Virtualization: Mechanisms

1 Introduction

Hello, and welcome to this lesson where we explore mechanisms for virtualization.

In the previous lesson we focused on the What and Why of virtualization, and in this lesson we will turn our attention to the How. We will explore how virtualization works.

First, we will look at an example to get a feeling of what the issue is specifically, and get a general feeling of what the hypervisor has to deal with when running virtual machines.

We will then explore three techniques to achieve system-level virtualization, namely, by de-privileging the guest OS to run in user mode instead than in kernel mode. The idea is that whenever the guest OS attempts to execute a privileged instruction, which requires kernel mode to execute, an exception occurs, and the hypervisor takes over. This is often also called "Trap-and-Emulate" virtualization.

Trap-and-emulate does not always work as we will see, and in these cases other approaches are used. One of them is para-virtualization, where we carve out the problematic instructions from the guest OS and move them into the hypervisor.

A third approach is to translate the binary on the fly and replace problematic instructions as they occur. This approach is very tricky, but has been very successful in the hands of VMWare.

As we explore each approach we will discuss their pros and cons. At the end of this course you will understand the difficulties in implementing virtualization. You will be able to describe and evaluate the three main techniques for virtualization. You will also understand the need over the last years to extend the CPU instruction sets to better support virtualization.

Let's begin.

2 Virtualization – What is the Issue?

Let's look at an example of some operating system code.

This code loads an address into the page table base register, CR3. In order to do that we mask interrupts using the Clear Interrupt Flag command, clear the last 12 bits of the address, load it into CRE, and enable interrupts again by setting the Interrupt Flag again. This code works perfectly fine as long as there is only one operating system running in the system. For a system with multiple virtual machines, each running its own copy of an OS, this code would not work.

Let's look at why as we go through the code.

The first move instruction is innocuous.

Having the guest OS execute a Clear Interrupt Flag instruction poses a problem, however. Rather than masking interrupts for the virtual machine, it would mask the interrupts of the entire system, affecting all virtual machines. This is clearly wrong. Rather than masking all interrupts, we would like to mark the fact that the interrupts for the running virtual machine are masked. We therefore set a flag, which we may call Virtual Interrupt Flag to zero.

The next instruction is innocuous.

The guest OS should not be allowed to change the page table base register. Instead, we handle setting CR3 in a separate function. The details of what is done here will be discussed in the lesson on virtualization of memory management.

Finally, enabling interrupts should not be done by the guest OS either, as this enables interrupts for the whole system as opposed to just the virtual machine. The desired code would set the Virtual Interrupt Flag for this virtual machine to 1. Because we handle interrupts basically in software, we have to explicitly check whether there are pending interrupts. If there are, we handle them, here by calling the function `handle_interrupts`.

While we know what the desired execution of code is given what the guest attempts to execute, the question is how to get from the guest execution to the desired execution. This question is at the core of virtualization.

3 Techniques in Classical Virtualization

There are four basic techniques for virtualization.

The first is to de-privilege the guest OS and have it run in user mode. Whenever the guest executes an instruction that requires kernel mode, an exception occurs and the hypervisor takes over. Because of this it is also called "Trap and Emulate". This approach is popular because it does not require changes to the guest OS code. Unfortunately, not all instructions that should trigger exceptions actually do, and so other solutions must be found.

One such solution is so-called "para-virtualization", where the problematic instructions are removed from the guest OS and replaced by high-level calls to the hypervisor.

An approach that used to be popular was to support interpretive execution as part of the instruction set. This was an execution mode where the hardware was aware that it was running a virtual machine, and problematic instructions were interpreted within the context of the virtual machine by the hardware.

A successful approach is to literally replace problematic instructions on-the-fly through binary translation. This is very tricky to do, in particular on some instruction sets, but VMWare has successfully pulled this off.

4 Trap & Emulate

Let's start by exploring Trap&Emulate.

Trap and Emulate relies on the assumption that all problematic instructions throw an exception if they are executed in user mode. This allows to capture the execution of these instructions in exception handlers and to replace their execution by specifically tailored emulation code in the exception handler. Let's look at the example we know from before.

As we pointed out the first instruction is innocuous, and nothing happens.

The mask interrupt instruction is problematic, and should throw an exception if we run it in user mode.

The exception handler now executes the desired instructions.

When it returns, the execution continues, when we try to set the page table base register it traps again.

And the exception handler takes over.

Another exception occurs when the guest OS tries to unmask the interrupts.

4.1 Trap & Emulate: Definitions

This approach looks promising, in particular because it does not requires us to change the guest OS at all. We simply run the guest OS in user mode and handle the exceptions that are triggered whenever it attempts to execute a problematic instruction. But does this approach actually work? To answer this, let's define a formal framework to describe the trap-and-emulate execution of the guest OS. For example, until now we have talked rather "flippantly" about "problematic" instructions. What do we mean by this? It's time for a few definitions. These come from the seminal paper by Popek and Goldberg from 1974.

