL VM:* un *compiler front end* genera código máquina abstracto en una ISA virtual que especifica la interfase de la VM. Este código de la ISA virtual, junto con información asociada de estructuras de datos, se distribuye para ser ejecutado en diferentes plataformas. Cada plataforma *host* implementa una VM capaz de cargar y ejecutar la ISA virtual y el conjunto de librerías de rutinas especificadas por una API estandarizada. En su forma más simple, la VM contiene un intérprete. VMs más sofisticados compilan el código máquina abstracto dentro del código máquina del *host* para ejecución directa en la plataforma del *host*.

Una ventaja de HLL VM es que el código de aplicación puede ser fácilmente portado una vez que la VM y las librerías son implementadas en una plataforma *host*.

System Virtual Machines. Un *system VM* provee un ambiente completo en donde un SO y muchos procesos, posiblemente pertenecientes a múltiples usuarios, pueden coexistir. Al usar *system VMs*, una plataforma de hardware *single-host* puede soportar múltiples sistemas operativos *guest* aislados simultaneamente. Quizás la aplicación actual más importante de la tecnología de *system VM* es el aislamiento que provee entre múltiples sistemas corriendo concurrentemente en la misma plataforma de hardware. Si la seguridad en un *guest* se compromete o si un SO falla, el software corriendo en los otros sistemas *guests* no se afecta.

Classic system VMs: El enfoque clásico pone la VMM en el hardware y la VM por encima. La VMM corre en el modo de privilgio más alto, mientras que todos los sistemas *guests* corren con privilegios reducidos de tal forma que la VMM puede interceptar y emular todas las acciones del SO *guest* que normalmente accederían o manipularían recursos de hardware.

Hosted VMs: la implementación de la VM construye software de virtualización por encima de un SO *host* existente, dando como resultado un *hosted VM*. Una ventaja de un *hosted VM* es que el usuario lo instala como si fuera una programa de aplicación típico. Además, el software de virtualización puede confiar en el SO *host* para proporcionar controladores de dispositivos y otros servicios de bajo nivel en lugar de la VMM.

Whole-system VMs: en algunas ocasiones los sistema *guest* y *host* no tiene un ISA en común. *Whole-system VMs* tratan con la situación en donde se virtualiza todo el software, incluyendo el SO y las aplicaciones. Debido a que los ISAs son diferentes, la VM debe emular ambos el código de la aplicación y el SO. Un ejemplo de este tipo de VM is Virtual PC, en donde un sistema Windows corre en una plataforma Macintosh. El software de VM ejecuta como un programa de aplicación soportado por el SO *host* y no usa operaciones ISA.

Multiprocessor virtualization: se da cuando la plataforma subyacente del *host* es un multiprocesador de memoria compartida grande. Aqui, un objetivo importante es la partición de un sistema grande en múltiples sistemas multiprocesador pequeños al distribuir los recursos subyacentes de hardware del sistema grande. Con *physical partitioning*, los recursos físicos que un sistema virtual usa están disjuntos de aquellos usados por otros sistemas virtuales. *Physical partitioning* provee un alto grado de aislamiento, de tal forma ninguna falla de hardware o software en una partición afecta los programas en otras particiones. Con

logical partitioning, los recursos de hardware subyacentes estan multiplexados por tiempo entre diferentes particiones, de este modo se mejora la utilización de los recursos del sistema. Sin embargo, algunos de los beneficios del aislamiento de hardware se pierden.

Codesigned VMs: implementan ISAs nuevas y propietarias dirigidas al mejoramiento del rendimiento, eficiencia energética o ambos. El ISA del *host* puede ser completamente nuevo, o puede ser una extensión de un ISA existente. Una *codesigned VM* no tiene una aplicaciones ISA nativas. En su lugar, la VMM parece ser de la implementación de hardware. Su único propósito es el de emular el ISA del *guest*. Para mantener esta ilusión, la VMM reside en una región de memoria reservada de todo el software convencional. Incluye un traductor binario que converte instrucciones *guest* en secuencias optimizadas de instrucciones ISA de *host* y las pone en el caché de la región reservada de memoria.

