CMPSCI 677 Operating Systems **Spring 2014**

Lecture 5: February 3

Lecturer: Prashant Shenoy *Scribe: Aditya Sundarrajan*

5.1 Virtualization

Virtualization is a technique that extends or replaces an existing interface to mimic the behavior of another system. It takes an interface and uses it to emulate another interface. Virtualization was first designed by IBM in the late 1970s. It enabled IBM customers to run their old code on newer systems, by giving them the option of exposing the old mainframe interface on newer machines. In other words, the virtualization layer is like a portability later that lets software compatible with an old harware archticture to run seamlessly on the new hardware. However, these ported application cannot use the new features provided by the new hardware.

5.1.1 Types of Interfaces

Virtualization can be implemented at different levels of the hardware and software stack, depending on what one is trying to mimic.

- Hardware level virtualization: This is the lowest level of virtualization. Here one hardware is used to mimic another hardware. The interface in this case is the instruction set. Hence one instruction set can emulate another instruction set.
- OS level virtualization: Here one OS level interface is used to mimic another OS level interface
- Application level virtualization: Here one application interface can be used to mimic another application interface. E.g. JVM is an example application level virtualization. It is exporting an abstract java virtual machine interface at the application level.

5.1.2 Types of Virtualization

- Full Emulation: This is used to emulate/simulate an entire machine in software. As a result you can emulate any machine you want on any other machine that have different CPU architectures. E.g. When Mac's were PowerPC based, Microsoft used to have a virtualization layer called VirtualPC that emulated an entire x86 PC on a PowerPC architecture. Inside VirtualPC you could run Windows, or Linux, or any other software that was designed to run on a PC unmodified. The advantage is that any hardware can be emulated as the entire functionality of hardware can be implemented in software. The disadvantage is that the speed is slow. Each assembly instruction will be replaced by one of more instructions in the native environment. Each instruction has to be dynamically translated and executed. So the speed can sometimes be an order of magnitude slower.
- Full/Native Virtualization: In this case, one interface is being used to emulate an underlying interface of the same type. E.g. emulating Intelx86 on Intel x86. This is mainly done to allow one operating system(OS) to run another, or to run one environment of an OS on another environment of the same

OS such that the failure of the guest OS does not affect the host OS. For example, one could run Linux on Windows, or Windows on Linux. The advantage is that a full translation is not needed because the hardware family is the same. But we will still need to emulate the disk, network interfaces, etc. Examples include VMware, VirtualBox.

- Para-virtualization: Here the VM does not simulate hardware but it requires modifications to the OS to run it on the virtual machine. In paravirtualization, the virtualization layer is called the hypervisor, which traps calls from the operating system and executes them. E.g Xen virtual machine monitor.

- OS-Level Virtualization: In this case, the hardware and OS are the same and it lets one OS interface (system call interface) emulate another. For example an application that runs on linux kernel 2.x can continue to run on linux 3.x using OS vitualization, where the old system calls are emulated by the new system calls. It can also be used to provide resource isolation. This provides a notion of a smaller virtual server/resource container which has its own copy of the OS. The process running inside one of these containers sees just an ordinary OS, same as the native OS, but the processes in one container cannot see the processes in another container. It is important to note that an entire copy of the OS is not running in the container. This is only a way to isolate the container, and allocate to it a part of the resources. For instance, the disk can be partitioned such that every container has access only to its own partition. OS level virtualization also lets you partition resources such as CPU such that one container can occupy only a certain percentage. In essence, resource isolation using OS level virtualization provides multiple system call interfaces in each container for the same OS. E.g. Solaris Containers and BSD Jails.

- Application Level Virtualization: It lets you emulate one application level interface on another. Examples include JVM as discussed before. Another example is WINE that lets you run windows application on linux or MAC by emulating the Win32 interface. Rosetta is another example that lets Mac PowerPC applications to run on Mac x86 machines by translating the code at the application level.

5.1.3 Virtualization Example

Using virtualization software like VMware or virtualbox, you can run any OS eg. linux (guest) on any other OS eg. Mac OSX (host). The guest OS (VM) is just another process as far as the host OS is concerned. Hence, the guest OS can have multiple processes running but they are invisible to underlying host OS. The virtual machine can also have multiple virtual CPUs to run multi threaded applications and it is up to the virtualization layer to map the virtual CPUs to the actual(host) CPU.

