WHITEPAPER

QNX | BlackBerry Subsidiary

# Hypervisor Part 1- What is a Hypervisor and How Does it Work?

Chris Ault
Senior Product Manager

QNX Software Systems

5/18/16

# Hypervisor Part 1- What is a Hypervisor and How Does it Work?

## Overview

*While a lot of our customers are familiar with at least the high-level concepts of hypervisors, the knowledge can be spotty – new technologies, terminologies, and applications can create knowledge gaps. This whitepaper's goal is to fill the knowledge gaps by laying the groundwork in an educational manner. A focus on the embedded market place is assumed, with relatively technical explanations.*

## Summary

*A hypervisor allows multiple operating systems to share a CPU (or CPUs, in the case of multicore). While the basis of the technology is half a century old, the applications to embedded systems are new. In this whitepaper, we examine the underlying technology for, and some implementation details of, hypervisors specifically tailored for embedded systems.*

# Introduction

Hypervisors are part of the broad area of computing known as “virtualization,” a concept that's been around for half a century. Ultimately, the goal of a hypervisor is to allow sharing of resources, like an operating system (OS) does. In this whitepaper, we’ll discuss the salient differences between a hypervisor and an OS, and illustrate how the hypervisor works.

## History

In the mid 1960’s, mainframes ruled the corporate computing world (such as it was). But they were relatively rare, and certainly extremely expensive (in the hundreds of thousands of 1960s dollars – millions today). It’s not like today where everyone has their own computer (and in fact, considering all the gadgets we carry around, everyone has multiple computers, all more powerful than the old mainframes). Owing to their rarity and expense, the early mainframe systems provided “timesharing” capabilities. This means that the machine would serve multiple users, giving each user a slice of time during which their job would get executed. Once the timeslice was consumed, the machine would move on to the next user. For simplicity, each user was given a virtual machine (VM). That is, from each user’s point of view, it looked like they had a mainframe computer to themselves. The mainframe provided a hypervisor (kind of an “operating system for operating systems,” if you like) that coordinated the multiple VMs. Because of that, hypervisors are also known as Virtual Machine Managers, or VMMs for short. From the system software designer’s point of view, hypervisors simplified their jobs because each user was (relatively) isolated from each other, and had a simple machine model. The hypervisor parceled out resources (such as CPU time, memory, disk space, etc.) to the VMs running on the mainframe, allocating equitable slices to each user. Even in the early days of mainframes, VMMs allowed different OSs to run concurrently.

Hypervisors are prevalent in today's modern data centres and desktops (think Linux containers, Docker, Xen, VMware, and so on), but have only lately found their way into embedded applications, such as medical devices and automotive infotainment and digital instrument cluster systems. However, these hypervisors have significantly different purposes, constraints, and characteristics from those for data centres and desktops, especially in the safety-critical field.

## VM86

The history of hypervisors in microprocessors is interesting, and bears directly on their architecture today. With Intel’s introduction of the 80386 chip (October, 1985), the first mainstream “virtualization” was born. Recall that the IBM PC was based on the 8088 chip (an 8-bit bus version of the 8086 chip), and could address 1M of memory. Programs quickly outgrew the limited memory of the early PC, and so the 80286 and 80386 chips evolved to provide more memory space for software. However, a desire to run “legacy” software (that is, real-mode 16-bit 8086/8088 programs) was also present, so Intel created a VM86 mode as part of the 80386. While the 80386 was a 32-bit protected mode processor, it could run multiple instances of VM86 virtual machines. We’ll see how the VM86 operated below. It’s important to keep in mind, however, that as far as today’s virtualization goes, VM86 mode was an almost trivial subset of the full power of the 80386’s native 32-bit protected mode. Modern VM implementations map the vast majority of the underlying implementation.

## Protected Mode Virtualization

Interestingly, it would be another twenty years before microprocessors sported full implementation of VMs. In November 2005, Intel released a Pentium IV model, and AMD followed in May 2006 with

an Athlon-64 model that allowed full protected mode virtualization. The ARM architecture got hardware virtualization at the end of 2011 with the introduction of the Cortex A-15 family.

