# Hardware Assisted Virtualization: The Virtual Machine Monitor

G. Lettieri

6 Nov. 2018

In the introduction we have seen that hardware-assisted virtualization is already used by multiprogrammed kernels to implement virtual CPUs and virtual memory. The same idea can be used to implement virtual *machines*, composed of full processors, memory and I/O peripherals. When used in this context, hardware-assisted virtualization is just another VM technique that can be compared to emulation and binary translation. Hardware-assisted virtualization is usually much faster than the alternatives, but it can only be used when the host machine and the target machine share the same basic architecture, so that a large fraction of the target instructions can be executed directly by the host processor.

Even if we are using some of the host hardware directly, we must not forget that we are still implementing a *virtual* machine. In particular, all the state of the virtual machine (the state of its processor, of its memory and of its peripherals) is implemented as data structures in the host memory.

In the following, we are going to describe the general structure of a system based on hardware-assisted virtualization, and how the several parts of the target machine are emulated.

## 1 The Virtual Machine Monitor

In the context of hardware-assisted virtualization, it is very common to introduce the concept of a Virtual Machine Monitor (VMM). The role that the VMM plays with respect to the software of the target machines is very similar to the role played by the Operating System Kernel w.r.t. the user software in a multiprogrammed system (indeed, early kernels were also called monitors).

The kernel of a multiprogrammed system creates several *processes*, at least one for each user program, and then it oversees their execution so that programs cannot interfere among themselves and with the kernel (see Fig.1). The VMM creates several virtual machines, one for each target machine, in order to run the target software inside them. The software running inside a VM is often called the *guest*, as opposed to the host, which we already know. The VMM will then oversee the execution of the guests, so that they cannot interfere among

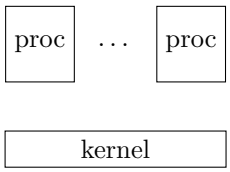


Figure 1: A multiprogrammed system.

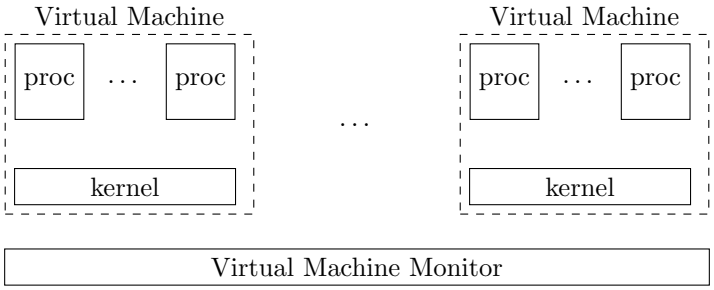


Figure 2: A Virtual Machine Monitor.

themselves and with the host. Note that each guest may be a multiprogrammed system, with its own kernel and processes (see Fig.2).

The VMM may be entirely dedicated to the execution of virtual machines, like Xen, or it may be a part of a general purpose system that can execute normal processes alongside virtual machines (e.g., KVM in Linux).

## 2 Emulating the target CPU

The distinctive feature of this technique is how we emulate the target CPU. First, the VMM loads the state of the virtual CPU into the host processor, then it lets the host CPU run the target code as it is, until the CPU meets an instruction that (for one of several reasons, to be examined later) cannot be executed directly. At this point the VMM must regain control of the host CPU, saving the CPU state back into the virtual-CPU data structure, emulating the offending target instruction in software and then re-loading the virtual CPU into the host CPU and continuing.

The host CPU alternates between direct execution of the target software and the execution of the VMM. Even when the host CPU is executing the guest code, however, the guest cannot have *complete* control of the host: the host CPU must be able to detect the instructions (or the events, such as external interrupts) that cannot be directly executed and, upon detection, forcibly return the control back to the VMM, so that the VMM can emulate them. The guest must not be aware of these "traps" and must have no way to prevent them.

If the context switches between the VMM and the guest are sufficiently rare,

this technique is obviously much faster than emulation and binary translation. We will see that context switches, however, are very costly. If they are too frequent, an hardware-assisted VM can become slower than a software-only one.

Note again how what we have described above looks very similar to how a multiprogrammed kernel operates (Figure 1): the kernel loads the state of a user process and gives it partial control of the CPU. The user does not have complete control, since the CPU is put into a "userspace" mode before the user receives it. If the CPU detects an interrupt, a trap or an exception, it forcibly returns the control back to the kernel. The similarity is strong and, in the previous lecture, we have indeed shown how a multiprogrammed kernel can be seen as a special kind of hardware-assisted virtual machine. There is another respect in which a VMM is more similar to kernel than to an emulator or binary-translator: like a kernel, and unlike an emulator/binary-translator, a VMM must run at an high CPU privilege, since it must have complete control of the host resources.

