System Virtualization

Lecture notes (Rutgers CS553 Internet Services)
Srinivas Narayana

April 2025

1 Introduction

System virtualization is the ability to run entire operating systems and their process workloads in a virtualized fashion: another entity, the virtual machine monitor (VMM, also called hypervisor) mediates access to all the system resources. Effective support for system virtualization on server hardware platforms ushered in the era of server workload consolidation and the public cloud. System virtualization on desktop/laptop platforms provide the ability to run legacy software or OS-specific software on a different or updated operating system.

2 Recap: The Popek/Goldberg Conditions

In the last lecture, we looked at the Popek/Goldberg sufficient conditions [1] that determine when a hardware architecture (i.e., instruction set) is fully virtualizable. When the sensitive instructions are a subset of the privileged instructions, a hardware architecture can be fully virtualized through trap-and-emulate virtualization.

Trapping control-sensitive instructions in the VMM allows the VMM to determine whenever a guest operating system or guest process attempts to change its own resources or privileges (processor mode). This provides resource control.

Trapping behavior-sensitive instructions in the hypervisor allows the VMM to provide the illusion that the guest OS is running with the appropriate privilege level and/or memory location, providing *equivalence.*

Innocuous (i.e., not sensitive) instructions execute directly on the processor hardware without interposition by the VMM, permitting efficient virtualization.

3 Hybrid Virtual Machines

The Popek/Goldberg paper includes three theorems, the first of which we discussed extensively in the last lecture. The third theorem discusses the idea of *hybrid virtual machines*, which are less efficient than full virtual machines (in the sense of theorem 1), but still allow resource control and equivalence. Understanding this theorem requires understanding the notion of *user-sensitive instructions*, which we discuss next.

Consider an instruction like `JRST 1` on the PDP-10 architecture. The action of the instruction is to return to the calling context. When called in the supervisor mode of the processor (e.g., from an interrupt processing routine), the instruction returns to the calling context in the user mode (and hence is a control-sensitive instruction). When called in user mode, it can also be used to return from a function call. This latter use of the instruction is innocuous. This instruction is not privileged (i.e., does not trap) when executed in user mode. Hence, a typical trap-and-emulate VMM that runs the guest operating system in the processor's user mode will be unable to differentiate the execution of this instruction in the guest's user process or the guest's supervisor (operating system).

To allow such instructions to be virtualized, a VMM must track which mode the guest is currently executing in, i.e., guest user or guest supervisor. The `JRST` instruction is sensitive when executed as guest supervisor but not as guest user. For the VMM to track the use of this instruction in guest supervisor mode (which is still executing in the user mode of the processor), the only option is to emulate every instruction executing in the guest supervisor mode. This is the central design principle of a *hybrid virtual machine:* since guest supervisor execution of this sensitive instruction cannot be trapped, each instruction in guest supervisor mode must now be emulated by the VMM. This VMM is not as efficient as a VMM on a fully-virtualizable architecture: the latter kind of VMM can directly execute instructions on the processor hardware as guest supervisor. This inefficiency may be acceptable when the virtualized system spends most of its time in guest user mode, which is true for compute-intensive application (process) workloads.

Note that any instruction that is sensitive when executed as guest user must still be trapped for the hybrid VMM to retain efficiency and resource control. This is the class of instructions termed "user-sensitive" in theorem 3. If user-sensitive instructions are privileged, a hybrid VMM can be constructed for that architecture, where the HVM emulates all instructions executing as guest supervisor.

4 Types of Virtualization

The term virtualization can be used to refer to many things. At least three different types exist:

1. *Language virtual machines:* Environments that provide a compute and memory abstraction for programs written in a specific language or bytecode format. JVM is an example of such a language VM. Typical execution of programs within such VMs involve interpretation of instructions, and just-in-time compilation to optimize frequently executed code segments.
2. *Lightweight virtual machines:* Environments that enhance the process/thread abstractions already typically available in operating systems to provide more isolation and features. Examples include process containers (Docker, cgroup/etc), Google's NativeClient (and its successor WebAssembly), and eBPF. The goal of such environments is to provide an environment with direct execution of instructions on hardware, but with additional protections, such as through static analysis or run-time interposition within the operating system.
3. System virtual machines: The topic of this lecture. These VMs run full operating systems and their constituent process workloads. VMMs virtualize the hardware for such VMs.

VMMs can be classified into one of two types: type-1 and type-2. Type-1 hypervisors replace the operating system kernel, while type-2 hypervisors run as a component over an existing host operating system. The VMM implements resource allocation in type-1 hypervisors, while the VMM leverages the operating system to implement resource allocation in type-2 hypervisors.

It is possible to implement virtualization either through *simulation* (i.e., interpret each instruction in the guest OS/process and implementing them in equivalent ways on the native processor architecture), or *direct execution* (aligned with the efficiency goal of Popek/Goldberg).

A few common tricks are used by VMMs for virtualization: (1) *multiplexing* makes a single physical resource appear as multiple virtual resources (e.g., one to each VM). For example, a single CPU core may be time-multiplexed to appear as a slower core each to multiple VMs. The physical memory of the machine may be space-multiplexed to use different parts of the memory for different VMs. (2) *Emulation* makes a physical resource appear as a different or standard one to the VM. For example, for compatibility reasons, many VMMs make the underlying disk appear as if it uses a standard interface (e.g., SCSI) even when the physical disk uses a different interface. (3) *Aggregation* can make many physical resources appear as one.

