CMPSCI 677 Operating Systems **Spring 2013**

Lecture 6: February 11

Lecturer: Prashant Shenoy *Scribe: Vani Gupta*

6.1 Virtualization Recap

Virtualization was discussed and defined as mimicking one interface using another. Different types of virtualization (like native, application-level etc.) were discussed. Understanding Type 1 virtualization requires some background on how CPUs is designed. A processor can operate in two modes: user mode or kernel mode. Also referred to in terms of rings of protection: ring 1 or ring 3 (for user mode) or ring 0 (for kernel mode). Out of the list of assembly instructions, some instructions can only be executed by the kernel. The rest of the instructions can be executed in either mode. When the OS is executing, the CPU is in kernel mode. When user programs execute, the CPU is in user mode. The kernel sets a mode bit to 1 when it runs. So when the mode bit is not set to one, the CPU runs in user mode. In a non-virtualized setting, the kernel executes in kernel mode, and user processes execute in user mode. In a virtualized world, the hypervisor executes in kernel mode. The virtual machine (containing the guest operating system and the user programs) executes in user mode. Although the kernel inside the virtual machine is executing in user mode, it will still try to execute all the kernel instructions as it would normally do in a non-virtualized setting. The CPU will not allow these instructions to be executed in user mode. Therefore, in order to enable the kernel in the virtual machine to execute kernel instructions, those kernel instructions in user mode must cause a trap. Whenever a kernel mode instruction executed in user mode causes a trap, it causes the hypervisor to run. Now the hypervisor runs in kernel mode, and can then execute the instructions on behalf of the guest OS kernel. Until recently, Intel processors ignored kernel instructions that were executed in user mode. The newer Intel processors have support for virtualization called virtualization technology (VT). Newer AMD processors also have support for virtualization and are called AMD SVM. In these newer processors, a bit vector can be set dynamically for those instructions that should cause a trap when they execute in user mode. So by setting the bit to one for all kernel instructions, one can cause traps for kernel instructions executed in user mode. Therefore, VT enabled Intel Processors or SVM enabled AMD processors can implement a type 1 hypervisor. Older Intel and AMD processors cannot.

In a virtualized world, the hypervisor does not have the ability to distinguish between instructions coming from the guest OS kernel and instructions coming from a user process inside the virtual machine. So, buggy or malicious user programs could potentially cause the hypervisor to execute instructions that can crash the kernel or take over the system. For this reason more than two levels of protection are used. Ring 0 is reserved for the hypervisor, as it is fully trusted. Ring 3 could be used for user processes, which are the least trusted of all. Rings in between could be assigned to the guest OS kernel, which is more trusted than the user programs, but less trusted than the hypervisor.

6.2 Virtualization Continued

6.2.1 Type 2 Hypervisor

In a type 2 setting, there is already an OS running on the machine. The hypervisor runs as a user process. E.g. VMWare example that was shown last time, VMWare was running as a user process. As the hypervisor

is running in user mode, it is not trusted. The host OS kernel is trusted and it runs on bare metal. The host OS kernel runs in kernel mode, everything else runs in user mode. Now, the guest OS is unaware that it is running in user mode, so it tries to execute kernel instructions as it would normally do in non-virtualized world. When this happens, it will not be allowed to execute them, as it runs in user mode. 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. 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. So type 2 hypervisors require binary translation from code. 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.

6.2.2 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 a 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. Paravirtualization can only be implemented for those OSs, which are open-source e.g Linux. For proprietary OSs like Windows, it can only be implemented internally by the company. Examples of paravirtualized hypervisors include Xen, VMWare ESX server. The VMWare ESX server was paravirtualized to run on older processors that did not support type 1 hypervisors, like older Intel CPUs.

Paravirtualization is efficient for two reasons: 1) There is no translation on the fly; 2) The virtual machine is running on a type 1 hypervisor, which runs on bare metal. In type 2 hypervisors, there are more layers, and so the overhead is larger.

6.2.3 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. So it thinks it is running on real hardware, and has full control of memory as it would normally have in a non-virtualized setting. Therefore, a portion of the RAM needs to be carved out for each VM and it needs to be able to address it like the actual RAM (with addresses starting at 0 and going to 'n', although the starting actual memory location will be an arbitrary location not necessarily 0). The hypervisor needs to deal with page table mapping as well. The OS manages page tables for all processes. The page table data structure tracks which pages of memory are allocated to which process. Typically manipulating page tables requires manipulating hardware. So, these are kernel

mode operations, and user processes cannot perform them. So the hypervisor needs to do it on behalf of the OS. The hypervisor maintains shadow page tables that mirror actual page tables. So there is a copy of page tables that the OS manages and copy of page tables that hypervisor manages. There is a one-to-one mapping between the two. If the OS tries to modify a page table, a trap is generated, and the hypervisor makes those changes. These changes are also reflected in the real page tables, as it is a mirrored copy. The OS thinks it modified the actual page table, although it does not have the privileges. Similarly, page faults are dealt with using interrupts. The process generates an interrupt that is handled by the hypervisor, and same concepts apply here. In essence, the hypervisor needs to provide an illusion as if the memory allocated to the VM is the RAM on that machine. Any modification to OS level data structures including memory management data structures have to be handled by the hypervisor on behalf of the OS.