So, is a VMM exactly the same as a multiprogrammed kernel, at least as far as the CPU emulation is concerned? No, it is not. There are many things in common, but the VMM must be able to do something more. A multiprogramming kernel virtualizes only a *part* of a full processor—namely, only the userspace visible part. There are many registers and CPU-accessed data structures (e.g., the interrupt descriptor table) that are invisible to the user processes (or, better, that are not part of emulated target machine). In a virtual machine, instead, we want to emulate the *entire* processor, since we want to run all the software that runs in the target machine, and this includes the guest operating system software. In Figure 2 we can see a VMM that is emulating several VMs. Inside each VM there is a multiprogrammed guest, with its own kernel and user processes.

In the target machine, the guest kernel has access to the privileged registers and instructions of the target processor. For the Intel x86 processor these would be the `%cr3` register (pointer to the page directory), the `%idtr` register (pointer to the interrupt descriptor table), ..., plus the instructions that manipulate them. A multiprogrammed kernel simply kills a userspace process that tries to manipulate these registers, while a VMM must allow the guest to issue them.

## 2.1 When to trap?

Which instructions and events can be executed directly, and which ones must instead be trapped? The answer depends on the particular host CPU architecture and on how the VMM has been implemented. From a theoretical point of view, we need to prove that the V-state and the T-state remain consistent forever. Assume that we want to know whether a given instruction can be directly executed or not. In order to able to prove consistency, we may reason as follows: assuming that the virtual and target machines start from the "same" state, immediately before the execution of the given instruction,

- what happens in the target machine (what is the new T-state) after the

execution of the instruction?

- what happens in the virtual machine (what is the new V-state) *and in the rest of the host* after the host CPU has executed the instruction?

As usual, we want the new V-state to be consistent with the new T-state. But note that now we have to worry also about the host state not belonging to the VM (this includes the VMM and the state of the other VMs), since we are directly executing instructions on the host CPU and this, in principle, may also affect things outside the VM. In general, any instruction that affects the VMM or the other VMs cannot be directly executed, for obvious reasons. The remaining instructions need to be considered carefully.

Instructions that just manipulate the general registers (e.g. `%rax`, `%rbx` etc. in x86) can be executed directly without any concern. Reading or writing these registers has no side effects, besides perhaps the implicit update of the `%rflags` register. The V-state remains consistent with the T-state and the integrity of the VMM and the other VMs is not affected.

In the following we consider the instructions with memory operands, instructions that access I/O devices, and (guest) privileged instructions related to (guest) interrupts and the (guest) Memory Management Unit.

# 3 Emulating the target Physical Memory

In hardware-assisted virtualization, the physical memory of the target machine is typically implemented using a subset of the host physical memory. Note that part of the host memory will be used by the VMM and other parts will be assigned to other virtual machines. The guest system must only have access to the memory subset the VMM has assigned to it, otherwise it would be able to interfere with other VMs or with the VMM itself. At the same time, the guest must think that it has access to all the physical memory that is installed on the target machine, starting from physical address 0. Figure 3 shows the typical setup. To map the target physical memory (or *guest* physical memory) to a subset of the host physical memory we use the host MMU. Assume that the target machine is in a state where its CPU is accessing physical memory at address $F$. In the corresponding virtual machine state, the CPU is also accessing address $F$, since in hardware assisted virtualization we assume that (for most of the time), the virtual CPU and the target CPU are executing exactly the same instruction. In the virtual machine state, however, the address is intercepted by the host MMU which will translate it to $F'$, an address that lies inside the portion of physical memory allocated to this VM[1]. With this translation in place, we can let the host CPU to directly execute all the instructions with memory operands. Note that some of these instructions may still be trapped by a page fault, if the VMM needs the trap for other reason (see, for example, Section 4 below).

