Lecture 35 — Virtualization

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

April 17, 2018

Virtualization and Virtual Machines

The word virtualization itself can refer to many different aspects of computing, but the part that we really want to talk about is “virtual machines”.

The goal is to abstract the hardware of a single computer into several different execution environments.

We might have different operating systems running, or multiple copies of the same operating system, depending on what is desired.

Enter the Matrix

From the perspective of the operating system, however, it does not usually know that it is executing on an abstraction of the hardware.

Comparisons to the movie “The Matrix” are apt: “How would you know the difference between the dream world and the real world?”

The Host and the VMM

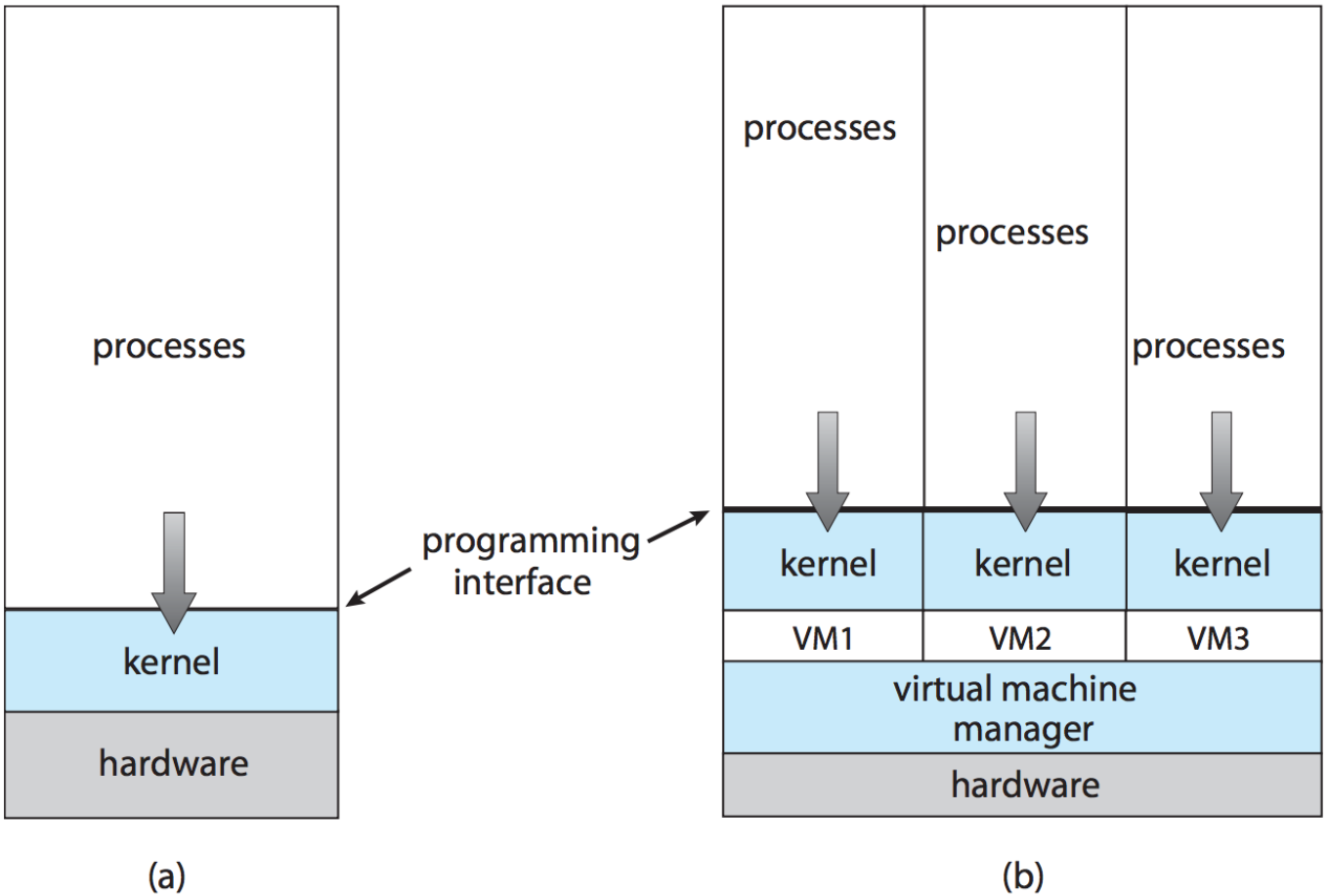
At the lowest level, there is the host, the underlying hardware system.