## Theory of Operation

There are similarities and differences when comparing what a VMM does and what an OS does. To understand them, let's step back and examine some details of how programs actually "run." Then we'll look at how the VMM works.

### Thread of Execution

A program runs because the CPU executes the program's instructions, one after another, following the flow of execution dictated by the program. This execution flow is often called a "thread" because it winds its way through the code in memory, much like a thread winds its way through fabric. In this model, the only reason a thread wouldn't continue at the next sequential address is if there was some kind of branch instruction (e.g., a jump, skip, subroutine call, goto – whatever you want to call it) which would cause the thread to continue at a different location. But the key point is that the thread follows the logical flow of execution of the program.

Of course, this one thread doesn't run forever. There are several things that can happen:

- The thread needs to wait for something to happen (e.g., user input, availability of a network packet),
- The thread encounters some kind of a problem (e.g., accesses memory it doesn't have permission to)
- A hardware event occurs (e.g., the timer fires and the OS decides it's time to schedule a different thread).

### How an OS Works

The handling of the above events is what the OS does. When the thread needs to wait for something to happen, the OS is called, puts the current thread to sleep, and starts a different thread. When the thread encounters a problem, the OS steps in, and either corrects the problem or terminates the thread. In either case, a different thread may be scheduled to run as a result. And finally, in the case of a hardware event, the OS either handles the event itself (as in the timer case), or forwards the event on to a different thread (such as a device driver).

Of course, the above is an oversimplification of what an OS does, but it gets the point across. In a realtime OS (RTOS) like QNX, the OS only steps in when it needs to. For the most part, the CPU is running the currently scheduled thread (or threads in the case of multi-core processors). Only when an exception occurs does the OS take over, figure out what to do (perhaps rescheduling some threads), and then gets out of the way again.

### How a Hypervisor Works

In a hypervisor, a similar set of events takes place, except one level higher. Instead of talking about an OS and multiple threads, we're now talking about a VMM and multiple OSs. The VMM schedules an OS to run, and steps out of the way. Just like with the thread and OS model above, the one OS doesn't get to run forever. There are several things that can happen:

- The OS encounters some kind of a problem
- A hardware event occurs

Just like the OS had set up memory regions for its threads in order to define their resource "sandbox," the VMM sets up memory regions (and other characteristics) for its OSs – let's call these "resource sandboxes." When an OS steps outside of its allowed resource sandbox, the VMM steps in and figures out what to do. In the VM case, the resource sandboxes are set up specifically to give the OS the illusion of having its own machine – everything that's "in" the sandbox can be used by the OS, anything outside of the sandbox needs to be mediated by the VMM.

The other reason a VMM takes over execution is because of a hardware event. In order to prevent a given OS from running forever, the VMM traps interrupts from the hardware, such as the timer tick. Based on these ticks, the VMM can manage the scheduling of the various OSs.

## Hardware Sandboxing

A natural question at this point is, "How does the VMM deal with different OSs all talking to the same hardware device?" This is a fundamental consideration in the design and implementation of VMMs and bears discussion. As you can imagine, most hardware is not designed to be concurrently accessed by different, unrelated threads – this is the principal reason OSs have device drivers. The device driver mediates the access from different, unrelated threads and presents itself as a single, unified user to the hardware. In the VMM world, some kind of scheme is required in order to allow different, unrelated OS device drivers to concurrently access hardware devices. In fact, there are three principal methods (in order of decreasing overhead):

- virtualized,
- para-virtualized, and
- physical (also known as "pass through" or "native").

## Physical/Pass Through Devices

The easiest mode to deal with is physical. This is also sometimes called "pass-through" mode, because the OS's accesses to hardware are simply allowed; they're passed through to the hardware, untouched. This is ideal for devices that aren't shared and don't present security risks. For example, a simple serial control port might be dedicated exclusively to one OS. In this case, the hypervisor is configured to allow physical / pass-through access to the port from that OS, and denies access from any other OS (in fact, the other OSs don't even see the device – otherwise they might try to automatically start drivers for it).

