

# Intel VMX technology

G. Lettieri

28 Oct. 2015

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

Since the needs are similar, also the solutions used to implement kernels and VMMs are similar. As we have seen, we need help from the hardware so that the actions of the processes and the guests can be monitored while the CPU is obeying their instructions.

## 2 Intel VMX

To solve the problems that we have seen in the previous lecture, both AMD and Intel have extended their CPUs with new features explicitly designed to support

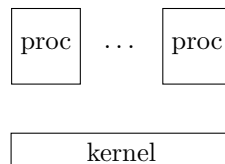


Figure 1: A multiprogrammed system.

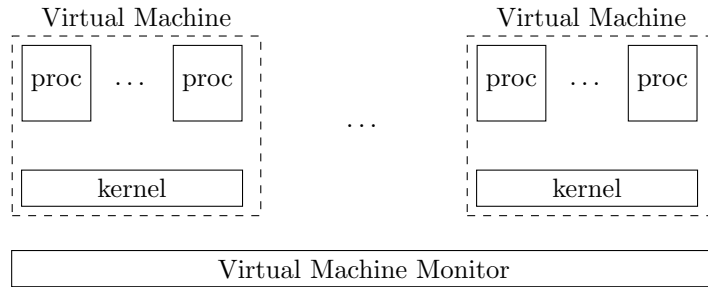


Figure 2: A Virtual Machine Monitor.

Virtual Machine Monitors. AMD and Intel extensions are mostly equivalent, but they are not compatible. In the following we will focus on Intel extensions, called VMX (Virtual Machine eXtensions).

## 2.1 Root and non-root modes

VMX technology introduces two new operating modes in the Intel CPU: the *root* mode and the *non-root* mode. These new modes are orthogonal to the already existing system and user modes, therefore we now have four combinations:

- root/system;
- root/user;
- non-root/system;
- non-root/user.

Root mode is intended for the VMM running on the host, while non-root mode is intended for the guest software running in the virtual machine. Root/system is more privileged than non-root/system, which is more privileged than non-root/user. The main purpose of these new modes is to put hardware-controlled limitations to the actions performed by the guest system software. Whenever the system code tries to execute an instruction that would either violate the isolation of the VMM, or that must be emulated via software, the hardware can trap it and switch back to the VMM. The CPU enters non-root mode via the new `VMLAUNCH` and `VMRESUME` instructions, and it returns to root mode for a number of reasons, collectively called *VM exits*. VM exits should return control to the VMM, which should complete the emulation of the action that the guest code was trying to execute, then give control back to the guest by re-entering non-root mode. All the new VM instructions are only allowed in root/system mode.

Since we have both non-root/system and non-root/user, this architecture allows us to keep the distinction between user applications and OS-kernel inside the virtual machine automatically. For example, while in non-root mode, the

INT instruction may cause a switch from non-root/user to non-root/system, and the IRET instruction may return from non-root/system to non-root/user.

The root/user mode is not directly comparable, in terms of privilege level, to non-root modes. The existence of both root/system and root/user, however, allows for implementations where the VMM is part of a standard OS running on the host, since normal host userspace applications may run in root/user mode, while virtual machines use the non-root modes.

## 2.2 The Virtual Machine Control Structure (VMCS)

Intel VMX adds a new Virtual Machine Control Structure (VMCS) that contains all the information needed to manage the new non-root mode. The VMM may maintain several VMCSs, typically one for each processor of each virtual machine. However, only one VMCS at a time is the *current* one on the physical processor: the processor has a register pointing to the current VMCS, and all VM instructions (such as VMLAUNCH) use the current VMCS. The VMM may use the VMPTRLD to load the address of a VMCS, making it current.

The VMCS data structure has several fields, that may be grouped as follows:

- Guest state: The state of the processor is loaded from here during a VM enter and stored back here during a VM exit;
- Host state: The state of the processor is loaded from here during a VM exit;
- VM execution control: here we can specify what is allowed and what is not allowed during non-root mode; unallowed actions will cause a VM exit;
- VM enter control: it contains several flags and fields that determine some optional behaviours of the root to non-root transition;
- VM exit control: likewise, but for the non-root to root transition;
- VM exit reason: this section contain several informations related to the reason that caused the latest VM exit.

The *Guest state* contains such registers as `%cr3` (the pointer to the page directory), `%idtr` (the pointer to the interrupt descriptor table), `%gdtr` (the pointer to the global descriptor table) and `%tr` (the selector of the current task). The guest state also contains the instruction pointer, which we can use to determine the first instruction that the guest should execute on VM enter, and to find the instruction that the guest was trying to execute before the latest VM exit.