Above that is the virtual machine manager (VMM) or hypervisor.

The VMM creates an interface that looks like the host.

The guests interact with their own virtual copy of the host.

Virtual and Non-Virtual Machine



Emulation

This is by no means the same thing as emulation.

When we have virtualization both Windows and Linux can be running on the same x86_64 architecture as guests.

In the case of the Android emulator running in an x86_64 machine:

The code of the emulator is running on the x86_64 environment to simulate an Android hardware device that would have a completely different CPU.

Emulation

Thus, an Android app (which would not otherwise run on an Intel/AMD machine) runs in a simulation of a mobile environment.

The emulation operation is incredibly slow, unfortunately, as anyone who has tried to use the Android emulator has found out.

If you are trying to play a classic game that ran under MS-DOS, emulating a 486 computer does not take much by way of resources.

Requires Windows XP

Use an OS other than Windows?

You are likely to have had the situation where one or more programs that are needed for some purpose (work, school...) function only under Windows.

And as the versions of Windows have proliferated and evolved, older programs have sometimes stopped working.

So this necessitates a past version (Windows XP will never, ever die...).

Pause and Play

Another neat thing is the ability to suspend (pause) execution.

This is like pausing a process: the current state is taken and saved, and can be restored at a later time.

Also like a process, it can be moved around to another system and resume after that move has taken place, or cloned to get an identical copy somewhere else.

Protection

A third reason for virtualization is protection.

The guests are isolated from the host and vice versa.

If a virus infects one of the guests, the other guests are not affected.

And with a virtual machine, it will be that much easier to delete and reinstall the guest or roll it back to an earlier state.

This would be exactly the state that we have stored when suspending it...

Consolidation

We often see consolidation in data centres.

Many servers running that could be sharing the same physical hardware.

Instead of having a lot of lightly used physical systems, convert them to virtual and put them all in the same physical machine.

This is combined with a utility: convert a physical machine to a virtual one.

All the system data and configuration and such is copied and turned into a snapshot and this snapshot is then started up inside the virtual machine.

Behind the Scenes: VCPU

One of the key building blocks of virtualization is the virtual CPU (VCPU).

It does not execute code; it is the state of the CPU according to the guest.

The VMM is responsible for maintaining the state of the VCPU.

Much like the process control block, the VCPU is a data structure that is used to store the state when the guest is not running.

The state is restored from the VCPU when the guest is scheduled to run.

User and Kernel Mode

Application processes run in user mode and the kernel runs in kernel mode, having access to privileged instructions.

The guest operating system runs in user mode, but it will want to do some things that require kernel mode.