5.1.4 Types of Hypervisors

- Type 1: In this case, the hypervisor or virtualization software runs on bare metal, and the system is booted on the vitualization software(hypervisor), not on an OS. On top of the hypervisor multiple virtual machines can be running with same hardware environment, but with different OSs. This type of hypervisors are typically only seen in server environments.

- Type 2: In this case, the machine runs some native OS. The type 2 hypervisor runs as an application on top of the OS. This hypervisor emulates the same underlying hardware and exposes the same hardware interface. In there you can create a virtual machine and run a second OS, which could be different from the first OS. E.g. Windows could be run on a Mac OS X. To the host OS, the entire virtual machine looks like a single process. It has no knowledge of the OS and processes running inside the virtual machine.

5.1.5 How Virtualization Works

In relation to hardware architecture and processors, there is a notion of protection rings. Rings define levels of protection in terms of what privileges a process has when it runs at that level. The OS kernel runs at ring 0, and has full control over things happening in the machine. User processes run at ring 3 i.e. they can only run some restricted set of instructions, not including certain special instructions that only the kernel can run. These rings offer protection from user processes that can corrupt components and cause the system to come down. Instruction sets on any CPU can be partitioned into instructions that only kernel can run, and those that any process can run. The instructions that only the kernel can run are called *sensitive instructions*, e.g. instructions that can change page table settings or instructions that trigger I/O. If user processes try to execute these sensitive instructions, their request will be denied, as only the kernel has the privileges to execute them. Any assembly instruction that causes a trap or an interrupt is known as a *privileged instruction*. Example include segmentation faults while running programs, divide by zero, etc.

Claim: It is possible to implement type 1 virtualization only if *sensitive instructions* are a subset of *privileged instructions*. In other words type 1 virtualization can be implemented if the sensitive instructions trigger a trap.

5.1.6 Type 1 Hypervisor

In a type 1 hypervisor, the hypervisor should run in kernel mode as the hyperviosr schedules the VM's, allocates CPU and so on. The user processes that run on the VM will run in user mode. The guest OS that runs in the VM should also run in user mode yet should implement instructions that the kernel needs to execute. This can be done by implementing traps, where the guest OS asks the underlying hypervisor to implement the sensitive instructions on its behalf. Eg. In Intel 386 sensitive instructions executed in user mode are ignored. Therefore, there no way to implement type 1 virtualization for these systems. However, today's Intel CPUs have extra support called VT(virtualization tchnology). If VT support is turned on, on an Intel processor, a bitmap will list the instructions that should trigger a trap, if they are executed in user mode. For type 1 hypervisors, a good way to design the CPU is to have multiple rings, where ring 0 as before is the most trusted mode that the hypervisor runs in. Ring 3 is the least trusted mode that the user processes in the guest OS would run, and the guest OS could run in one of the middle modes. This way sensitive instructions from the guest OS will be executed but sensitive instructions from the user process on the guest OS will not.

5.1.7 Type 2 Hypervisor

In a type 2 setting, there is already an OS running on the machine and the hypervisor runs as a user process. The host OS kernel is trusted and it runs in kernel mode, everything else runs in user mode. Hence, any sensitive instruction that is executed by the guest OS kernel will not be allowed to execute. To solve this problem type 2 hypervisors scan guest OS instructions as they are about to execute, and whenever there is a kernel instruction it is replaced with a function call to the hypervisor (virtualization layer). The hypervisor, in turn, triggers a system call and requests the host OS to execute that instruction. The host OS then executes that instruction for the hypervisor. Hence, type 2 hypervisors require binary translation where guest OS kernel instructions are changed on-the-fly, and are replaced with hypervisor instructions. The disadvantage is that the systems takes a performance hit and runs more slowly. However, the advantage is that no hardware support is needed for virtualization. So, you can use type 2 hypervisors on older systems as well (e.g. on Intel processors without VT support). All sensitive instructions are replaced by procedures that emulate them.