In a more complicated device, such as one providing DMA (direct memory access), such access wouldn't be allowed due to security. That's because a malicious (or even simply malfunctioning) program on a given OS could program the DMA hardware to read or write memory outside of that OS's sandbox.

So what happens with devices that are shared? For example, an Ethernet port connects several OSs to the network. But not all OSs can have access to the hardware; their drivers are written to assume that they alone have exclusive access to the device. This is where the other two options come into play.

## Virtualized Devices

In virtualized mode, the device is completely emulated. The way this is set up is that the VMM is configured such that any accesses within the device's address range should be trapped (effectively, the device is placed outside of any OS's sandbox – no OS has access to the device). A handler is set up for the trap, and the VMM supplies code that emulates the device. This means that when an OS accesses a configuration register on the device, the hardware generates a trap. The VMM's trap handler examines the OS's request, figures out what it "means" in terms of the state of the emulated device, and then resumes the OS with the emulated result. As you can imagine, certain types of devices rely on many such operations to configure and operate their hardware – this results in a dramatic slowdown in throughput.

## Para-Virtualized Devices

Para-virtualized mode comes to the rescue. By abstracting the logical operation of the device from its physical manifestation, we can significantly reduce the number of operations required to operate the device. Para-virtualization achieves this by creating a logical "ideal" device, and inviting the OSs to deal with that device, rather than the actual hardware.

Imagine a hard disk controller, for example. In the normal hardware case, the controller presents from dozens to hundreds of registers dealing with such diverse operations as timing, caching, SATA bus control, command requests and response queues, data queues, interrupt operation, memory management, block size, power control, and so on. In order to "talk" to the disk, and perform conceptually simple operations (such as "read block number 7") requires many register-based operations. In the virtualized model, this is slow (each individual register access causes the OS to be trapped, the VMM to emulate the operation, and the OS to be restarted). Moreover, the emulation software itself could be reasonably complicated – after all, it has to keep track of all state associated with each emulated register operation, and deal with sharing a simulated device whose underlying hardware implementation wasn't designed for sharing to begin with. Additional complexity equates to additional cost and bugs.

Consider instead, an ideal disk device (one designed by software people, rather than hardware people), where there's one register that contains the block number that we begin the read operation from, and another saying how many blocks to read. In this case, the OS simply writes the value "7" into the first register, and "1" into the second. The OS's device driver then expects the contents of one block, namely block 7, to show up in a buffer. In this manner, the only emulation that needs to take place is the emulation of this ideal device, rather than the much more complicated actual device.

There is a small, additional cost for para-virtualization – each OS must provide a native device driver that knows how to talk to this para-virtualized hardware must be written. Since the para-virtualized devices are idealized, this is much simpler than writing a driver for real-world hardware (the example of the ideal disk device above is only slightly simplified, but otherwise representative). Additionally, the interfaces presented by these ideal hardware devices are standardized, so there's only a limited number of such drivers to write. A popular framework is called VirtIO, and provides idealized hardware devices for disk storage, console access, network interfaces, and so on. VirtIO is supported by open source and commercial OSs alike.

### Hardware Acceleration

In situations where throughput is paramount (network adaptors, for example), hardware acceleration is also available. Recall from above that the main problem being solved by the VMM (when sharing peripherals) was the coordination of access to hardware from drivers in multiple, independent OSs. What if the hardware supported that directly? New technology in NICs (Network Interface Controller) presents multiple, independent channels within the hardware. Using such hardware, the VMM is responsible for assigning channels to OSs, and each OS then uses its dedicated channel in the NIC, without regards to the other OSs running concurrently.

## Conclusion

The virtual machine manager (VMM) or hypervisor has evolved considerably from the mainframe days of the mid 1960's. Just like an OS scheduling threads, the VMM schedules OSs, and each OS has a sandbox. Peripheral management can have a huge impact on overall throughput.