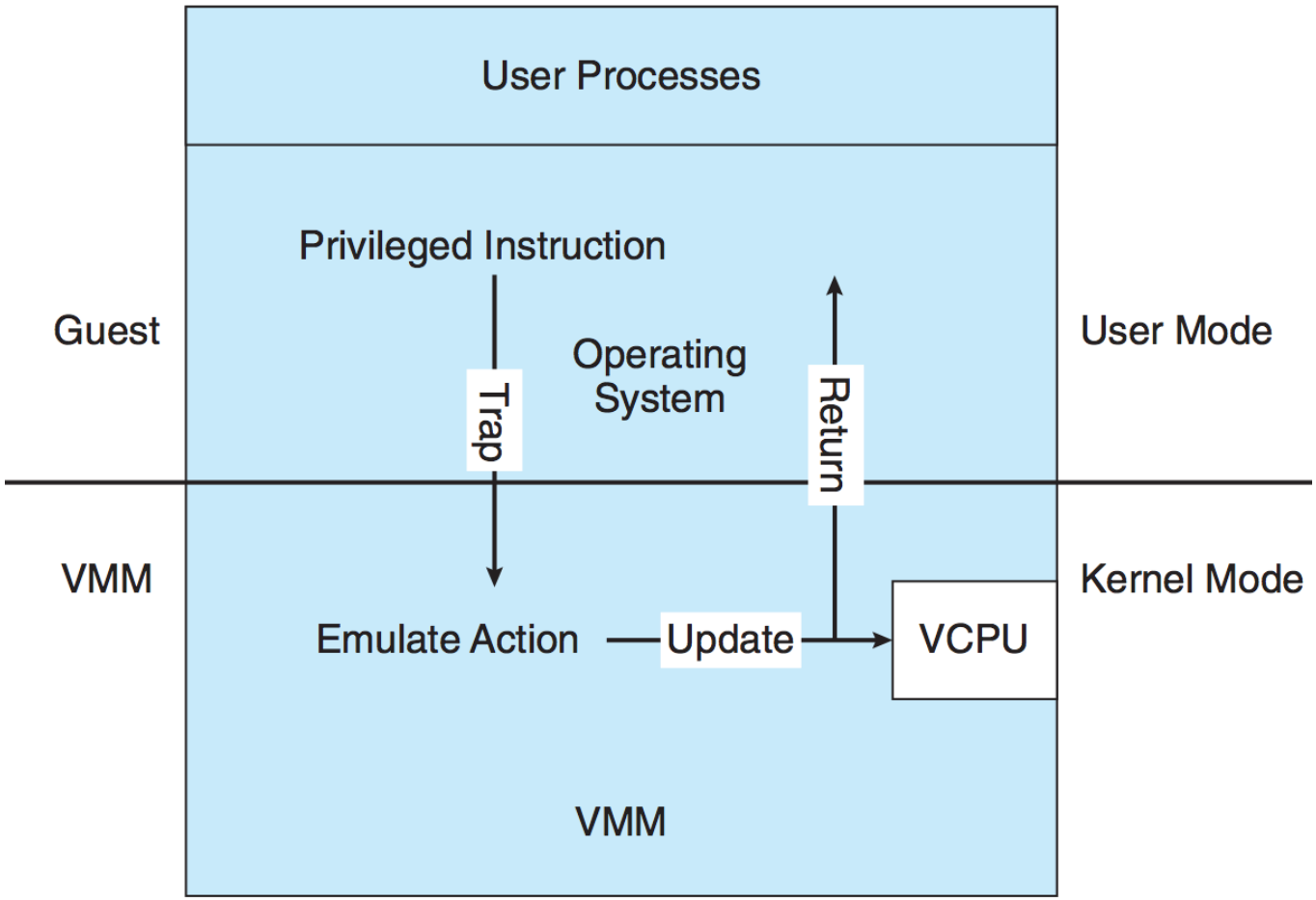
So we will need to have virtual user mode and virtual kernel mode.

Trap-and-Emulate

If the guest attempts a privileged instruction, it will generate a trap (error) because it is in user mode.

The VMM should then pick this up and executes, emulating (or simulating, if you prefer) the requested operation.

Trap-and-Emulate



Trap-and-Emulate Speed

Non-privileged instructions just execute natively on the hardware; they are as fast as they would be if they were being executed outside a virtual machine.

Unfortunately, with trap-and-emulate, privileged instructions have this extra overhead, causing the guest to run more slowly than it otherwise would.

Hardware designers have come to the rescue again: some CPUs have more than just the two simple modes (user/kernel).

They keep track in hardware of virtual user and virtual kernel mode.

That relieves the VMM from the responsibility of keeping track of it.

There's a Catch...

Sadly, some CPUs do not have clear definitions of privileged vs. non-privileged instructions, including the intel x86 architecture.

There were a lot of decisions that make no sense if we look at them with what we know today, but they "seemed like a good idea at the time".

The x86 architecture started back in the early 1970s and we can hardly fault the designers for not anticipating what was going to happen 30-40 years later.

As a Danish proverb of disputed origin says:

"Making predictions is hard, especially about the future."

The x86 has an instruction `POPF` that illustrates the problem.

It loads the flag register from the contents of the stack.

If the CPU is in kernel mode, all flags are replaced from the stack; otherwise only some flags are replaced.

No trap will be generated if `POPF` is executed in user mode, so the trap-and-emulate solution will not catch this and react.

All instructions that fall into this category: special instructions.

Binary Translation

To get around this problem: binary translation.