5 Two Key Challenges for x86 virtualization (prior to the mid 2000s)

x86-32 was not a fully virtualizable architecture according to the Popek/Goldberg conditions. However, since it was a dominant architecture when consolidation of server-side workloads was emerging, it was necessary to address its lack of hardware compatibility.

The first kind of incompatibility arises from the existence of sensitive but unprivileged instructions. Prior work documents 17 such sensitive instructions which are not trapped. Any VMM that must efficiently virtualize x86 must deal with these instructions. One benefit of x86 was that it was possible to efficiently identify/separate guest execution as either guest user or supervisor, by using two native processor privilege levels within (just) the guest, since the x86 supported not just two (like Popek/Goldberg's machine model) but four native processor privilege levels. Hence, on x86, it is possible to use one privilege level corresponding to the guest user (least privileged), one to the guest supervisor (next most privileged), and one to the VMM (most privileged).

The second kind of incompatibility arises in memory virtualization. When we think about virtualizing processes within an operating system, the translation process from virtual to physical addresses works through segmentation and paging, which are natively implemented in the MMU hardware. When the entire operating system is virtualized, there is an additional level of translation, since the memory footprint of the guest operating system may relocated to a region of memory not typically expected by (or exposed to) the guest operating system.

Since there are many addresses involved, let's give them different names. The guest user-level process always generates virtual addresses. The guest operating system translates such virtual addresses into *guest physical addresses*. The VMM must translate the guest physical address into a *host physical address*, which is the "true" address that is sent out on the system's memory bus to read or write into memory.

If the hardware does not provide native support for two memory translations, it is necessary to do one or both memory translations in software. That would be too inefficient, since each memory access (and its corresponding translation) would have to be emulated in software.

What if we could have processor hardware directly apply translations from (guest) virtual to host physical addresses? A software-managed TLB (i.e., where the processor traps into the VMM to populate the TLB upon a miss) could make the task of virtualizing memory easy. The VMM could do the two translations in software the first time a page is accessed (within the TLB miss handler), and then all further translations go through efficiently in hardware. However, x86 uses a hardware-managed TLB: whenever the TLB misses, the processor hardware walks the page table in memory to determine the translation to cache in the TLB. For memory translation to be efficient, we need a page table that directly captures the mapping from (guest) virtual to host physical addresses somewhere in memory. Each VMM system had to employ a particular set of tricks to make such page tables visible to the hardware.

6 VMWare Workstation

To address the first incompatibility described above, VMware workstation used the technique of *dynamic binary translation:* every time a code segment was initialized by the guest, it must be checked for sensitive instructions and turned dynamically into their emulated versions that included checks for resource control.

Such translation could be thought of as an efficient form of simulation. Binary translation can use many efficient techniques, e.g., chaining, to make sequences of code executions called from one another efficient. Direct execution can be used most of the time, with translation only requires for code segments with sensitive instructions. VMware workstation used a procedure to determine which mode (DBT or direct execution) the VMM should run in.

A translation-based approach for compute virtualization requires keeping separate versions of code—the original version from the guest and a version after translation. Such translated regions may be cached in memory for efficient execution of frequently-invoked code regions in the guest. These additional copies of code must further be attributed to the guest's resources. These translated pages were protected from the guest VM through segmentation, an approach often called *segment truncation*: any segment allocated by the guest will be truncated to avoid overlap with this VMM-managed region.

To address efficient memory virtualization, VMware managed copies of the guest page tables, termed *shadow page tables*, which were kept in sync with the guest OS every time the guest updated its own page table (through memory traps upon access). The guest OS's view of the page table uses guest physical addresses, but the shadow page table translates to host physical address, and is the version of the page table that is walked by the processor hardware during TLB misses. Shadowing is done on a page-by-page basis. Keeping page tables in sync, and maintaining two copies of the page tables, can be expensive. However, maintaining shadow copies also provides some benefits such as sharing page tables across VMs (e.g., copy-on-write) or removing page table entries when it is deemed that the VM may not need it.

7 Xen and Paravirtualization

Xen used an alternative approach to work around the inefficiency of full virtualization of unmodified operating systems in VMware workstation. The main philosophy of *paravirtualization* is that it is OK to give up a little bit of *equivalence* (to unmodified guest OSes) to gain a lot of *efficiency.* It was OK to change the operating system to make it aware of and more friendly to virtualization. In fact, sometimes, it was better to expose the physical resources to guest operating systems as it enabled better performance or correctness.

Xen used a few key ideas for efficient CPU virtualization. Xen undefined the 17 sensitive instructions, and removed them from the OS source code for Linux. This action removed the need for dynamic binary translation. Any code segment loaded by the guest could be checked once (for the existence of sensitive instructions) and run efficiently with direct execution ever after. Second, any action by the guest that legitimately involved allocating system resources or changing privilege level could be directly called from the guest operating system into the VMM—a notion termed *hypercalls*, similar to system calls from a process into the operating system.

For memory virtualization, Xen exposed the virtual to host physical page table translation in a read-only fashion to guests. Guest OSes could populate these mappings within the host physical region reserved for the guest, and then have Xen validate and write these guest-computed mappings into the hardware-visible page table through a hypercall. The efficiency of such validation can be improved through batching many updates before the hypercall to write the new page table mappings.

References

[1] Gerald Popek and Robert Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Communications of the ACM*, 1974.