We define the "privileged state" of the machine to be that part of the system state that describes all kinds of resource allocations, for example the addressing state, such as page tables, the TLB. Also to the privileged state belongs the processor mode, that is, whether the processor is in user mode or in kernel mode. In a virtualization environment, changing any part of the privileged state change the state for all virtual machine, not just the running one. We can then separate instructions by their relation to the privileged state.

Instructions that can change the privileged state are called "control sensitive" instructions. Examples are instructions that manipulate the status register of the system, or the return from interrupt instructions. The latter changes the processor mode from kernel to user. It may also unmask the interrupts and grab the next queued interrupt.

Instructions that expose the privileged state are called "behavior sensitive" instructions. Such instructions allow the virtual machine to infer whether it runs on the native hardware or in a virtual machine.

An example is loading a physical address.

We group control sensitive and behavior sensitive instructions together as simply "sensitive instructions".

We have defined sensitive instructions.

On the other hand, we call instructions that are neither control nor behavior sensitive "innocuous" instructions. We say that an instruction is a "privileged instruction" if it runs fine in kernel mode, but throws an exception in user mode under otherwise exactly the same circumstances.

4.2 Trap & Emulate: Requirements

Once we have these definitions in place, one can formulate a set of requirements that need to be satisfied for the architecture to support Trap-and-emulate virtualization.

Popek and Goldberg formulated the following theorem, which states that an any conventional machine, a virtual machine monitor may be constructed, they meant constructed using trap-and-emulate, if the set of sensitive instructions for that machine is a subset of privileged instructions.

In English, they were meaning to say that a machine is virtualizable by Trap and Emulate if all sensitive instructions are privileged.

We remember that by sensitive we mean either control or behavior sensitive, that is, it either modifies the privileged state or has access to it.

Popek and Goldberg were able to prove that this simple theorem was correct in predicting whether a particular Instruction Set Architecture supported Trap-and-Emulate virtualization or not.

4.3 Obstacles to Trap & Emulate Virtualization

The unfortunate reality is that now real-world system supports Trap and Emulate virtualization. And this is for two reasons.

One is that CPUs often allow the guest OS to see the privileged state. Once it has access to the privileged state, duplication of the virtualization is violated. The guest OS can know whether it runs on the native system or in a virtual machine.

On the x86, for example, the current privilege level is stored in the code segment register, which can be inspected by the guest.

Similarly, the interrupt descriptor table can be accessed through virtual memory accesses.

The problem with these instructions is that they are behavior sensitive but not privileged.

A second problem are instructions that are control sensitive but not privileged.

For example, some instructions that should be privileged simply don't execute, thus act as NOOPs, when called in user mode, rather than generating a trap.

Another example is the "pop flags" instruction, which pops the system flags off the stack and pushes them into the EFLAGS register. Among these flags is the interrupt flags, which controls whether interrupts are enabled.

4.4 Example: More Troubles with `POPF`

The situation with "pop flags" is actually a bit more tricky.

When the x86 calls POPF in user mode, no exception is thrown. But the interrupt flag is not updated. This makes sense, as otherwise any user could indiscriminately mask interrupts. But the problem is that a guest OS running in user mode has no idea whether to enable interrupts or not. There have been many other instructions like this.

Also it looks like we are singling out the x86, but all other CPU architectures, such as the MIPS or the ARM, have similar problems.

It must also be said that most current architectures have added virtualization extensions over the years that have eliminated many of these problematic instructions.

5 Para-Virtualization

For systems that don't support trap-and-emulate one can use other approaches. One of them is **para-virtualization**.

In a trap-and-emulate environment we have the unmodified guest OS handing over control whenever privileged instructions are to be executed.

The interface of the guest OS is so-to-say the instruction set of the hardware. The problem is that some instructions are problematic, and trap-and-emulate does not work. The approach used in para-virtualization is to identify these problematic portions of code and have them handled by the hypervisor.

The interface to the guest OS is now not the instruction set of the hardware anymore, but a modified version of it. We call this the **"para-API"**. In order to make use of the para-API we need to modify the original guest OS, The result is the guest OS prime, depicted here with the updated interface. Rather than executing problematic instructions, the para-virtualization-aware guest OS issues high-level calls to the hypervisor.

The problem clearly is that one cannot run a normal distribution of an OS that is not para-virtualization-aware on top of a para-virtualized hypervisor. For open-source OSes this is not a

problem, as it is easy to generate versions that are para-virtualization-aware. One has access to the source code after all.

For closed-source OSes this is trickier. The Xen hypervisor uses para-virtualization and address the issue of making closed-source OSes para-virtualization-aware by providing specially crafted device drivers to be installed with the OS. These device drivers handle memory management, device access, and other potentially problematic portions of OS code.

6 Binary Translation

Another technique for virtualization is **Binary Translation**. The idea is to inspect the code as it is executed and replace problematic instructions on-the-fly.

On-the-fly translation has been used in software-based virtualization solutions before, but it has generally been too slow because the instructions were literally interpreted in software.