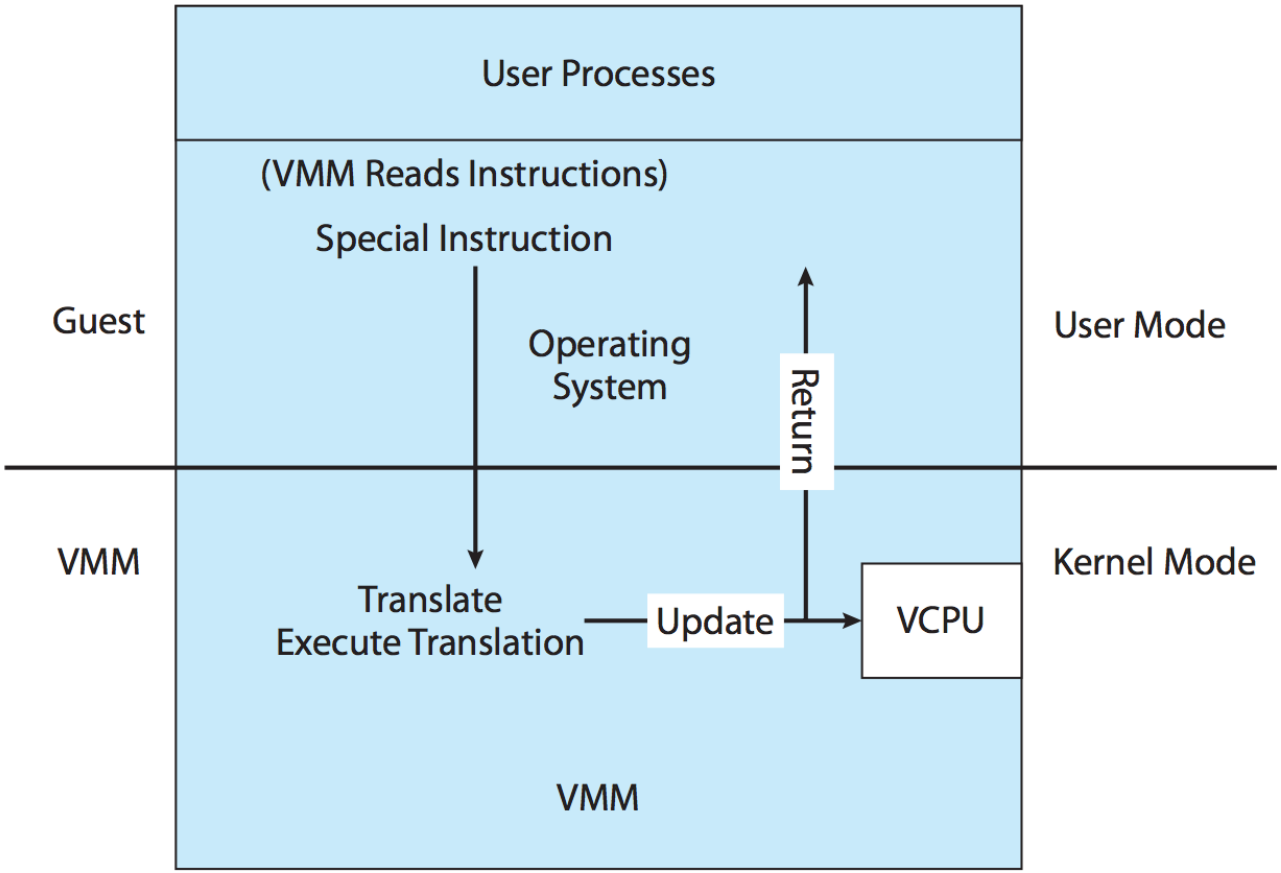
If the guest VCPU is in user mode, the guest can run its instructions natively.

If the guest VCPU is in kernel mode the VMM looks at every instruction before they get to the CPU to execute.

If they are regular instructions, they can execute natively.

If they are special instructions, they are translated into (replaced with) alternative instructions that produce the same result.

Binary Translation



Binary Translation Performance

We have a performance decrease as a result of having to examine and replace some of the instructions.

Most instructions can run natively; only a small number need to be emulated.

The empirical test of “what is being used in industry” seems to indicate that the performance of binary translation is adequate.

There is now widespread adoption of virtual machines running x86 hardware.

One More Hardware Rescue

Additional hardware support makes virtualization work a lot better.

In 2005, Intel added virtualization support to x86; AMD added it in 2006.

This changes dual-mode (kernel/user) processors into multi-mode operation.

The VMM can enable host mode, define a guest's characteristics, and then switch to guest mode, passing control to the guest.

If the guest tries to access a virtualized resource, the VMM takes over.

The Impact of Virtualization

The short version of what happens under virtualization is:
Things get complicated.

There are more demands on and difficulties in resources.

We will consider the impact on scheduling, memory management, I/O, disk.

Virtualized Scheduling

Even if there is only one CPU in the physical machine, virtualization presents one or more virtual CPUs to the guests.

The challenge is to schedule the virtual CPUs' operations on physical CPU(s).

A thread may be a VMM thread or a guest thread.

CPU Assignment

A guest system is configured with some number of CPUs.