The *Host state* contains the values that must be loaded into `%cr3`, `%idtr`, and so on when there is a VM exit. Since now we have a way to save and restore these registers, the guest software is free to manipulate them without affecting the host. This section also contains the value that must be loaded in the instruction pointer on VM exit. This should be the entry point of a VMM routine that will examine the exit reason, perform the necessary emulation and then re-enter the non-root mode.

The *VM execution control* section contains many flags. The most important ones are:

- a flag that determines what should happen when the CPU receives an external interrupt while running in non-root mode; we may let the CPU serve the interrupt using the guest IDT, without leaving non-root mode, or we may cause a VM exit; in the latter case, the value of the IF flag is ignored (the VMM regains control on external interrupts, regardless of what the guest was doing);
- a set of flags that determine whether some critical instructions should cause a VM exit or not; there are flags for `hlt`, `invlpg`, reading from `%cr3`, writing to `%cr3` and a few other instructions;
- one flag for each kind of exception; so we can say that page fault, e.g., should cause a VM exit, while other exceptions should not;
- a set of flags to cause a VM exit for I/O operations; there is a general flag (VM exit on any `in` and `out` instruction) and a more specific bitmap with a bit for each one of the 65536 possible I/O registers;

The *VM exit reason* section contains a code that specifies the general reason that caused the exit (e.g., external interrupt, I/O access), then several fields that give more informations about the actual reason. For example, if the general reason was “I/O access”, then the additional fields will contain the address of the I/O register and the direction of transfer, among other things. If the reason was an exception, the additional fields will contain the type of the exception, and so on.

The most important fields in the *VM enter control* and *VM exit control* sections are related to interrupts. We give a short description here, but we will have to come back to this topic in another lecture. In the VM exit control section there is a flag that determines what happens during a VM exit caused by an external interrupt. Recall that the interrupt controller sends the request, but then the processor has to reply and obtain the interrupt vector from the controller. Using the flag we can choose between two options: either (i) the processor obtains the vector during the VM exit, and stores it in the VMCS, or (ii) it does nothing. Note that in neither case the processor automatically jumps to the interrupt handler: a VM exit always jumps to the address stored in the Host section of the VMCS. In case (i), the VMM may read the vector in the VMCS and jump to the handler via software (e.g., by using an `INT` instruction). In case (ii), the VMM can use the fact that external interrupts are disabled during the VM exit: by re-enabling them (e.g., with a `STI`), the processor will complete the protocol with the interrupt controller, thus obtaining the vector and jumping to the proper interrupt handler.

In the VM enter control section there are a few fields that can be used to *inject* an event during a VM enter. This are mainly used to make the guest receive a fake external interrupt, but we can also inject exceptions and faults.

## 2.3 Examples

In the previous lecture we have seen a pair of example instructions that were difficult to virtualize on the Intel x86: `popf` and `mov %cr3,%eax`.

The `popf` instruction is difficult to virtualize without VMX since: we cannot let the guest system software execute it at host system privilege, since it may be used to disable host interrupts; we cannot let the guest system software execute it at host user privilege, since this would silently ignore any change in the enabled/disabled state of the interrupts in the virtual machine. Now, using VMX, we can execute the `popf` instruction in any non-root mode. The guest system software is now free to change the value of the interrupt flag, so we do not lose this information. At the same time, the guest system software cannot disable or enable the host interrupts, if the host does not agree: by setting the proper flag in the VMCS structure (which must not be accessible from the guest), the processor will still see all external interrupts while running in non-root mode, independently of the state of the interrupt flag in the `EFLAGS` register.

The `mov %cr3,%eax` instruction is difficult to virtualize without VMX, since it never causes a fault. Recall that, without VMX, we cannot allow the guest software to write into `%cr3`, since this would give the guest full access to all the host memory. Therefore, while the guest is running, the contents of `%cr3` will be different from what the guest has written. This would not be a problem by itself, but it becomes a problem if the guest tries to read from `%cr3`: then it will learn the truth, and we have no way to prevent it. With VMX, we can force the read attempt to cause a VM exit. The VMM will then be able to put into `%eax` the value that the guest expects to see, increment the guest instruction pointer (in the guest section of the VMCS) to skip the `mov %cr3,%eax` instruction, and finally restart the guest from the new state.