Virtual Machine Taxonomy

- **Process VMs**
  - Same ISA
    - Multiprogrammed systems
      Same-ISA dynamic binary optimizers
  - Different ISA
    - Dynamic translators
      High-level-language VMs
- **System VMs**
  - Same ISA
    - Classic System VMs
      Hosted VMs
  - Different ISA
    - Whole-system VMs
      Codesigned VMs

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

Los sistemas computacionales existen y continuan evolucionando porque son diseñados como jerarquías con interfaces bien definidas que separan niveles de abstracción. Al utilizar estas interfaces bien definidas se facilita el desarrollo independiente de subsistemas por equipos de hardware y software.

Desafortunadamente, las interfaces bien definidas tambien tienen sus limitaciones. Subsistemas y componentes diseñados para una las especificaciones de una interfaz no van a trabajar con aquellos diseñados para otra. Por ejemplo, los programas de aplicación cuando son distribuidos como binarios compilados se atan a una ISA en específico y depende de una interface de sistema operativo específica. Esta falta de interoperatibilidad puede convertirse en algo muy restrictivo, especialmente en un mundo de computadoras en red donde es muy ventajoso mover el software tan libremente como los datos.

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

La virtualización se ha convertido en una herramienta importante en el diseño de sistemas de computación y, las máquinas virtuales son usadas en un gran número de subdisciplinas que van desde los sistemas operativos, lenguajes de programación y arquitectura de procesadores. Al liberar a los programadores y usuarios de interfaces trandicionales y restricciones de recursos, las máquinas virtuales mejoran la interoperabilidad del software, impregnabilidad del sistema y versatibilidad de plataforma.

Debido a que las máquinas virtuales son el producto de grupos diversos con objetivos diferentes, existe relativamente poca unificación de conceptos. Por esta razón es útil hacer un alto y considerar la variedad de arquitecturas de máquinas virtuales y describirlas en una forma unificada, poniendo tanto la noción de virtualización y los tipos de máquinas virtuales en perspectiva.

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

Ambas *system* y *process* VMs datan de los años 60s. De acuerdo con Wikipedia[1] las *system VMs* crecieron a partir del tiempo compartido el cual permitía que múltiples usuarios usaran la computadora de forma concurrente: cada programa parecía tener acceso total a la máquina pero sólo un programa se ejecutaba a la vez. Esto evolucionó en máquinas virtuales: IBM M44/44X, CP-40 SIMMON. La primera arquitectura de VM disponible fue la CP-67/CMS. Las *process VMs* surgen originalmente como plataformas abstractas para lenguaje intermedio usado como una interpretación intermedia de un programa por el compilador. Los primeros ejemplos datan alreadedor de 1966.

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

La virtualización provee una forma para proveer interoperabilidad y brindar solución a las restricciones de recursos e interfaces. Al virtualizar un sistema o un componente -como un procesador, memoria o un dispositivo de I/O- en un nivel de abstracción dado se mapea su interface y sus recursos visibles sobre la interface y los recursosde de un sistema real subyacente (posiblemente diferente). Consecuentemente, el sistema real se parece a un sistema virtual differente o inclusive a múltiples sistemas virtuales.

5. ¿QUÉ TAN EXITOSA ES ESTA SOLUCIÓN?

Las VMs son ahora ampliamente utilizadas para habilitar interoperatibilidad entre hardware, software de sistema y aplicación. Dado la gran dependencia en estándares y la consolidación que ocurre en la industria, es muy probable que cualquier ISA nuevo, SO o lenguaje de programación se llegue a basar en tecnología de VM. En el futuro, VMs deberían ser vistas como una disciplina unificada de la misma forma en la que el hardware, los SO y los lenguajes de programación son vistos hoy en día.

[1] https://en.wikipedia.org/wiki/Virtual_machine