As long as there are enough CPUs in the system to meet the allocation commitments (virtual CPU count is less than *n*), we have no problem.

Map each virtual CPU to a physical CPU and we are all set.

CPU Assignment

If the resources are fully committed (e.g., there are *n* virtual CPUs allocated), it gets a bit more interesting.

The VMM does not (usually) need too much time on its own, so it can basically "steal" cycles here and there.

VMM operations run on CPUs that are not busy at the moment, or taking evenly from all the CPUs so as to be "fair".

CPU Overcommitment

If the situation is overcommitted (there are more virtual CPUs than physical ones), the problem is more interesting.

The VMM will have to figure out a way to map the virtual CPUs to the physical ones according to some scheduling strategy.

Like scheduling processes/threads, we can use a scheduling algorithm.

CPU Overcommitment

When overcommitment is the situation, the expectation of the guest operating system of certain time deadlines becomes inaccurate.

If the scheduler in the guest operating system defines a timeslice as, for example, 50 ms, in reality the actual length of a time slice will vary.

It could, in fact, be significantly longer than the 50 ms intended.

This is has a tendency to get the system clock out of whack, but may be fatal for any real-time operating system or any task with serious wall-clock deadlines.

Virtualized Memory Management

Virtualization makes the memory problem a lot worse.

The processes that run take up plenty of memory all on their own.

Now there are multiple operating systems and their structures taking up memory space, too.

The problem is only exacerbated if memory is overcommitted.

Nested Page Tables

The guest operating system, unaware it is in a virtual machine, thinks it controls memory and page table management.

In reality, the virtual machine manager has a nested page table that re-translates the guest's page table to the real (physical) page table.

The VMM can provide double-paging, where it has its own page replacement strategy and tries to help out the guest.

The problem is that the VMM knows less about the guest's memory patterns than the guest itself, so this strategy is inefficient.

Device Driver

The next idea is then to install, where possible, a device driver into the guest that allows the VMM to exercise some measure of control over the guest.

When needed, this “balloon” memory manager is told to request a whole bunch of (empty) memory and asks the guest to pin its pages in physical memory.

This makes the guest think that memory is in short supply;
The guest will start to free up memory.

The VMM knows that the balloon pages are not real and can allocate them to some other guest.

If the memory pressure in the whole system goes down, the balloon pages can be deallocated, allowing the guest to feel as if it has more free memory.

Duplicate Detection

Idea: look to see if the same page is loaded more than once.

This is obviously more likely if the guests are identical (i.e., the same OS).

This will result in significant savings, as the operating system may take up a significant portion of memory on its own.

A hash value for memory may be taken; if two hashes match then a byte-by-byte comparison will reveal whether they are actually identical.

If there is a modification to a shared page, we need to copy the page before the modification is made.

Virtualized I/O Devices

Unlike CPUs and RAM, the guest OSes are likely to tolerate the fact that the I/O devices may change periodically and during operation.

Example: when a user plugs in a USB key or moves into range of a WiFi network.

When running in a virtual machine, a guest may see several virtual devices as if they were real.

Virtualized I/O Devices

The VMM can decide how to allocate I/O devices to guests.

One way is to just dedicate the I/O device to a guest such as a USB key being allocated to one guest in particular.

In other cases, the VMM provides drivers that are just translating commands into the actual device commands.

Virtualized Disk

The typical approach is to create a disk image, that contains all the contents of the root disk of the guest.

This is one big file (though it can be split up into manageable-sized chunks) and as far as the guest is concerned it has the run of that whole disk.

The guest is totally unaware that the disk is just a file inside another system.

This makes it easy to move a virtual machine from one system to another.

Live Migration

One of the really impressive features of virtualization is the ability to migrate a running system from one physical machine to another.

The goal is to copy one system to another in a way that users logged into the guest and programs running on the guest do not notice.

If we were moving users to another physical machine, we would have to warn them in advance, close processes, start them up again...

Live Migration Steps

1. The source VMM connects to the destination VMM.
2. The destination VMM creates a configuration for a new guest: new VCPU, new nested page table, state, etc.
3. The source sends all read-only memory pages to the destination.
4. The source sends all read-write pages (marking them as clean).
5. Any pages that were modified during the previous step (if any) are sent and marked as clean.
6. If there were sufficiently few pages that were dirty in the previous two steps, the source freezes the guest, sends the final state (including VCPU data) and any final dirty pages.
7. The destination acknowledges receipt and begins execution of the guest.
8. The source terminates the guest (how rude!).

Live Migration