5.1.8 Paravirtualization

Paravirtualization is a technique to run a guest OS on top of a type 1 hypervisor, without requiring hardware support. Up until now, the assumption was that the OS is unmodified. However, in paravirtualization, the assumption is that the OS can be modified. In effect, a programmer modifies the OS by taking each kernel mode instruction and replacing it with a function call to the hypervisor. So, in some sense, it is similar to what was being done in type 2 hypervisors, except now it is not done on-the-fly, but instead, it is done beforehand in the source code, by the programmer. The programmer looks at kernel code, and every time the kernel makes a system call, it is replaced with a function call to the hypervisor. These calls to the hypervisor are called 'hypercalls'. So the programmer replaces all sensitive instructions with hypercalls. Paravirtualization can be implemented without hardware support, as there are no kernel instructions left in the kernel. All kernel instructions are replaced by hypercalls to a type 1 hypervisor, which runs in kernel mode and can execute those instructions on behalf of the OS.

5.1.9 Memory Virtualization

Virtualization needs to deal with all the resources on the machine (memory, disk, network interfaces), not just processors. Similar concepts of virtualization need to be applied to other resources as well. The memory virtualization problem can be described as follows. Multiple virtual machines can reside in memory. The hypervisor needs to allocate memory to each VM. The OS that runs inside each VM is unaware of the fact that it has access to only a portion of the actual RAM, and as far as the host OS is concerned, the VM is a user process. So even though the virtual machine has been allocated a section of memory, the guest OS cannot modify its page tables as it is in user mode. Hence, every time the guest OS needs to modify its page tables it needs to trap into the hypervisor to make these changes. The hypervisor deals with this issue by maintaining a shadow page table that mirrors the page table maintained by the guest OS. The guest OS then triggers a trap whenever it wants to change its page table, and the changes are reflected in both the shadow page table and the original page table of the guest OS.

5.1.10 I/O Virtualization

Disks and network interfaces also have to be virtualized.

- Virtualizing the disk: There is one large physical disk, and the storage on that disk is partitioned and allocated to the VMs that are running. These partitions are called virtual disk files which is a regular file as far as the host OS is concerned. Hence, when ever data is written to disk in the guest OS, it is emulated as a file I/O in the host OS.
- Virtualizing the network interfaces: The physical machine has a ethernet card that is used to do network I/O. The VMs share this physical ethenet card. Each VM is assigned a logical ethernet card and has its own IP address using which it does network I/O. Network packets that are sent by the VMs are multiplexed by the hypervisor to the actual physical ethernet card. Similarly, when packets arrive at the actual network card, the hypervisor figures out which VM those are for, and delivers them accordingly.

5.1.11 Examples

JVM is an example of virtualization at application level. The JVM is an abstract machine. JVM then emulates the logical machine using the underlying hardware and runs the java programs on the hardware.

5.1.12 Virtual Appliances and Multi-Core CPUs

- Virtual Appliances: One advantage of virtualization is that you can download prepackaged VMs, called appliances, that already have pre-installed and pre-configured software, including OS and applications. These appliances can be downloaded and run as is, and have made VMs a popular method of distributing complex applications that require expert configuration.
- Multi-core CPUs: Having multiple cores on one system has made it attractive to use VMs. Rather than slicing a single CPU across different machines, each VM can be run on one core. Multiple cores can also be assigned to one VM. This is often used in datacenter environments or other server environments where there are large multiple core servers. One popular way to use it is to run a type 1 hypervisor, create lots of smaller VMs and inside each run different OSs and applications. So what was earlier only possible with ‘n‘ different physical machines, can now be consolidated into one large server using virtualization and multi-cores.

5.1.13 Use of Virtualization Today

- Data center environment: Newer servers have more resources so it is possible to take three old servers and pack them onto one new server using virtualization. Each old server can be put inside a VM, which in turn, runs in the new machine. There are fewer machine to manage, and power and cooling costs are reduced. This solution has made virtualization popular in the enterprise environment.
- Cloud computing: Virtualization is very useful in cloud computing. This will be looked at in later sections.
- Desktop computing: Users need to develop software and test it on different types of systems. Using a type 2 hypervisor, it is possible to run Linux on a Windows system. So software developers/testers can have access to a Windows machine, without buying one. Hence when the guest OS fails due to some bug in the software being developed, the host OS is not affected.