Binary translation is different, as most of the code remains the same.

Only sensitive instructions are replaced by emulation code. Also, once a portion of code has been translated, it does in general not have to be re-translated again during the execution of the program. In this way binary translation is a bit like on-the-fly compilation of the binary code to safe binary code.

This approach is called binary translation because it translates the binary code, and does not need access to the source code.

The translation happens at run-time and only the code that is being executed needs to be translated.

The input of the translator is x86 code and the output is safe code.

Let's illustrate how this translation works on a little example of C function that computes whether a given number is prime.

The function simply iterates through all the integers smaller than the argument a, starting from 2, and checks whether a can be divided by one of them without remainder. If it does, A is not a prime, and we return 0. Otherwise we return 1 for success.

Let's pass this through a compiler and see what comes out.

First, we store the argument in Register EXC.

Next we initialize the iteration variable ESI.

We then check if we reached the end of the iteration. We do this in two places because we split the for loop into an if branch and a do/while loop.

At the end of each iteration we increment the iteration variable ESI.

Now we divide A (stored in ECX) by the iteration variable ESI and check whether the remainder is zero.

If it is we jump to label "notPrime", which returns a 0 in Register EAX.

If we reach the end of the iteration, the argument was a prime, and we set Register EAX to 1 and return.

Now the binary translation works as follows: Think of the code to be executed as a stream of instructions that is read by the system. This stream of instructions can be just as well read by the binary translator and presented to the system for execution after possibly some modification.

The binary translator reads from the instruction stream, and stops after a few instructions or after it reads a control flow instruction. This batch of instructions is called the Translation Unit.

It then inspects the instructions in the Translation Unit.

Innocuous instructions are left alone.

The remaining, presumably problematic instructions, are translated.

The result is a modified batch of instructions, called the "compiled-code-fragment", which is forwarded to the system for execution.

As an example, when the system starts executing our little prime check function, the instruction stream contains these instructions, which we represent here in assembly code. As we pass this Translation Unit through the binary translator, The following compiled-code fragment is generated. We notice that the most of the instructions are left alone. This comes to no surprise, given that the function really does very simple computation and does not access system resources in any way.

The two jump instructions at the end may be a bit strange. They are because, as the binary translator makes its way through the code as it is being executed, it generates code in an order differently than what has been generated by the compiler. Some of the jump destinations have not been generated yet. The translator will fill them in as it needs them.

Let's observe what the binary translator does as the system is to execute the function "isprime" with argument 49. On the left we have the original code as generated by the compiler.

When the function is called, the first translation unit is called in. It stops after the jump if greater equal instruction. All instructions are innocuous, and are not translated. The address for the jump instruction is left open.

We read the next Translation Unit, which again contains only innocuous instructions.

Here we increment the iteration variable. All innocuous instructions again. We keep iterating in this code until the iteration variable reaches 7, and the division shows no remainder.

Now we translate the code to return 0. We notice that since we now know where the code for notPrime prime is located, we can update the jump instruction when we check for the remainder.

We also notice that we don't need to translate all the code, only the code that is executed. In this execution the label "prime" is never executed, and so we don't need to translate it.

6.1 Binary Translation: Observations

Since very early on, VMWare has been using binary translation in their prototypes and products. The performance is comparable or better than Trap-and-emulate because it does not incur context switch penalties.

The translation process is scalable. Early experiments with Windows XP boot and halt generated many tens of thousands of translation units.

Also, the generated code has good instruction cache locality. This is for two reasons. First, the code is translated in the order it is executed. The memory layout of the executable is therefore very close to an execution trace of the guest code. This is great for instruction caches. In addition, rarely executed code never gets translated and therefore the remaining code is much more compact.

6.2 Most Instructions need no Translation!

We noted in the example that most instructions don't need to be translated. Only two types of instructions do.

One type of instructions are those that are affected by the code layout changes caused by the translation. These are instructions that rely on PC relative addressing and other control flow instructions, such as branches and jumps.

The other are the sensitive instructions, which need to be translated into safe code. In some cases the translations run faster than the original instructions. For example, masking interrupts may take a long time, but is typically replaced by a simple setting of a software flag, which can be done very quickly. Some other operations, such as context switches, may take much longer to execute safely.

6.3 What about User-Level Code?

A point needs to be made that in the vast majority of the cases, user-level code does not need to be translated.

While the guest OS code must be binary translated to make sure that it is correctly virtualized,

The user code can be executed directly. As a result, the applications and other user-level code run at native speed.

7 Summary

With this discussion of binary translation we have come to the end of our exploration of Virtualization mechanisms.

In this lesson we have learned how virtualization works.

The issue is that the guest OS wants to manage the entire system, and it is the hypervisor's job to make sure that the guest OS only manages its own virtual slice of the overall system, without affecting the other virtual machines.

We explored three approaches to do that, namely Trap&Emulation, Para-Virtualization, and Binary Translation.

In the next lesson we will explore how memory and devices are virtualized.

Thank you for watching!