In some VMs, resource allocation is static at the RAM level, but in other cases, hypervisors can change memory allocation on-the-fly. Some VMs allocate more memory to VMs than what is available in reality, in the hope that all the VMs will not use the maximum allocation at the same time (like virtual memory). This is called overbooking. At the page table level, memory allocation is dynamic as processes come and go.

6.2.4 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. Each VM thinks that it is accessing a physical disk. This is achieved by creating a virtual disk abstraction, where all the translations from addresses of the virtual disk to addresses of the physical disk are done by the hypervisor. This can be done in many ways. Actual partitions can be created on the disk and allocated to the VMs, or the entire virtual disk can be a file on the disk.

- 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 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.

6.2.5 Examples

Java is an example of virtualization at application level. The JVM is an abstract machine. It has an API that it exposes to Java programs, and you implement the JVM using underlying interfaces, e.g you could write it in C. Java code runs regardless of the underlying hardware, therefore it is portable. Rosetta was a technology that Apple introduced for Mac when they switched from PowerPC processors to Intel processors. So, Rosetta is an application-level abstraction that translates PowerPC instructions to equivalent Intel instructions.

6.2.6 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 (e.g. some could have 16 cores). One popular way is to use 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.

6.2.7 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. When users have transient needs for machines, they need not buy one, instead they can rent a machine by the hour, on the cloud. It is cheaper as users only pay for the time they are actively using the machine. Renting machines on the cloud has been made possible through virtualization. Users are not trusted, so virtual machines provide a way for system resources to be rented without unknown users having the ability to run malicious/buggy code and crashing the cloud's systems. Cloud providers control the hypervisor, and they give a user a VM with root access. This way untrusted applications only impact the VM they are running on. So virtualization is an attractive way to implement renting of resources, while retaining control of the infrastructure, and having a level of security.

- 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. In addition, virtualization allows user to take advantage of thin clients and virtualized desktops. The actual PC is a VM that sits in a cloud or a datacenter. Users use that PC through think clients like remote desktops, VNC etc. Many enterprises are switching to this model as it is easier to manage. All their PCs are virtualized and users connect to those PCs using desktop applications. This makes IT management centralized and therefore easier.

6.2.8 Case Study: Planet Lab

Planet Lab is a distributed set of machines (contributed by many different universities) that are used for research. It is used mainly for networking/distributed systems research. If a student doing research in, say, networking needs a distributed set of machines at 'n' different locations, they can get them from Planet Lab. Planet Lab makes this possible via virtualization. They have many physical machines physically sitting at several different locations. They create a VM at 'n' of those locations and give the student access to them. Where as most others have used hardware level virtualization, Planet Lab uses OS level virtualization. They assign a slice of the machine to a user by giving them a container (like the Solaris container or BSD jails that were described earlier). Therefore, the user does not get a VM running another copy of the OS, but they get a container with the allocated amount of resources (including memory, CPU etc.). This enables users to run experiments without impacting other users. This could have been implemented using hardware virtualization as well, however, OS level virtualization makes it more efficient as another copy of the kernel does not need to be there for container.

6.3 Code and Process Migration

As part of distributed scheduling, code/processes may need to be migrated from one machine to the other. In some cases, entire VMs may need to be migrated. Both these cases will be discussed in this chapter.

6.3.1 Motivation

- Performance and flexibility are two important reasons why code/process migration is done. By moving code/processes, the load on a system can be redistributed for better performance. The program can be executed on any machine, which offers the flexibility of not being tied to just one machine.
- Process Migration (aka *strong mobility*): In this case, a process that is running is moved to another machine. This is more complicated than code migration, as every process has state (like an address space and resources) that needs to be transferred, and then it needs to be resumed from where it was suspended.
- Code Migration (aka *weak mobility*): In this case, code is shipped from one machine to another. Code migration can only be done before the process begins, once the code starts executing, it is essentially process migration. Code migration is somewhat simpler as only the code needs to be moved and started on the receiving machine. This is much more common than process migration. Here are some examples: 1) Filling a form on a browser. Code that checks the format of entered texts is migrated from the server to the client machine, and runs in the context of the browser; 2) A Java applet is another example of of code migration, where the client downloads the applet code from sever and executes it on the local machine; 3) SQL queries or keywords search queries also generate code that is sent to the server for processing

Downloading drivers on-the-fly, as and when needed, can now be done on newer versions of Mac. So suppose a printer is plugged in for the first time, the OS offers to download it for the user. The OS goes to a driver repository, and downloads the appropriate driver, configures it on-the-fly, and the user can start using the printer. Once the printing is over, the driver can be thrown away or decommissioned. This allows for dynamic configuration of machines on the fly. Neither is there a need to have all the drivers on the machine at all times, nor does the user need to have the driver with every new hardware device they would like to use. This type of flexibility is the motivation behind code and process migration.

6.3.2 Migration Models

- Processes have an address space that consists of a code segment, a stack segment, and a heap segment. If we migrate a process we must at least transfer these three segments, and instantiate them in the context of the new process.
- Migrating code is simpler. The binary code or the script needs to transferred and executed. The process is instantiated after the code moves.
- Migration can either be sender-initiated or receiver-initiated. There is a difference between whether the sender pushed the code (sender-initiated) or whether the receiver requested it (receiver-initiated). E.g. code that checks the format for entered text in online forms is sender-initiated. A Java applet is receiver-initiated.