[1] In Figure 3 the gray areas represent the *whole* in-memory translation data structure; for Intel machines these include all the necessary page tables

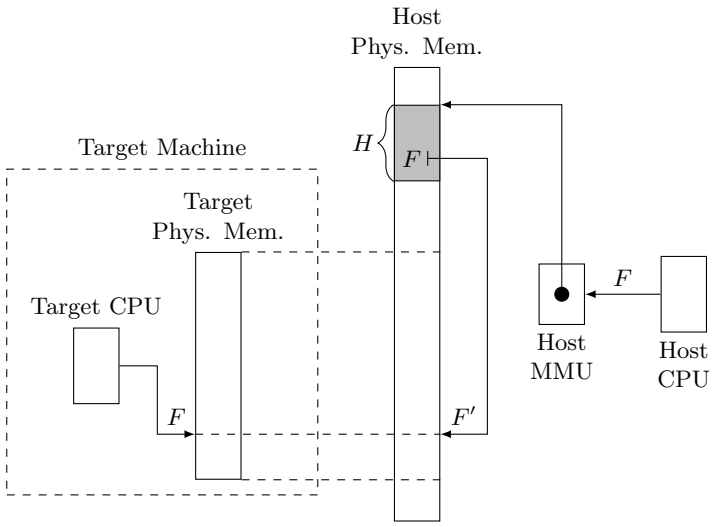


Figure 3: Implementation of the target physical memory using the host MMU for a target machine with no MMU.

Note that this is very similar to what multiprogrammed kernels do with processes. In fact, in the previous lecture, we have shown how also a virtual memory system can be seen as a special kind of hardware-assisted VM. The VMM, however, must do something considerably more complex if the target machine *itself* possess an MMU. We will consider this important case in another lecture.

# 4 Emulating the target I/O devices

I/O is where a VMM and a kernel differ the most. Typically, a multiprogrammed kernel prevents the execution of I/O instruction from userspace and the existing I/O devices are not virtualized. Instead, the devices must be accessed through some abstraction implemented by the kernel (e.g., a file system to access the disk), or they must be used in mutual exclusion (e.g., when accessing a printer). A VMM, instead, must create the illusion that the guest can talk directly to the emulated peripherals, since this is what the guest kernel running inside the VM expects to be able to do. Moreover, the VM peripherals are in general virtual and may have no relation with the host peripherals.

As far as the virtual peripherals are concerned, there is no fundamental difference between an hardware-assisted virtual machine or an emulator or binary translator: each virtual peripheral is implemented as a set of data structures in memory, updated when the the guest reads or writes in the emulated registers or when some external event occurs. So, when the guest reads, say, from I/O address 0x60 in a machine emulating a PC, we still want to call some method

of a software object that emulates a keyboard controller, exactly like we do in an emulator. There are, however, some differences:

- In an emulator, the host CPU is always executing the emulator code, so there is no problem in calling the necessary keyboard-emulation code while emulating the I/O instruction; in a binary translator the call can be added in the translation of the I/O instruction; in hardware assisted virtualization, instead, the I/O instruction is fetched while the host CPU is controlled by the guest code, so we must arrange for the host CPU to trap it and switch back to the VMM, which can then call the keyboard emulation function;

- emulator and binary-translators can be implemented entirely in user-space, where they can (and must) use the available OS primitives to implement the emulated peripherals backends. The VMM, instead, may have to work in a different context, since it is a privileged program. In its context the VMM may have more power (e.g., it has direct access to the host peripherals) but some of the facilities typically available to userspace programs may not be usable, or may be more complicated to use.

So, all I/O instructions typically need to be trapped by the VMM. Note that, in the Intel architecture, this is not limited to the `%in` and `%out` instructions and their variants. Some peripherals may have their registers mapped in memory, and therefore any instruction with a memory operand that actually points to an I/O register must be trapped as well. The VMM can accomplish this task by marking not-present all the pages that contain such addresses.

## 4.1 Emulating interrupts

Let us now consider interrupts coming from the emulated devices. These must not be confused with the interrupts coming from the *host* devices, which must be invisible for the guests and should be handled by the VMM. There will typically be an host interrupt descriptor table, managed by the VMM and inaccessible to the guests, that the host CPU uses to dispatch the host interrupts. Each VM may have its own interrupt descriptor table, whose content is determined by the guest OS kernel. When the VMM wants to emulate the reception of an interrupt in a VM, it must essentially do the same actions performed by an emulator: look up the guest interrupt gate (the address of the guest interrupt descriptor table can be obtained from the relative guest register) save the current state in the guest stack, change the guest instruction pointer according to the interrupt gate and so on. The next time that the guest state will be loaded, the guest will start executing its interrupt handling routine. Note the VMM must know when the guest disables and enables the interrupts: if the guest interrupts are disabled, the VMM must wait until they are enabled again before emulating the interrupt reception. This can be achieved, for example, if the host CPU is capable of trapping the instructions that manipulate the interrupt-enable flag.